
IDAES Documentation

Release 1.9.0

IDAES team

Mar 04, 2021

CONTENTS

1	Project Goals	1
2	Collaborating institutions	3
3	Contact us	5
4	Contents	7
4.1	Getting Started	7
4.2	User Guide	11
4.3	Advanced User Guide	209
4.4	Tutorials and Examples	294
4.5	Technical Specifications	295
4.6	Related Packages	776
4.7	License	779
4.8	Copyright	780
5	Indices and tables	783
	Python Module Index	785
	Index	787

PROJECT GOALS

The Institute for the Design of Advanced Energy Systems (IDAES) will be the world's premier resource for the development and analysis of innovative advanced energy systems through the use of process systems engineering tools and approaches. IDAES and its capabilities will be applicable to the development of the full range of advanced fossil energy systems, including chemical looping and other transformational CO₂ capture technologies, as well as integration with other new technologies such as supercritical CO₂.

For a more detailed overview of the *IDAES integrated platform*, see [*this page*](#).

COLLABORATING INSTITUTIONS

The IDAES team is comprised of collaborators from the following institutions:

- National Energy Technology Laboratory (Lead)
- Sandia National Laboratory
- Lawrence Berkeley National Laboratory
- Carnegie-Mellon University (subcontract to LBNL)
- West Virginia University (subcontract to LBNL)
- University of Notre Dame (subcontract to LBNL)

CONTACT US

General, background and overview information is available at the [IDAES main website](#). Framework development happens at our [GitHub repo](#) where you can [report issues/bugs](#) or [make contributions](#). For further enquiries, send an email to: [<idaes-support@idaes.org>](mailto:idaes-support@idaes.org)

CONTENTS

4.1 Getting Started

4.1.1 Installation

To install the IDAES PSE framework, follow the set of instructions below that are appropriate for your needs and operating system. If you get stuck, please contact idaes-support@idaes.org.

After installing and testing IDAES, it is strongly recommended to do the IDAES tutorials located on the [examples online documentation page](#).

If you expect to develop custom models, we recommend following the *advanced user installation*.

The OS specific instructions provide information about optionally installing Miniconda. If you already have a Python installation you prefer, you can skip the Miniconda install section.

Note: IDAES supports Python 3.6 and above.

System	Section
Linux	<i>Linux</i>
Windows	<i>Windows</i>
Mac OSX	<i>Mac/OSX</i>
Generic	<i>Generic Install</i>

Warning: If you are using Python for other complex projects, you may want to consider using environments of some sort to avoid conflicting dependencies. There are several good options including conda environments if you use Anaconda.

4.1.2 Windows

Install Miniconda (optional)

1. Download: https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe
2. Install anaconda from the downloaded file in (1).
3. Open the Anaconda Prompt (Start -> “Anaconda Prompt”).
4. In the Anaconda Prompt, follow the *Generic Install* instructions.

4.1.3 Linux

Install Miniconda (optional)

1. Download: https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
2. Open a terminal window
3. Run the script you downloaded in (1).

Install Dependencies

1. The IPOPT solver depends on the GNU FORTRAN, GOMP, Blas, and Lapack libraries, If these libraries are not already installed on your Linux system, you or your system administrator can use the sample commands below to install them. If you have a Linux distribution that is not listed, IPOPT should still work, but the commands to install the required libraries may differ. If these libraries are already installed, you can skip this and proceed with the next step.

Note: Depending on your distribution, you may need to prepend `sudo` to these commands or switch to the “root” user.

apt-get (Current Ubuntu based distributions):

```
sudo apt-get install libgfortran4 libgomp1 liblapack3 libblas3
```

yum (Current RedHat based distributions, including CentOS):

```
yum install lapack blas libgfortran libgomp
```

Complete Generic Install

Follow the [Generic Install](#) instructions.

4.1.4 Mac/OSX

Install Miniconda (optional)

1. Download: https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
2. For the next steps, open a terminal window
3. Run the script you downloaded in (1).

Complete Generic Install

Follow the [Generic Install](#) instructions.

4.1.5 Generic Install

The remaining steps performed in either the Linux or OSX Terminal or Powershell. If you installed Miniconda on Windows use the Anaconda Prompt or Anaconda Powershell Prompt. Regardless of OS and shell, the following steps are the same.

Install IDAES

1. Install IDAES with pip by one of the following methods
 - a. To get the latest release:

```
pip install idaes-pse
```

- b. To get a specific release, for example 1.7:

```
pip install idaes-pse==1.7
```

- c. To get the latest development main branch:

```
pip install git+https://github.com/idaes/idaes-pse
```

- d. To get a specific fork or branch, for example myfork (of idaes-pse) and mybranch:

```
pip install git+https://github.com/myfork/idaes-pse@mybranch
```

2. Run the *idaes get-extensions command* to install the compiled binaries:

```
idaes get-extensions
```

Warning: The IDAES binary extensions are not yet supported on Mac/OSX.

As a fallback (assuming you are using a conda env) you can install the generic ipopt solver with the command `conda install -c conda-forge ipopt` though this will not have all the features of our extensions package.

3. Run the *idaes get-examples command* to download and install the example files:

```
idaes get-examples
```

By default this will install in a folder “examples” in the current directory. The command has many options, but an important one is `--dir`, which specifies the folder in which to install.

for Mac and Linux users this would look like:

```
idaes get-examples --dir ~/idaes/examples
```

or, for Windows users, it would look like:

```
idaes get-examples --dir C:\Users\MyName\IDAES\Examples
```

Refer to the full *idaes get-examples command documentation* for more information.

4. Run tests:

```
pytest --pyargs idaes -W ignore
```

5. You should see the tests run and all should pass to ensure the installation worked. You may see some “Error” level log messages, but they are okay, and produced by tests for error handling. The number of tests that failed and succeeded is reported at the end of the pytest output. You can report problems on the [Github issues page](#) (Please try to be specific about the command and the offending output.)

Install IDAES using Conda

As an alternative to the `pip install` method described above, IDAES can also be installed using the Conda package manager.

1. Create a new Conda environment with a name of your choice (in this example, `my-idaes-env`):

```
conda create --yes --name my-idaes-env python=3.8
```

Note: The `--yes` optional flag can be used to perform the installation without having it pausing to ask for confirmation.

2. Activate the `my-idaes-env` Conda environment:

```
conda activate my-idaes-env
```

Note: This step is needed when starting a new session or a new console tab/window.

3. Install the IDAES Conda package using the `conda install` subcommand:

```
conda install --yes -c IDAES-PSE -c conda-forge idaes-pse
```

Note: The most recent stable release will be selected by default. To instead select a particular version, specify the version tag using an `=` after the package name, e.g. `idaes-pse=1.9.0rc0`.

4. To complete the installation, follow the instructions described in the previous section from Step 2 (“Run the `idaes get-extensions` command...”)
onward.

4.1.6 Optional Dependencies

Some tools in IDAES may require additional dependencies. Instructions for installing these dependencies are located [here](#).

Optional Dependencies

(content will be added soon)

4.1.7 Updating an existing installation

When a new version is released, an IDAES installation can be updated without having to remove and reinstall it from scratch.

The following steps describe how to upgrade an existing installation in-place, assuming that the installation was done using one of the methods described earlier in this section.

Warning: If IDAES was installed in a dedicated environment (e.g. a Conda environment, or Python virtual environment), activate the environment before running any of these commands.

1. Open a terminal and verify the currently installed version of IDAES:

```
idaes --version
```

2. Install the upgraded version of the `idaes-pse` package using `pip install`:

```
pip install --upgrade idaes-pse
```

If a newer version of the `idaes-pse` package is available, the currently installed version will be removed and replaced by the newest available version. Check again the IDAES version to verify that the upgrade was successful:

```
idaes --version
```

3. Run the `idaes get-extension` command to install compiled binaries compatible with the newly upgraded IDAES version:

```
idaes get-extensions
```

4. Finally, use the `idaes get-examples` command to install the most recent version of the IDAES examples compatible with the upgraded IDAES version.

Warning: If the examples target installation directory is not empty, its contents, including examples installed with a previous IDAES version and other files, **will be overwritten without warning**. To avoid losing data, **it is strongly recommended that you make a backup copy of any existing examples directory** before proceeding.

After creating a backup copy of the existing examples directory, run:

```
idaes get-examples
```

4.2 User Guide

4.2.1 Why IDAES

The National Energy Technology Laboratory's Institute for the Design of Advanced Energy Systems (IDAES) is a powerful and versatile computational platform offering next-generation engineering capabilities for optimizing the design and operation of innovative chemical process and energy systems beyond current constraints on complexity, uncertainty, and scales ranging from materials to process to market.

The IDAES Integrated Platform was conceived in 2016 to specifically address the gaps between state-of-the-art simulation packages and algebraic modeling languages.

Major strengths of commercial simulation packages are their libraries of unit models and thermophysical properties. However, such simulation packages often have difficulty optimizing flowsheets and have limited support for incorporating models of non-standard, dynamic units, such as solids handling, and uncertainty quantification. On the other hand, AMLs are eminently flexible and readily support large-scale optimization, but considerable work is required to construct process models, which are often only useful for a one-time application.

The IDAES Integrated Platform represents an innovative approach for the design and optimization of chemical and energy processes by integrating an extensible, equation-oriented process model library with Pyomo (a Python-based AML). Built specifically to enable rigorous large-scale mathematical optimization, the platform includes capabilities for conceptual design, steady-state and dynamic optimization, multi-scale modeling, uncertainty quantification, and the automated development of thermodynamic, physical property, and kinetic sub-models from experimental data.

Key Features

Open Source

All IDAES Code is completely free and redistributable, the license is available [here](#). Users are free to modify and redistribute code, and community development is encouraged.

Equation Oriented

By using an equation-oriented platform, users gain access to a wide range of highly efficient, derivative-based numerical solvers for a wide range of problem types, including support for both linear and non-linear problems, ordinary and partial differential equations, and problems involving binary and integer variables.

Fully-Featured Programming Environment

By building off of Python, a fully-featured programming environment, users gain access to a wide range of libraries for tools such as data visualization and management.

Extensible

The source code for all models and tools is fully-open and visible to the user. This allows users to both see and understand what is happening in each model, but also modify and extend models to suit their needs.

Flexible Form

No single model form is best suited to all applications, thus the IDAES Integrated Platform is built to provide users with access to a range of different model forms. This allows users to easily pick-and-choose from the available model forms to find the one best suited to their particular application.

Access to Advanced Capabilities

IDAES aims to provide an integrated platform for development of not just process models but also tools for solving and analyzing these problems. The platform supports conceptual design, parameter estimation, model predictive control, uncertainty quantification, and surrogate modeling.

4.2.2 Concepts

This page gives a conceptual overview of the IDAES platform and provides terminology for the different components of that platform.

Contents

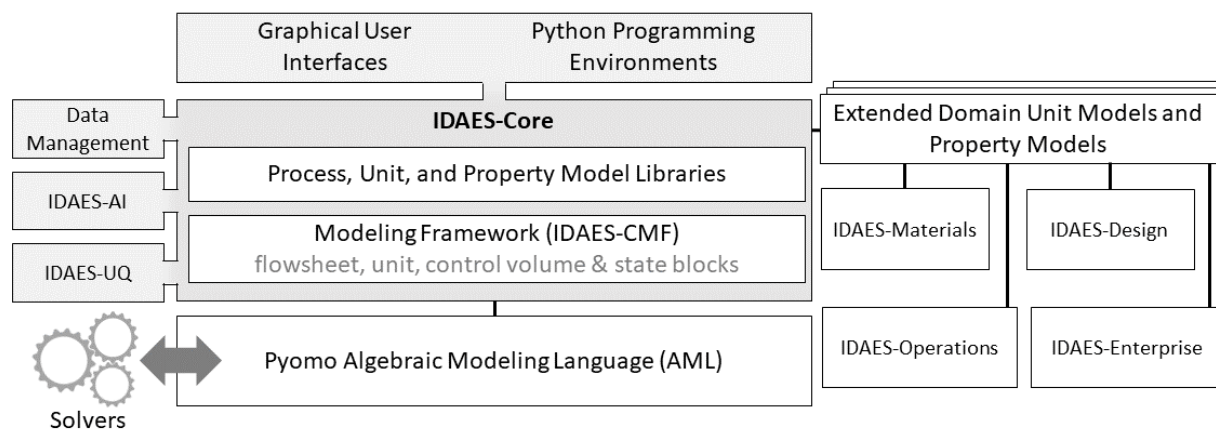
- *IDAES-IP*
 - *Description*
 - *Terminology*
- *IDAES-CMF*
 - *Overview*
 - *Modeling Components*
 - *Model Libraries*
 - *Modeling Extensions*

IDAES-IP

Description

The IDAES integrated platform (IDAES-IP) supports the full process modeling lifecycle from conceptual design to dynamic optimization and control within a single modeling environment. At the center of this platform is the Core Modeling Framework (*IDAES-CMF*) which leverages the open source, U.S. Department of Energy-funded extensible algebraic modeling environment, *Pyomo*.

Below is a diagram showing the components of the IDAES Integrated Platform (IDAES-IP).



Terminology

The following terms are used in the diagram above, and throughout this documentation.

IDAES-IP IDAES integrated platform, described on this page

IDAES-Core The software package that includes the Core Modeling Framework (IDAES-CMF); process, unit, and property model libraries; data management, artificial intelligence and uncertainty quantification tools; and graphical user interfaces.

IDAES-CMF The IDAES-CMF is the center of the IDAES-Core. It extends *Pyomo*'s block-based hierarchical modeling constructs to create a library of models for common process unit operations and thermophysical properties, along with a framework for the rapid development of process flowsheets.

IDAES-AI Artificial intelligence and machine learning tools

IDAES-UQ Tools supporting rigorous uncertainty quantification and optimization under uncertainty

Graphical User Interfaces Tools for graphical interactive work, such as visualization of IDAES flowsheets

Python programming environments Jupyter Notebook examples and extensions for interactive scripting in Python

Data Management (IDAES-DMF) Data Management Framework (DMF) supporting provenance for IDAES workflows

Pyomo Open source, U.S. Department of Energy-funded extensible algebraic modeling environment (AML). For more information, see the [Pyomo website](#).

IDAES-Materials

IDAES-Design

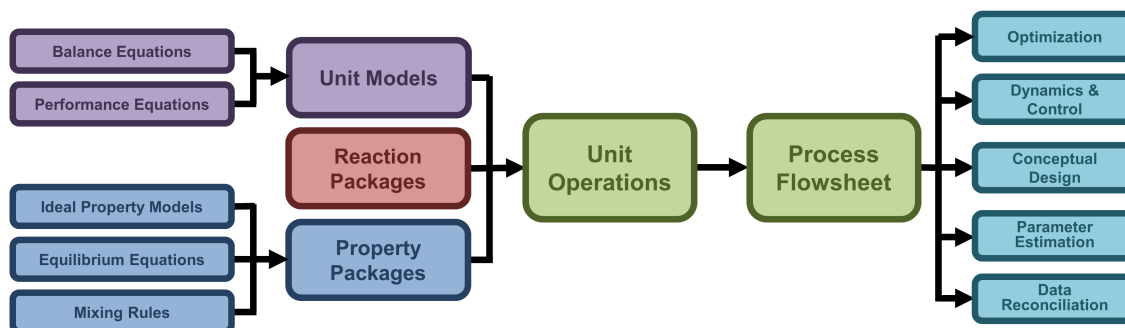
IDAES-Enterprise

IDAES-Operations Domain-specific tools for materials design, process design, enterprise-wide optimization, and control.

IDAES-CMF

Overview

The core modeling framework of IDAES was designed to be modular, and is based on the block-hierarchical structure shown below:



An IDAES process model begins with a process flowsheet, which is the canvas on which the representation of the user's process will be constructed. Each process consists of a network of unit operations which represent different pieces of equipment within the process (such as reactors, heater and pumps) and are connected together to form the overall process. Each unit operation in turn is made up of modular components – a unit model which describes the behavior and performance of the piece of equipment, a thermophysical property package which represents the material being processed by the unit operation, and a reaction package (if applicable) which represents any chemical reactions that may occur within the unit. Each of these components can be further broken down into sub-modules:

- Unit models consist of a set of material, energy and momentum balance equations which describe how material flows through the system, coupled with a set of performance equation which describe phenomena such as heat and mass transfer.

- Thermophysical property packages (generally) consist of a set of ideal, pure component properties for each component, a set of mixing rules and departure functions which describe how the mixture properties depend on the ideal properties, and a set of equations describing phase-equilibrium phenomena.

At the other end of the spectrum, IDAES process models are designed to be general purpose and to be applicable to a wide range of modeling activities. By providing access to a wide range of different numerical solvers and modeling tools, IDAES process models can be applied to a wide range of different problems, such as:

- process optimization and simulation of both steady-state and dynamic applications,
- data reconciliation,
- parameter estimation and uncertainty quantification,
- optimization under uncertainty, and
- conceptual design (superstructure problems).

Modeling Components

The IDAES Integrated Platform represents each level within the hierarchy above using “modeling components”. Each of these components represents a part of the overall model structure and form the basic building blocks of any IDAES process model. An introduction to each of the IDAES modeling components can be found [here](#).

Model Libraries

To provide a starting point for modelers in using the process modeling tools, the IDAES Integrated Platform contains a library of models for common unit operations and thermophysical properties. Modelers can use these out-of-the-box models to represent their process applications or as building blocks for developing their own models. All models within IDAES are designed to be fully open and extensible, allowing users to inspect and modify them to suit their needs. Documentation of the available model libraries can be found [here](#).

Modeling Extensions

The IDAES Integrated Platform also provides users with access to a number of cutting edge tools not directly related to process modeling. These tools are collected under the heading of Modeling Extensions, and information on them can be found [here](#).

4.2.3 Components

The purpose of this section of the documentation is to provide a general introduction to the top level components of the IDAES Integrated Platform. Each component is described in greater detail with a link in their description.

Note: IDAES is based on python-based algebraic modeling language, Pyomo. The documentation for its components (i.e. sets, parameters, variables, objectives, constraints, expressions, and suffixes) are provided in the [Pyomo documentation](#).

Flowsheet

Time Domain

Time domain is an essential component of the IDAES framework. When a user first declares a Flowsheet model a time domain is created, the form of which depends on whether the Flowsheet is declared to be dynamic or steady-state (see [FlowsheetBlock](#)). In situations where the user makes use of nested flowsheets, each sub-flowsheet refers to its parent flowsheet for the time domain.

Different models may handle the time domain differently, but in general all IDAES models refer to the time domain of their parent flowsheet. The only exception to this are blocks associated with Property calculations. PropertyBlocks (i.e. StateBlocks and ReactionBlocks) represent the state of the material at a single point in space and time, and thus do not contain the time domain. Instead, PropertyBlocks are indexed by time (and space where applicable) - i.e. there is a separate StateBlock for each point in time. The user should keep this in mind when working with IDAES models, as it is important for understanding where the time index appears within a model.

In order to facilitate referencing of the time domain, all Flowsheet objects have a *time* configuration argument which is a reference to the time domain for that flowsheet. All IDAES models contain a *flowsheet* method which returns the parent flowsheet object, thus a reference to the time domain can always be found using the following code: *flowsheet().config.time*.

Another important thing to note is that steady-state models do contain a time domain. While the time domain for steady-stage models is a single point at time = 0.0, they still contain a reference to the time domain and the components (e.g. StateBlocks) are indexed by time.

Flowsheet models are the top level of the IDAES modeling hierarchy. The flowsheet is implemented with a [FlowsheetBlock](#), which provides a container for other components. Flowsheet models generally contain three types of components:

1. *Unit models*, representing unit operations
2. *Property packages*, representing the parameters and relationships for property calculations
3. Arcs, representing connections between unit models

The FlowsheetBlock is also where the *time domain* is implemented. While the time domain is essential for dynamic modeling, the time domain exists even for steady state models (single point in time).

Flowsheet models may also contain additional constraints relating to how different unit models behave and interact, such as control and operational constraints. Generally speaking, if a constraint is purely internal to a single unit, and does not depend on information from other units in the flowsheet, then the constraint should be placed inside the relevant unit model. Otherwise, the constraint should be placed at the flowsheet level.

Property Package

- *Overview*
- *Units of Measurement*
- *Physical properties*
- *Reaction properties*
- *Component and Phase Objects*
- *As Needed Properties*
- *Generic Property Package Framework*

- *Generic Reaction Package Framework*

Overview

Component Object

Component objects are used to identify the chemical species of interest in a property package and to contain information describing the behavior of that component (such as properties of that component). Additional information on the *Component Class* is provided in the technical specifications.

The following types of components are currently supported.

- *Component* - general purpose object for representing chemical species.
- *Solute* - component object for representing species which should be treated as a solute in a *LiquidPhase*.
- *Solvent* - component object for representing species which should be treated as a solvent in a *LiquidPhase*.
- *Ion* - general purpose component object for representing ion species (*LiquidPhase* only). Users should generally use the *Anion* or *Cation* components instead.
- *Anion* - component object for representing ion species with a negative charge (*LiquidPhase* only).
- *Cation* - component object for representing ion species with a positive charge (*LiquidPhase* only).

Component objects are intended to store all the necessary information regarding a given chemical species for use within a process model. Examples of such information include the methods and parameters required for calculating thermophysical properties. Additionally, certain unit operations handle components in different ways depending on certain criteria. An example of this is Reverse Osmosis, where the driving force across the membrane is calculated differently for solvent species and solute species.

Component objects implement the following methods for determining species behavior:

- *is_solute()* - returns *True* if species is a solute (*Solute*, *Ion*, *Anion* or *Cation* component objects), otherwise *False*.
- *is_solvent()* - returns *True* if species is a solvent (*Solvent* component object), otherwise *False*.

Note: The general purpose *Component* object does not distinguish solutes and solvents, and these methods will raise a *TypeError* instead.

Phase Object

Phase objects are used to identify the thermodynamic phases of interest in a property package and to contain information describing the behavior of that phase (for example the equation of state which describes that phase). Additional information on the *Phase Class* is provided in the technical specifications.

The following types of phases, along with a generic Phase object, are supported:

- *LiquidPhase*
- *SolidPhase*
- *VaporPhase*

In a number of unit operations, different phases behave in different ways. For example, in a Flash operation, the vapor phase exits through the top outlet whilst liquid phase(s) (and any solids) exit through the bottom outlet. In order to

determine how a given phase should behave in these situations, each Phase object implements the following three methods:

- *is_liquid_phase()*
- *is_solid_phase()*
- *is_vapor_phase()*

These methods return a boolean (*True* or *False*) indicating whether the unit operation should treat the phase as being of the specified type in order to decide on how it should behave. Each type of phase returns *True* for its type and *False* for all other types (e.g. *LiquidPhase* returns *True* for *is_liquid_phase()* and *False* for *is_solid_phase()* and *is_vapor_phase()*).

The generic Phase object determines what to return for each method based on the user-provided name for the instance of the Phase object as shown below:

- *is_liquid_phase()* returns *True* if the Phase name contains the string *Liq*, otherwise it returns *False*.
- *is_solid_phase()* returns *True* if the Phase name contains the string *Sol*, otherwise it returns *False*.
- *is_vapor_phase()* returns *True* if the Phase name contains the string *Vap*, otherwise it returns *False*.

Users should avoid using the generic *Phase* object, as this is primarily intended as a base class for the specific phase classes and for backwards compatibility.

Physical Parameter Block

PhysicalParameterBlocks serve as a central location for linking to a property package, and contain all the parameters and indexing sets used by a given property package.

The role of the *PhysicalParameterBlock Class* is to set up the references required by the rest of the IDAES Core Modeling Framework for constructing instances of *StateBlocks* and attaching these to the PhysicalParameterBlock for ease of use. This allows other models to be pointed to the PhysicalParameterBlock in order to collect the necessary information and to construct the necessary StateBlocks without the need for the user to do this manually.

Several attributes in the PhysicalParameterBlock are used to inform the construction of other components. These attributes include:

- *state_block_class* - a pointer to the associated class that should be called when constructing StateBlocks. This should only be set by the property package developer.
- *phase_list* - a Pyomo Set object defining the valid phases of the mixture of interest.
- *component_list* - a Pyomo Set defining the names of the chemical species present in the mixture.
- *element_list* - (optional) a Pyomo Set defining the names of the chemical elements that make up the species within the mixture. This is used when doing elemental material balances.
- *element_comp* - (optional) a dictionary-like object which defines the elemental composition of each species in *component_list*. Form: *component*: {*element_1*: value, *element_2*: value, ... }.
- supported properties metadata - a dictionary of supported physical properties that the property package supports, along with instruction to construct the associated variables and constraints, and the units of measurement used for the property. This information is set using the *add_properties* attribute of the *define_metadata* class method.

Reaction Block

ReactionBlocks are used within IDAES UnitModels (generally within ControlVolumeBlocks) in order to calculate reaction properties given the state of the material (provided by an associated StateBlock). ReactionBlocks are notably different to other types of Blocks within IDAES as they are always indexed by time (and possibly space as well), and are also not fully self contained (in that they depend upon the associated state block for certain variables). ReactionBlocks are composed of two parts:

- ReactionBlockDataBase forms the base class for all ReactionBlockData objects, which contain the instructions on how to construct each instance of a Reaction Block.
- ReactionBlockBase is used for building classes which contain methods to be applied to sets of Indexed Reaction Blocks (or to a subset of these). See the documentation on *declare_process_block_class* and the IDAES tutorials and examples for more information.

ReactionBlocks can be constructed directly from the associated ReactionParameterBlock by calling the *build_reaction_block()* method on the ReactionParameterBlock. The *parameters* construction argument will be automatically set, and any other arguments (including indexing sets) may be provided to the *build_reaction_block* method as usual.

Additional details on *ReactionBlocks* are located in the technical specifications.

Reaction Parameter Block

ReactionParameterBlocks serve as a central location for linking to a property package, and contain all the parameters and indexing sets used by a given property package.

The role of the *ReactionParameterBlock Class* is to set up the references required by the rest of the IDAES framework for constructing instances of *ReactionBlocks* and attaching these to the ReactionParameterBlock for ease of use. This allows other models to be pointed to the ReactionParameterBlock in order to collect the necessary information and to construct the necessary ReactionBlocks without the need for the user to do this manually.

Reaction property packages are used by all of the other modeling components to inform them of what needs to be constructed when dealing with chemical reactions. In order to do this, the IDAES modeling framework looks for a number of attributes in the ReactionParameterBlock which are used to inform the construction of other components. These attributes include:

- *reaction_block_class* - a pointer to the associated class that should be called when constructing ReactionBlocks. This should only be set by the property package developer.
- *phase_list* - a Pyomo Set object defining the valid phases of the mixture of interest.
- *component_list* - a Pyomo Set defining the names of the chemical species present in the mixture.
- *rate_reaction_idx* - a Pyomo Set defining a list of names for the kinetically controlled reactions of interest.
- *rate_reaction_stoichiometry* - a dict-like object defining the stoichiometry of the kinetically controlled reactions. Keys should be tuples of (*rate_reaction_idx*, *phase_list*, *component_list*) and values equal to the stoichiometric coefficient for that index.
- *equilibrium_reaction_idx* - a Pyomo Set defining a list of names for the equilibrium controlled reactions of interest.
- *equilibrium_reaction_stoichiometry* - a dict-like object defining the stoichiometry of the equilibrium controlled reactions. Keys should be tuples of (*equilibrium_reaction_idx*, *phase_list*, *component_list*) and values equal to the stoichiometric coefficient for that index.
- supported properties metadata - a list of supported reaction properties that the property package supports, along with instruction to construct the associated variables and constraints, and the units of measurement used for the property. This information is set using the *add_properties* attribute of the *define_metadata* class method.

- required properties metadata - a list of physical properties that the reaction property calculations depend upon, and must be supported by the associated StateBlock. This information is set using the `add_required_properties` attribute of the `define_metadata` class method.

State Block

StateBlocks are used within all IDAES UnitModels (generally within ControlVolumeBlocks) in order to calculate physical properties given the state of the material. StateBlocks are notably different to other types of Blocks within IDAES as they are always indexed by time (and possibly space as well). StateBlocks consist of two parts:

- StateBlockData forms the base class for all StateBlockData objects, which contain the instructions on how to construct each instance of a State Block.
- StateBlock is used for building classes which contain methods to be applied to sets of Indexed State Blocks (or to a subset of these). See the documentation on `declare_process_block_class` and the IDAES tutorials and examples for more information.

StateBlocks can be constructed directly from the associated PhysicalParameterBlock by calling the `build_state_block()` method on the PhysicalParameterBlock. The *parameters* construction argument will be automatically set, and any other arguments (including indexing sets) may be provided to the `build_state_block` method as usual.

Additional details on *State Blocks* are located in the technical specifications.

Defining Units of Measurement

All property packages within IDAES are expected to define a metadata class as part of the package's ParameterBlock, which amongst other things contains a definition of the base units of measurement used by that property package. An example of defining the default units for a property package is shown below.

```
from pyomo.environ import units

@classmethod
def define_metadata(cls, obj):
    obj.add_default_units({'time': units.s,
                          'length': units.m,
                          'mass': units.kg,
                          'amount': units.mol,
                          'temperature': units.K})
```

Each property package should define a default units for 7 base quantities listed below:

- time
- length
- mass
- amount of substance
- temperature
- current (optional)
- luminous intensity (optional)

Units must be defined using Pyomo's Units container (*from pyomo.environ import units*), and all quantities within the property package must be based on the chosen set of base units. Parameters and correlations may be based on different sets of unit as necessary (e.g. from literature sources using different base units), however the final quantity must be converted to the set of base units defined in the metadata.

Generic Property Package Framework

Contents

Defining Property Packages

Contents

- *Defining Property Packages*
 - *Introduction*
 - *Units of Measurement*
 - *Property Parameters*
 - *Config Dictionary*
 - *Class Definition*
 - *Examples*

Introduction

In order to create and use a property package using the IDAES Generic Property Package Framework, users must provide a definition for the material they wish to model. The framework supports two approaches for defining the property package, which are described below, both of which are equivalent in practice.

Units of Measurement

When defining a property package using the generic framework, users must define the base units for the property package (see [link](#)). The approach for setting the base units depends on the approach used to define the property package, and is discussed in more detail in each section.

The Generic Property Package Framework includes the necessary code to convert between different units of measurement as required, allowing users to combine property methods with different sets of units into a single property package. In these cases, each property method is written in its natural units (including parameters), and the final result is automatically converted to the base units.

For example, the Antoine equation is generally written with pressure in bars and temperature in either Kelvin or Celsius (depending on source). Using the generic property framework, the users provide the Antoine coefficients in their original units (i.e. bar and Kelvin/Celsius) and the property calculation is written in these units. However, the final result (saturation pressure) is then converted to the base units specified in the property package definition.

Property Parameters

Thermophysical property models all depend upon a set of parameters to describe the fundamental behavior of the system. For the purposes of the Generic Property Framework, these parameters are grouped into three types:

1. Component-specific parameters - these are parameters that are specific to a given chemical species, and are defined in the *parameter_data* argument for each component and stored in the associated *Component* block. Examples of these parameters include those used to calculate the ideal, pure component properties.
2. Phase-specific parameters - these are parameters that are specific to a given phase, and are defined in the *parameter_data* argument for each phase and stored in the associated *Phase* block. These types of parameters are relatively uncommon.
3. Package-wide parameters - these are parameters that are not necessarily confined to a single phase or species, and are defined in the *parameter_data* argument of the overall property package and stored in the main *Physical Parameter* block. Examples of these types of parameters include binary interaction parameters, which involve multiple species and can be used in multiple phases.

Config Dictionary

The most common way to use the Generic Property Package Framework is to create an instance of the *GenericParameterBlock* component and provide it with a dictionary of configuration arguments, as shown below:

```
m = ConcreteModel()

m.fs = FlowsheetBlock()

m.fs.properties = GenericParameterBlock(default=config_dict)
```

Users need to populate *config_dict* with the desired options for their system as described in the other parts of this documentation. An example of a configuration dictionary for a benzene-toluene VLE system is shown below.

Using this approach, units of measurement are defined using the *base_units* option in the configuration dictionary. Users must provide units for the 5 core quantities, and may also provide units for the other 2 SI base quantities (if required). For details on other configuration options, please see the relevant documentation.

```
from pyomo.environ import units as pyunits

config_dict = {
    "base_units": {"time": pyunits.s,
                  "length": pyunits.m,
                  "mass": pyunits.kg,
                  "amount": pyunits.mol,
                  "temperature": pyunits.K},
    "components": {
        'benzene': {"type": Component,
                    "elemental_composition": {"C": 6, "H": 6},
                    "dens_mol_liq_comp": Perrys,
                    "enth_mol_liq_comp": Perrys,
                    "enth_mol_ig_comp": RPP,
                    "pressure_sat_comp": RPP,
                    "phase_equilibrium_form": {("Vap", "Liq"): fugacity},
                    "parameter_data": {
                        "mw": (78.1136E-3, pyunits.kg/pyunits.mol), # [1]
                        "pressure_crit": (48.9e5, pyunits.Pa), # [1]
                        "temperature_crit": (562.2, pyunits.K), # [1]
```

(continues on next page)

(continued from previous page)

```

"dens_mol_liq_comp_coeff": {
    '1': (1.0162, pyunits.kmol*pyunits.m**-3), # [2] pg. 2-98
    '2': (0.2655, None),
    '3': (562.16, pyunits.K),
    '4': (0.28212, None)},
"cp_mol_ig_comp_coeff": {
    'A': (-3.392E1, pyunits.J/pyunits.mol/pyunits.K), # [1]
    'B': (4.739E-1, pyunits.J/pyunits.mol/pyunits.K**2),
    'C': (-3.017E-4, pyunits.J/pyunits.mol/pyunits.K**3),
    'D': (7.130E-8, pyunits.J/pyunits.mol/pyunits.K**4)},
"cp_mol_liq_comp_coeff": {
    '1': (1.29E2, pyunits.J/pyunits.kmol/pyunits.K), # [2]
    '2': (-1.7E-1, pyunits.J/pyunits.kmol/pyunits.K**2),
    '3': (6.48E-4, pyunits.J/pyunits.kmol/pyunits.K**3),
    '4': (0, pyunits.J/pyunits.kmol/pyunits.K**4),
    '5': (0, pyunits.J/pyunits.kmol/pyunits.K**5)},
"enth_mol_form_liq_comp_ref": (
    49.0e3, pyunits.J/pyunits.mol), # [3]
"enth_mol_form_vap_comp_ref": (
    82.9e3, pyunits.J/pyunits.mol), # [3]
"pressure_sat_comp_coeff": {'A': (-6.98273, None), # [1]
                             'B': (1.33213, None),
                             'C': (-2.62863, None),
                             'D': (-3.33399, None)}},
'toluene': {"type": Component,
"elemental_composition": {"C": 7, "H": 8},
"dens_mol_liq_comp": Perrys,
"enth_mol_liq_comp": Perrys,
"enth_mol_ig_comp": RPP,
"pressure_sat_comp": RPP,
"phase_equilibrium_form": {("Vap", "Liq"): fugacity},
"parameter_data": {
    "mw": (92.1405E-3, pyunits.kg/pyunits.mol), # [1]
    "pressure_crit": (41e5, pyunits.Pa), # [1]
    "temperature_crit": (591.8, pyunits.K), # [1]
    "dens_mol_liq_comp_coeff": {
        '1': (0.8488, pyunits.kmol*pyunits.m**-3), # [2] pg. 2-98
        '2': (0.26655, None),
        '3': (591.8, pyunits.K),
        '4': (0.2878, None)},
    "cp_mol_ig_comp_coeff": {
        'A': (-2.435E1, pyunits.J/pyunits.mol/pyunits.K), # [1]
        'B': (5.125E-1, pyunits.J/pyunits.mol/pyunits.K**2),
        'C': (-2.765E-4, pyunits.J/pyunits.mol/pyunits.K**3),
        'D': (4.911E-8, pyunits.J/pyunits.mol/pyunits.K**4)},
    "cp_mol_liq_comp_coeff": {
        '1': (1.40E2, pyunits.J/pyunits.kmol/pyunits.K), # [2]
        '2': (-1.52E-1, pyunits.J/pyunits.kmol/pyunits.K**2),
        '3': (6.95E-4, pyunits.J/pyunits.kmol/pyunits.K**3),
        '4': (0, pyunits.J/pyunits.kmol/pyunits.K**4),
        '5': (0, pyunits.J/pyunits.kmol/pyunits.K**5)},
    "enth_mol_form_liq_comp_ref": (
        12.0e3, pyunits.J/pyunits.mol), # [3]
    "enth_mol_form_vap_comp_ref": (
        50.1e3, pyunits.J/pyunits.mol), # [3]
    "pressure_sat_comp_coeff": {'A': (-7.28607, None), # [1]
                                'B': (1.38091, None),

```

(continues on next page)

(continued from previous page)

```

        'C': (-2.83433, None),
        'D': (-2.79168, None) } } } },

    "phases": { 'Liq': { "type": LiquidPhase,
                        "equation_of_state": ideal },
                'Vap': { "type": VaporPhase,
                        "equation_of_state": ideal } },

    "state_definition": FcPh,
    "state_bounds": {
        # Note format is (lower, nominal, upper, units)
        "flow_mol": (0, 100, 1000, pyunits.mol/pyunits.s),
        "temperature": (273.15, 300, 450, pyunits.K),
        "pressure": (5e4, 1e5, 1e6, pyunits.Pa)},
    "pressure_ref": (1e5, pyunits.Pa),
    "temperature_ref": (300, pyunits.K),
    "phases_in_equilibrium": [ ("Vap", "Liq") ],
    "phase_equilibrium_state": { ("Vap", "Liq"): smooth_VLE },
    "bubble_dew_method": IdealBubbleDew

```

Data Sources:

1. The Properties of Gases and Liquids (1987), 4th edition, Chemical Engineering Series - Robert C. Reid
2. Perry's Chemical Engineers' Handbook 7th Ed.
3. Engineering Toolbox, <https://www.engineeringtoolbox.com>, Retrieved 1st December, 2019

Class Definition

Alternatively, the IDAES Generic Property Package Framework supports defining classes derived from the IDAES *GenericParameterData* with methods for defining configuration options and parameters.

Users can define two methods which are called automatically when an instance of the property package is created:

1. *configure*, which defines the users selection of sub-models, and
2. *parameters*, which defines the parameters necessary for the selected property methods.

A basic outline of a user defined Property Parameter Block is shown below.

```

@declare_process_block_class("UserParameterBlock")
class UserParameterData(GenericParameterData):
    def configure(self):
        # Set configuration options
        self.config.option_1 = value

    def parameters(self):
        # Define parameters
        self.param_1 = Var(index_set, initialize=value)

```

Users should populate the *configure* and *parameters* methods as discussed below.

Configure

The `configure` method is used to assign values to the configuration arguments, using the format `self.config.option_name = value`. Users will also need to set the units of measurement in the property package metadata.

Parameters

The `parameters` method is used to construct all the parameters associated with the property calculations and to specify values for these. The list of necessary parameters is based on the configuration options and the selected methods. Each method lists their necessary parameters in their documentation. Users need only define those parameters required by the options they have chosen.

Examples

Examples of using the IDAES Generic Property Package Framework can be found in the `idaes/property_models/core/examples` folder.

Defining Components

The first step in defining a generic property package is to describe each of the chemical species of interest within the system, including methods for calculating the necessary thermophysical properties of the pure component. Components are defined using *IDAES Component objects*, and are automatically constructed using the `components` configuration argument from the *GenericParameterBlock*.

The `components` Argument

Each *GenericParameterBlock* has a configuration argument named `components` which is used to construct the *Component* objects and populate them with instructions on how to calculate thermophysical properties for that component. The `components` configuration argument is expected to be a dict-of-dicts, where the keys are the names for the chemical species of interest, and the values are a dict of configuration arguments for the named component (which are passed to the *Component* object as it is instantiated).

```
"components": {
    "species_1": {options},
    "species_2": {options}}
```

Configuration Arguments

The configuration arguments for each chemical species are used to define methods for calculating pure component properties and defining the parameters associated with these. A full list of the supported configuration arguments for *Component* objects can be found [here](#).

Type Argument

Each component in the *component* argument must be assigned a valid component type from those supported by the IDAES Framework (e.g. Component, Solvent, Solute, etc.). This should be provided using the *type* argument.

Valid Phases

In many cases, a given chemical species can only exist in certain phases; the most common example being ionic solids which dissociate upon dissolution (thus forming new ionic species in an aqueous phase). For each component, the user can set a list of the valid phase types for the component (liquid, vapor and/or solid) using the *valid_phase_types* configuration argument. This configuration argument should be a list containing *PhaseType Enums* (imported from *idaes.core.phases*) indicating the types of phases in which this component can exist.

This information is used by the Generic Property Framework to automatically determine the valid phase-component pairs for the user defined system. Users can override this automatic definition by providing a component list for a given phase in the definition of each *Phase* as discussed later (note however that user-defined phase-component lists are validated against the valid phases, and an exception will be raised if a component is assigned in a phase for which it is not valid).

Elemental Composition

If a user wishes to use elemental balances as part of their flowsheet (e.g. a Gibbs equilibrium reactor), it is necessary to specify the elemental composition of each Component. This can be done using the *elemental_composition* configuration argument, which takes a dictionary where the keys are the constituent elements and the values are the number of atoms of that element which compose the Components.

```
"components": {  
  "water": {"elemental_composition": {"H": 2, "O": 1}}}
```

If users specify an elemental composition for one Component, they must specify elemental compositions for all Components. The Generic Property Package framework will then compile the list of elements composing all species and the overall composition matrix automatically.

Pure Component Property Methods

Most methods for calculating the thermophysical properties of materials start from estimating the properties of each component in its pure form, before applying mixing rules to determine the properties of the mixture. Pure component properties generally take the form of empirical correlations as a function of material state (generally temperature) derived from experimental data. Data and correlations for many components are readily available in literature. However due to the empirical nature of these correlations and the wide range of data available, different sources use different forms for their correlations.

Within the IDAES Generic Property Package Framework, pure component property correlations can be provided as either Python functions or classes;

- functions are used for self-contained correlations with hard-coded parameters,
- classes are used for more generic correlations which require associated parameters.

When providing a method via the *components* configuration argument, users can either provide a pointer to the desired class/method directly, or to a Python module containing a class or method with the same name as the property to be calculated. More details on the uses of these and how to construct your own can be found in the [developer documentation](#).

Pure Component Libraries

As a starting point for users, the IDAES Generic Property Package Framework contains a library of some common methods for calculating properties of interest. These libraries are organized by source, and are listed below.

Note: Users should be careful about mixing-and-matching methods from different libraries, especially for the same component. Thermodynamic properties are intrinsically coupled, thus many correlations are also linked and often share parameters. Mixing-and-matching correlations may result in two correlations using parameters with the same name but with different expectations.

Additionally, sources often use different approaches for defining the thermodynamic reference state of the material, thus users need to ensure that a consistent reference state is being used when combining methods from different sources.

NIST Webbook (NIST)

Contents

- *NIST Webbook (NIST)*
 - *Source*
 - *Ideal Gas Molar Heat Capacity (Constant Pressure)*
 - *Ideal Gas Molar Enthalpy*
 - *Ideal Gas Molar Entropy*
 - *Saturation (Vapor) Pressure*

Source

Pure component properties as used by the NIST WebBook, <https://webbook.nist.gov/chemistry/> Retrieved: September 13th, 2019

Ideal Gas Molar Heat Capacity (Constant Pressure)

NIST uses the Shomate equation for the ideal gas molar heat capacity, which is shown below:

$$c_{p\text{ ig}} = A + B \times t + C \times t^2 + D \times t^3 + \frac{E}{t^2}$$

where $t = \frac{T}{1000}$. Units are J/mol·K.

Parameters

Symbol	Parameter Name	Units	Description
<i>A</i>	cp_mol_ig_comp_coeff_A	J/mol·K	
<i>B</i>	cp_mol_ig_comp_coeff_B	J/mol·K·kK	
<i>C</i>	cp_mol_ig_comp_coeff_C	J/mol·K·kK ²	
<i>D</i>	cp_mol_ig_comp_coeff_D	J/mol·K·kK ³	
<i>E</i>	cp_mol_ig_comp_coeff_E	J·kK ² /mol·K	
<i>F</i>	cp_mol_ig_comp_coeff_F	kJ/mol	
<i>G</i>	cp_mol_ig_comp_coeff_G	J/mol·K	
<i>H</i>	cp_mol_ig_comp_coeff_H	kJ/mol	

Note: Due to the division of temperature by 1000 in the expression form, most temperature units are in kilo-Kelvins and reference enthalpies (*F* and *H*) are in kJ/mol. The parameter *cp_mol_ig_comp_coeff* is also used when calculating specific enthalpy and entropy and parameters ‘*F*’, ‘*G*’ and ‘*H*’ are only required for these properties.

Ideal Gas Molar Enthalpy

The correlation for the ideal gas molar enthalpy is derived from the correlation for the molar heat capacity and is given below:

$$\frac{h_{ig} - h_{ig,ref}}{1000} = A \times (t - t_{ref}) + \frac{B}{2} \times (t^2 - t_{ref}^2) + \frac{C}{3} \times (t^3 - t_{ref}^3) + \frac{D}{4} \times (t^4 - t_{ref}^4) + E \times \left(\frac{1}{t} - \frac{1}{t_{ref}}\right) + F - H$$

Units are J/mol.

Symbol	Parameter Name	Units	Description
<i>A</i>	cp_mol_ig_comp_coeff_A	J/mol·K	
<i>B</i>	cp_mol_ig_comp_coeff_B	J/mol·K·kK	
<i>C</i>	cp_mol_ig_comp_coeff_C	J/mol·K·kK ²	
<i>D</i>	cp_mol_ig_comp_coeff_D	J/mol·K·kK ³	
<i>E</i>	cp_mol_ig_comp_coeff_E	J·kK ² /mol·K	
<i>F</i>	cp_mol_ig_comp_coeff_F	kJ/mol	
<i>G</i>	cp_mol_ig_comp_coeff_G	J/mol·K	
<i>H</i>	cp_mol_ig_comp_coeff_H	kJ/mol	

Note: This correlation uses the same parameters as for the ideal gas heat capacity with additional parameters *F* and *H*. These parameters account for the enthalpy at the reference state defined by NIST, where *F* is the constant of integration and *H* is the standard molar heat of formation. Note that the default form of the expression used by NIST subtracts the heat of formation from the specific enthalpy. This behavior can be controlled using the global configuration argument *include_enthalpy_of_formation* - if this is set to *True* (the default setting), then the *H* term is not used when calculating specific enthalpies. Due to the division of temperature by 1000 in the expression form, most temperature units are in kilo-Kelvins and reference enthalpies (*F* and *H*) are in kJ/mol.

Ideal Gas Molar Entropy

The correlation for the ideal gas molar entropy is derived from the correlation for the molar heat capacity and is given below:

$$s_{ig} = A \times \ln(t) + B \times t + \frac{C}{2} \times t^2 + \frac{D}{3} \times t^3 + \frac{E}{2 \times t^2} + G$$

Units are J/mol·K.

Symbol	Parameter Name	Units	Description
<i>A</i>	cp_mol_ig_comp_coeff_A	J/mol·K	
<i>B</i>	cp_mol_ig_comp_coeff_B	J/mol·K·kK	
<i>C</i>	cp_mol_ig_comp_coeff_C	J/mol·K·kK ²	
<i>D</i>	cp_mol_ig_comp_coeff_D	J/mol·K·kK ³	
<i>E</i>	cp_mol_ig_comp_coeff_E	J·kK ² /mol·K	
<i>F</i>	cp_mol_ig_comp_coeff_F	kJ/mol	
<i>G</i>	cp_mol_ig_comp_coeff_G	J/mol·K	
<i>H</i>	cp_mol_ig_comp_coeff_H	kJ/mol	

Note: This correlation uses the same parameters as for the ideal gas heat capacity with additional parameter *G*, which accounts for the standard entropy at the reference state defined by NIST. Users wanting to use a different reference state will need to update *G*. Due to the division of temperature by 1000 in the expression form, most temperature units are in kilo-Kelvins and reference enthalpies (*F* and *H*) are in kJ/mol.

Saturation (Vapor) Pressure

NIST uses the Antoine equation to calculate the vapor pressure of a component, which is given below:

$$\log_{10}(P_{sat}) = A - \frac{B}{T + C}$$

Units are bar and Kelvin.

Parameters

Symbol	Parameter Name	Units	Description
<i>A</i>	pressure_sat_comp_coeff_A	None	
<i>B</i>	pressure_sat_comp_coeff_B	K	
<i>C</i>	pressure_sat_comp_coeff_C	K	

Perry's Chemical Engineers' Handbook (Perrys)

Contents

- *Perry's Chemical Engineers' Handbook (Perrys)*
 - *Source*
 - *Ideal Liquid Molar Heat Capacity (Constant Pressure)*

- *Ideal Liquid Molar Enthalpy*
- *Ideal Liquid Molar Entropy*
- *Liquid Molar Density*

Source

Methods for calculating pure component properties from:

Perry's Chemical Engineers' Handbook, 7th Edition Perry, Green, Maloney, 1997, McGraw-Hill

Ideal Liquid Molar Heat Capacity (Constant Pressure)

Perry's Handbook uses the following correlation for ideal liquid molar heat capacity:

$$c_{p \text{ liq}} = C_1 + C_2 \times T + C_3 \times T^2 + C_4 \times T^3 + C_5 \times T^4$$

Units are J/kmol·K.

Parameters

Symbol	Parameter Name	Units	Description
C_1	cp_mol_ig_comp_coeff_1	J/kmol·K	
C_2	cp_mol_ig_comp_coeff_2	J/kmol·K ²	
C_3	cp_mol_ig_comp_coeff_3	J/kmol·K ³	
C_4	cp_mol_ig_comp_coeff_4	J/kmol·K ⁴	
C_5	cp_mol_ig_comp_coeff_5	J/kmol·K ⁵	

Ideal Liquid Molar Enthalpy

The correlation for the ideal liquid molar enthalpy is derived from the correlation for the molar heat capacity and is given below:

$$h_{\text{liq}} - h_{\text{liq ref}} = C_1 \times (T - T_{\text{ref}}) + \frac{C_2}{2} \times (T^2 - T_{\text{ref}}^2) + \frac{C_3}{3} \times (T^3 - T_{\text{ref}}^3) + \frac{C_4}{4} \times (T^4 - T_{\text{ref}}^4) + \frac{C_5}{5} \times (T^5 - T_{\text{ref}}^5) + \Delta h_{\text{form, Liq}}$$

Units are J/kmol.

Parameters

Symbol	Parameter Name	Units	Description
C_1	cp_mol_ig_comp_coeff_1	J/kmol·K	
C_2	cp_mol_ig_comp_coeff_2	J/kmol·K ²	
C_3	cp_mol_ig_comp_coeff_3	J/kmol·K ³	
C_4	cp_mol_ig_comp_coeff_4	J/kmol·K ⁴	
C_5	cp_mol_ig_comp_coeff_5	J/kmol·K ⁵	
$\Delta h_{\text{form, Liq}}$	enth_mol_form_liq_comp_ref	Base units	Molar heat of formation at reference state

Note: This correlation uses the same parameters as the ideal liquid heat capacity. Units of molar heat of formation will be derived from the base units defined for the property package.

Ideal Liquid Molar Entropy

The correlation for the ideal liquid molar entropy is derived from the correlation for the molar heat capacity and is given below:

$$s_{\text{liq}} - s_{\text{liq ref}} = C_1 \times \ln(T/T_{\text{ref}}) + C_2 \times (T - T_{\text{ref}}) + \frac{C_3}{2} \times (T^2 - T_{\text{ref}}^2) + \frac{C_4}{3} \times (T^3 - T_{\text{ref}}^3) + \frac{C_5}{4} \times (T^4 - T_{\text{ref}}^4) + s_{\text{form, Liq}}$$

Units are J/kmol·K.

Parameters

Symbol	Parameter Name	Units	Description
C_1	cp_mol_ig_comp_coeff_1	J/kmol·K	
C_2	cp_mol_ig_comp_coeff_2	J/kmol·K ²	
C_3	cp_mol_ig_comp_coeff_3	J/kmol·K ³	
C_4	cp_mol_ig_comp_coeff_4	J/kmol·K ⁴	
C_5	cp_mol_ig_comp_coeff_5	J/kmol·K ⁵	
$s_{\text{form, Liq}}$	entr_mol_form_liq_comp_ref	Base units	Standard molar entropy of formation at reference state

Note: This correlation uses the same parameters as the ideal liquid heat capacity. Units of molar entropy of formation will be derived from the base units defined for the property package.

Liquid Molar Density

Perry's Handbook uses the following correlation for liquid molar density:

$$\rho_{\text{liq}} = \frac{C_1}{C_2^{1+(1-\frac{T}{C_3})^{C_4}}}$$

Units are kmol/m³.

Parameters

Symbol	Parameter Name	Units	Description
C_1	dens_mol_comp_liq_coeff_1	kmol/m ³	
C_2	dens_mol_comp_liq_coeff_2	None	
C_3	dens_mol_comp_liq_coeff_3	K	
C_4	dens_mol_comp_liq_coeff_4	None`	

Note: Currently, only the most common correlation form from Perry's Handbook is implemented. Some components use different forms which are not yet supported.

Properties of Gases and Liquids 3rd edition (RPP3)

Contents

- *Properties of Gases and Liquids 3rd edition (RPP3)*
 - *Source*
 - *Ideal Gas Molar Heat Capacity (Constant Pressure)*
 - *Ideal Gas Molar Enthalpy*
 - *Ideal Gas Molar Entropy*
 - *Saturation (Vapor) Pressure*

Source

Methods for calculating pure component properties from:

The Properties of Gases & Liquids, 3rd Edition Reid, Prausnitz and Polling, 1977, McGraw-Hill

Ideal Gas Molar Heat Capacity (Constant Pressure)

Properties of Gases and Liquids uses the following correlation for the ideal gas molar heat capacity:

$$c_{p\text{ ig}} = A + B \times T + C \times T^2 + D \times T^3$$

Units are calories per gram-mole kelvin and Kelvin.

Parameters

Symbol	Parameter Name	Units	Description
A	cp_mol_ig_comp_coeff_A	cal/mol·K	
B	cp_mol_ig_comp_coeff_B	cal/mol·K ²	
C	cp_mol_ig_comp_coeff_C	cal/mol·K ³	
D	cp_mol_ig_comp_coeff_D	cal/mol·K ⁴	

Ideal Gas Molar Enthalpy

The correlation for the ideal gas molar enthalpy is derived from the correlation for the molar heat capacity and is given below:

$$h_{\text{ig}} - h_{\text{ig ref}} = A \times (T - T_{\text{ref}}) + \frac{B}{2} \times (T^2 - T_{\text{ref}}^2) + \frac{C}{3} \times (T^3 - T_{\text{ref}}^3) + \frac{D}{4} \times (T^4 - T_{\text{ref}}^4) + \Delta h_{\text{form, Vap}}$$

Units are calories per gram-mole kelvin and Kelvin.

Parameters

Symbol	Parameter Name	Units	Description
A	cp_mol_ig_comp_coeff_A	cal/mol·K	
B	cp_mol_ig_comp_coeff_B	cal/mol·K ²	
C	cp_mol_ig_comp_coeff_C	cal/mol·K ³	
D	cp_mol_ig_comp_coeff_D	cal/mol·K ⁴	
$\Delta h_{\text{form, Vap}}$	enth_mol_form_vap_comp_ref	Base units	Molar heat of formation at reference state

Note: This correlation uses the same parameters as the ideal gas heat capacity correlation. Units of molar heat of formation will be derived from the base units defined for the property package.

Ideal Gas Molar Entropy

The correlation for the ideal gas molar entropy is derived from the correlation for the molar heat capacity and is given below:

$$s_{\text{ig}} = A \times \ln(T/T_{\text{ref}}) + B \times (T - T_{\text{ref}}) + \frac{C}{2} \times (T^2 - T_{\text{ref}}^2) + \frac{D}{3} \times (T^3 - T_{\text{ref}}^3) + s_{\text{form, Vap}}$$

Units are calories per gram-mole kelvin and Kelvin.

Parameters

Symbol	Parameter Name	Units	Description
A	cp_mol_ig_comp_coeff_A	cal/mol·K	
B	cp_mol_ig_comp_coeff_B	cal/mol·K ²	
C	cp_mol_ig_comp_coeff_C	cal/mol·K ³	
D	cp_mol_ig_comp_coeff_D	cal/mol·K ⁴	
$s_{\text{form, Vap}}$	entr_mol_form_vap_comp_ref	Base units	Standard molar entropy of formation at reference state

Note: This correlation uses the same parameters as the ideal gas heat capacity correlation. Units of molar entropy of formation will be derived from the base units defined for the property package.

Saturation (Vapor) Pressure

Properties of Gases and Liquids 3rd edition uses the following correlation to calculate the vapor pressure of a component:

$$\ln(P_{\text{sat}}) = A - \frac{B}{T + C}$$

Units are mmHg and Kelvin.

Parameters

Symbol	Parameter Name	Units	Description
A	pressure_sat_comp_coeff_A	None	
B	pressure_sat_comp_coeff_B	K	
C	pressure_sat_comp_coeff_C	K	

Properties of Gases and Liquids 4th edition (RPP4)

Contents

- *Properties of Gases and Liquids 4th edition (RPP4)*
 - *Source*
 - *Ideal Gas Molar Heat Capacity (Constant Pressure)*
 - *Ideal Gas Molar Enthalpy*
 - *Ideal Gas Molar Entropy*
 - *Saturation (Vapor) Pressure*

Source

Methods for calculating pure component properties from:

The Properties of Gases & Liquids, 4th Edition Reid, Prausnitz and Polling, 1987, McGraw-Hill

All methods use SI units.

Ideal Gas Molar Heat Capacity (Constant Pressure)

Properties of Gases and Liquids uses the following correlation for the ideal gas molar heat capacity:

$$c_{p\text{ ig}} = A + B \times T + C \times T^2 + D \times T^3$$

Parameters

Symbol	Parameter Name	Units	Description
<i>A</i>	cp_mol_ig_comp_coeff_A	J/mol·K	
<i>B</i>	cp_mol_ig_comp_coeff_B	J/mol·K ²	
<i>C</i>	cp_mol_ig_comp_coeff_C	J/mol·K ³	
<i>D</i>	cp_mol_ig_comp_coeff_D	J/mol·K ⁴	

Ideal Gas Molar Enthalpy

The correlation for the ideal gas molar enthalpy is derived from the correlation for the molar heat capacity and is given below:

$$h_{\text{ig}} - h_{\text{ig ref}} = A \times (T - T_{\text{ref}}) + \frac{B}{2} \times (T^2 - T_{\text{ref}}^2) + \frac{C}{3} \times (T^3 - T_{\text{ref}}^3) + \frac{D}{4} \times (T^4 - T_{\text{ref}}^4) + \Delta h_{\text{form, Vap}}$$

Parameters

Symbol	Parameter Name	Units	Description
<i>A</i>	cp_mol_ig_comp_coeff_A	J/mol·K	
<i>B</i>	cp_mol_ig_comp_coeff_B	J/mol·K ²	
<i>C</i>	cp_mol_ig_comp_coeff_C	J/mol·K ³	
<i>D</i>	cp_mol_ig_comp_coeff_D	J/mol·K ⁴	
$\Delta h_{\text{form, Vap}}$	enth_mol_form_vap_comp_ref	Base units	Molar heat of formation at reference state

Note: This correlation uses the same parameters as the ideal gas heat capacity correlation. Units of molar heat of formation will be derived from the base units defined for the property package.

Ideal Gas Molar Entropy

The correlation for the ideal gas molar entropy is derived from the correlation for the molar heat capacity and is given below:

$$s_{ig} = A \times \ln(T/T_{ref}) + B \times (T - T_{ref}) + \frac{C}{2} \times (T^2 - T_{ref}^2) + \frac{D}{3} \times (T^3 - T_{ref}^3) + s_{form, Vap}$$

Parameters

Symbol	Parameter Name	Units	Description
A	cp_mol_ig_comp_coeff_A	J/mol·K	
B	cp_mol_ig_comp_coeff_B	J/mol·K ²	
C	cp_mol_ig_comp_coeff_C	J/mol·K ³	
D	cp_mol_ig_comp_coeff_D	J/mol·K ⁴	
$s_{form, Vap}$	entr_mol_form_vap_comp_ref	Base units	Standard molar entropy of formation at reference state

Note: This correlation uses the same parameters as the ideal gas heat capacity correlation. Units of molar entropy of formation will be derived from the base units defined for the property package.

Saturation (Vapor) Pressure

Properties of Gases and Liquids uses the following correlation to calculate the vapor pressure of a component:

$$\ln\left(\frac{P_{sat}}{P_{crit}}\right) \times (1 - x) = A \times x + B \times x^{1.5} + C \times x^3 + D \times x^6$$

where $x = 1 - \frac{T}{T_{crit}}$.

Symbol	Parameter Name	Units	Description
A	pressure_sat_comp_coeff_A	None	
B	pressure_sat_comp_coeff_B	None	
C	pressure_sat_comp_coeff_C	None	
D	pressure_sat_comp_coeff_D	None	
P_{crit}	pressure_crit_comp	Same as system pressure	Critical pressure
T_{crit}	temperature_crit_comp	Same as system temperature	Critical temperature

Note: This correlation is only valid at temperatures **below** the critical temperature. Above this point, there is no real solution to the equation.

Properties of Gases and Liquids 5th edition (RPP5)

Contents

- *Properties of Gases and Liquids 5th edition (RPP5)*
 - *Source*
 - *Ideal Gas Molar Heat Capacity (Constant Pressure)*
 - *Ideal Gas Molar Enthalpy*
 - *Ideal Gas Molar Entropy*
 - *Saturation (Vapor) Pressure*

Source

Methods for calculating pure component properties from:

The Properties of Gases & Liquids, 5th Edition Reid, Prausnitz and Polling, 2001, McGraw-Hill

All methods use SI units.

Ideal Gas Molar Heat Capacity (Constant Pressure)

Properties of Gases and Liquids uses the following correlation for the ideal gas molar heat capacity:

$$\frac{c_{p,ig}}{R} = a_0 + a_1 \times T + a_2 \times T^2 + a_3 \times T^3 + a_4 \times T^4$$

Parameters

Symbol	Parameter Name	Units	Description
a_0	cp_mol_ig_comp_coeff_a0	None	
a_1	cp_mol_ig_comp_coeff_a1	K ⁻¹	
a_2	cp_mol_ig_comp_coeff_a2	K ⁻²	
a_3	cp_mol_ig_comp_coeff_a3	K ⁻³	
a_4	cp_mol_ig_comp_coeff_a4	K ⁻⁴	
R	gas_constant	Same as heat capacity	Universal gas constant

Ideal Gas Molar Enthalpy

The correlation for the ideal gas molar enthalpy is derived from the correlation for the molar heat capacity and is given below:

$$\frac{h_{ig} - h_{ig,ref}}{R} = a_0 \times (T - T_{ref}) + \frac{a_1}{2} \times (T^2 - T_{ref}^2) + \frac{a_2}{3} \times (T^3 - T_{ref}^3) + \frac{a_3}{4} \times (T^4 - T_{ref}^4) + \frac{a_4}{5} \times (T^5 - T_{ref}^5) + \Delta h_{form, Vap}$$

Parameters

Symbol	Parameter Name	Units	Description
a_0	cp_mol_ig_comp_coeff_a0	None	
a_1	cp_mol_ig_comp_coeff_a1	K ⁻¹	
a_2	cp_mol_ig_comp_coeff_a2	K ⁻²	
a_3	cp_mol_ig_comp_coeff_a3	K ⁻³	
a_4	cp_mol_ig_comp_coeff_a4	K ⁻⁴	
$\Delta h_{\text{form, Vap}}$	enth_mol_form_vap_comp_ref	J/mol	Molar heat of formation at reference state

Note: This correlation uses the same parameters as the ideal gas heat capacity correlation.

Ideal Gas Molar Entropy

The correlation for the ideal gas molar entropy is derived from the correlation for the molar heat capacity and is given below:

$$\frac{s_{\text{ig}}}{R} = a_0 \times \ln(T/T_{\text{ref}}) + a_1 \times (T - T_{\text{ref}}) + \frac{a_2}{2} \times (T^2 - T_{\text{ref}}^2) + \frac{a_3}{3} \times (T^3 - T_{\text{ref}}^3) + \frac{a_4}{4} \times (T^4 - T_{\text{ref}}^4) + s_{\text{form, Vap}}$$

Parameters

Symbol	Parameter Name	Units	Description
a_0	cp_mol_ig_comp_coeff_a0	None	
a_1	cp_mol_ig_comp_coeff_a1	K ⁻¹	
a_2	cp_mol_ig_comp_coeff_a2	K ⁻²	
a_3	cp_mol_ig_comp_coeff_a3	K ⁻³	
a_4	cp_mol_ig_comp_coeff_a4	K ⁻⁴	
$s_{\text{form, Vap}}$	entr_mol_form_vap_comp_ref	J/mol·K	Standard molar entropy of formation at reference state

Note: This correlation uses the same parameters as the ideal gas heat capacity correlation.

Saturation (Vapor) Pressure

Properties of Gases and Liquids 5th edition uses the following correlation to calculate the vapor pressure of a component:

$$\text{Log}(P_{\text{sat}}) = A - \frac{B}{T + C}$$

Units are bar and Kelvin.

Parameters

Symbol	Parameter Name	Units	Description
A	pressure_sat_comp_coeff_A	None	
B	pressure_sat_comp_coeff_B	K	
C	pressure_sat_comp_coeff_C	K	

Phase Equilibrium Formulation

For those applications involving phase equilibria, there are number of different approaches that can be taken to specify the equilibrium condition. For example, equilibrium may be described in terms of an empirical partitioning coefficient or in terms of fugacities in each phase. To allow users to specify the approach they wish to use, each *Component* object contains a *phase_equilibrium_form* configuration argument.

As a given system may incorporate multiple phase equilibria, the *phase_equilibrium_form* argument should be a *dict* with keys being a tuple of interacting phases and values being a Python method describing how the equilibrium condition should be defined. A simple example for a VLE system is shown below:

```
"phase_equilibrium_form": {("Vap", "Liq"): fugacity}
```

The IDAES Generic Property Package Framework contains a library of common forms for the equilibrium condition, which is described [here](#).

Parameter Data

Most pure component property correlations depend upon empirical parameters which need to be specified by the user. All the in-built property libraries built these parameters automatically expect the user to provide values these parameters via the *parameter_data* configuration argument. The *parameter_data* configuration argument should be a *dict* with keys being the name of the required parameters and the values being a value or dict of values to use when initializing the parameter (i.e. the dict must have keys which match the indexing set of the parameter).

Users can specify the units of measurement for each parameter value, which will be automatically converted to match the set of units required by the property method. Users are encouraged to explicitly state the units of each parameter value for clarity, which is done using a tuple with the form (value, units), as shown in the example below. Users may choose to omit the units, providing only a value for the parameter (not as a tuple) in which case the units are assumed to match those defined for the associated parameter.

```
"parameter_data": {
    "property": (value, units),
    "indexed_property": {
        "index_1": (value, units),
        "index_2": (value, units)}}}
```

Note: A *dict* is used for specifying parameter values to allow users greater flexibility in defining their own methods with custom parameters.

Additionally, the following quantities are properties of the component (i.e. not a function of state) and are included in the component parameters.

- Molecular weight: “mw”
- Critical Pressure: “pressure_crit”
- Critical Temperature: “temperature_crit”

Defining Phases

The second step in defining a property package using the Generic Property Package Framework is to define the phases of interest in the system. Due to the equation-oriented nature of the IDAES modeling framework, it is necessary to define any phases the user believes may be important *a priori* as it is not possible to determine what phases should be included on-the-fly. Phases are defined using *IDAES Phase objects* `<user_guide/components/property_package/phase:Phase Object>`, and are automatically constructed using the *phases* configuration argument from the *GenericParameterBlock*.

The *phases* Argument

Each *GenericParameterBlock* has a configuration argument named *phases* which is used to construct the *Phase* objects and populate them with instructions on how to calculate thermophysical properties for that phase. The *phases* configuration argument is expected to be a *dict* where the keys are the names for the phases of interest and the values are a configuration arguments for the named phase (which are passed to the *Phase* object as it is instantiated).

```
"phases": {
  "phase_1": {
    "type": Phase,
    "equation_of_state": EoS,
    "equation_of_state_options": {},
    "parameter_data": {}},
  "phase_2": {
    "type": Phase,
    "equation_of_state": EoS,
    "equation_of_state_options": {},
    "parameter_data": {}}}
```

Type Argument

Each phase in the *phases* argument must be assigned a valid phase type from those supported by the IDAES Framework (e.g. LiquidPhase, SolidPhase, VaporPhase). This should be provided using the *type* argument.

Equations of State

Equations of state (or equivalent methods) describe the relationship between different thermophysical properties within a mixture and ensure that the behavior of these are thermodynamically consistent. Each phase must be assigned an Equation of State (or equivalent method) in the form of a Python module which will assemble the necessary variables, constraints and expressions associated with the desired approach.

A wide range of equations of states are available in literature for different applications and levels of rigor, and the IDAES Generic Property Package Framework provides a number of prebuilt modules for users, which are listed below.

Equation of state packages may allow for user options (e.g. choosing a specific type of cubic equation of state). The options are set using the *equation_of_state_options* argument, and the options available are described in the documentation of each equation of state module.

Equation of State Libraries

Ideal Gases and Liquids (Ideal)

Contents

- *Ideal Gases and Liquids (Ideal)*
 - *Introduction*
 - *Mass Density by Phase*
 - *Molar Density by Phase*
 - *Molar Enthalpy by Phase*
 - *Component Molar Enthalpy by Phase*
 - *Molar Entropy by Phase*
 - *Component Molar Entropy by Phase*
 - *Component Fugacity by Phase*
 - *Component Fugacity Coefficient by Phase*
 - *Molar Gibbs Energy by Phase*
 - *Component Gibbs Energy by Phase*

Introduction

Ideal behavior represents the simplest possible equation of state that ensures thermodynamic consistency between different properties.

Mass Density by Phase

The following equation is used for both liquid and vapor phases, where p indicates a given phase:

$$\rho_{mass,p} = \rho_{mol,p} \times MW_p$$

where MW_p is the mixture molecular weight of phase p .

Molar Density by Phase

For the vapor phase, the Ideal Gas Equation is used to calculate the molar density;

$$\rho_{mol,Vap} = \frac{P}{RT}$$

whilst for the liquid phase the molar density is the weighted sum of the pure component liquid densities:

$$\rho_{mol,Liq} = \sum_j x_{Liq,j} \times \rho_{Liq,j}$$

where $x_{Liq,j}$ is the mole fraction of component j in the liquid phase.

Molar Enthalpy by Phase

For both liquid and vapor phases, the molar enthalpy is calculated as the weighted sum of the component molar enthalpies for the given phase:

$$h_{mol,p} = \sum_j x_{p,j} \times h_{mol,p,j}$$

where $x_{p,j}$ is the mole fraction of component j in the phase p .

Component Molar Enthalpy by Phase

Component molar enthalpies by phase are calculated using the pure component method provided by the users in the property package configuration arguments.

Molar Entropy by Phase

For both liquid and vapor phases, the molar entropy is calculated as the weighted sum of the component molar entropies for the given phase:

$$s_{mol,p} = \sum_j x_{p,j} \times s_{mol,p,j}$$

where $x_{p,j}$ is the mole fraction of component j in the phase p .

Component Molar Entropy by Phase

Component molar entropies by phase are calculated using the pure component method provided by the users in the property package configuration arguments.

Component Fugacity by Phase

For the vapor phase, ideal behavior is assumed:

$$f_{vap,j} = P$$

For the liquid phase, Raoult's Law is used:

$$f_{liq,j} = P_{sat,j}$$

Component Fugacity Coefficient by Phase

Ideal behavior is assumed, so all $\phi_{p,j} = 1$ for all components and phases.

Molar Gibbs Energy by Phase

For both liquid and vapor phases, the molar Gibbs energy is calculated as the weighted sum of the component molar Gibbs energies for the given phase:

$$g_{mol,p} = \sum_j x_{p,j} \times g_{mol,p,j}$$

where $x_{p,j}$ is the mole fraction of component j in the phase p .

Component Gibbs Energy by Phase

Component molar Gibbs energies are calculated using the definition of Gibbs energy:

$$g_{mol,p,j} = h_{mol,p,j} - s_{mol,p,j} \times T$$

Cubic Equations of State (`Cubic`)

Contents

- *Cubic Equations of State (`Cubic`)*
 - *Introduction*
 - *General Cubic Equation of State*
 - *Property Package Options*
 - *Required Parameters*
 - *Calculation of Properties*
 - *Mass Density by Phase*
 - *Molar Density by Phase*
 - *Molar Enthalpy by Phase*
 - *Component Molar Enthalpy by Phase*
 - *Molar Entropy by Phase*
 - *Component Molar Entropy by Phase*
 - *Component Fugacity by Phase*
 - *Component Fugacity Coefficient by Phase*
 - *Molar Gibbs Energy by Phase*
 - *Component Gibbs Energy by Phase*

Introduction

This module implements a general form of a cubic equation of state which can be used for most cubic-type equations of state. The following forms are currently supported:

- Peng-Robinson
- Soave-Redlich-Kwong

General Cubic Equation of State

All equations come from “The Properties of Gases and Liquids, 4th Edition” by Reid, Prausnitz and Poling. The general cubic equation of state is represented by the following equations:

$$P = \frac{RT}{V - b} - \frac{a}{V^2 - ubV + wb^2}$$

An equivalent form of the previous equation is:

$$0 = Z^3 - (1 + B - uB)Z^2 + (A - uB - (u - w)B^2)Z - AB - wB^2 - wB^3$$

$$A = \frac{a_m P}{R^2 T^2}$$

$$B = \frac{b_m P}{RT}$$

where Z is the compressibility factor of the mixture, a_m and b_m are properties of the mixture and u and w are parameters which depend on the specific equation of state being used as show in the table below.

Equation	u	w	Ω_A	Ω_B	α_j
Peng-Robinson	2	-1	0.45724	0.07780	$(1 + (1 - T_r^2)(0.37464 + 1.54226\omega_j - 0.26992\omega_j^2))^2$
Soave-Redlich-Kwong	1	0	0.42748	0.08664	$(1 + (1 - T_r^2)(0.48 + 1.574\omega_j - 0.176\omega_j^2))^2$

The properties a_m and b_m are calculated from component specific properties a_j and b_j as shown below:

$$a_j = \frac{\Omega_A R^2 T_{c,j}^2}{P_{c,j}} \alpha_j$$

$$b_j = \frac{\Omega_B R T_{c,j}}{P_{c,j}}$$

$$a_m = \sum_i \sum_j y_i y_j (a_i a_j)^{1/2} (1 - \kappa_{ij})$$

$$b_m = \sum_i y_i b_i$$

where $P_{c,j}$ and $T_{c,j}$ are the component critical pressures and temperatures, y_j is the mole fraction of component j , κ_{ij} are a set of binary interaction parameters which are specific to the equation of state and Ω_A , Ω_B and α_j are taken from the table above. ω_j is the Pitzer acentric factor of each component.

The cubic equation of state is solved for each phase via a call to an external function which automatically identifies the correct root of the cubic and returns the value of Z as a function of A and B along with the first and second partial derivatives.

Property Package Options

When using the general cubic equation of state module, users must specify the type of cubic to use. This is done by providing a *type* option in the *equation_of_state_options* argument in the *Phase* definition, as shown in the example below.

```
from idaes.generic_models.properties.core.eos.ceos import Cubic, CubicType

configuration = {
    "phases": {
        "Liquid": {
            "type": LiquidPhase,
            "equation_of_state": Cubic,
            "equation_of_state_options": {
                "type": CubicType.PR}}}
```

Required Parameters

Cubic equations of state require the following parameters to be defined:

1. *omega* (Pitzer acentricity factor) needs to be defined for each component (in the *parameter_data* for each component).
2. *kappa* (binary interaction parameters) needs to be defined for each component pair in the system. This parameter needs to be defined in the general *parameter_data* argument for the overall property package (as it can be used in multiple phases).

Calculation of Properties

Many thermophysical properties are calculated using an ideal and residual term, such that:

$$p = p^0 + p^r$$

The residual term is derived from the partial derivatives of the cubic equation of state, whilst the ideal term is determined using pure component properties for the ideal gas phase defined for each component.

Mass Density by Phase

The following equation is used for both liquid and vapor phases, where *p* indicates a given phase:

$$\rho_{mass,p} = \rho_{mol,p} \times MW_p$$

where MW_p is the mixture molecular weight of phase *p*.

Molar Density by Phase

Molar density is calculated using the following equation

$$\rho_{mol, Vap} = \frac{P}{ZRT}$$

Molar Enthalpy by Phase

The residual enthalpy term is given by:

$$h_i^r b_m \sqrt{u^2 - 4w} = \left(T \frac{da}{dT} - a_m \right) \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right) + RT(Z - 1)b_m \sqrt{u^2 - 4w}$$

$$\frac{da}{dT} \sqrt{T} = -\frac{R}{2} \sqrt{\Omega_A} \sum_i \sum_j y_i y_j (1 - k_{ij}) \left(f_{w,j} \sqrt{a_i \frac{T_{c,j}}{P_{c,j}}} + f_{w,i} \sqrt{a_j \frac{T_{c,i}}{P_{c,i}}} \right)$$

The ideal component is calculated from the weighted sum of the (ideal) component molar enthalpies.

Component Molar Enthalpy by Phase

Component molar enthalpies by phase are calculated using the pure component method provided by the users in the property package configuration arguments.

Molar Entropy by Phase

The residual entropy term is given by:

$$s_i^r b_m \sqrt{u^2 - 4w} = R \ln \frac{Z - B}{Z} b_m \sqrt{u^2 - 4w} + R \ln \frac{Z P^{ref}}{P} b_m \sqrt{u^2 - 4w} + \frac{da}{dT} \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right)$$

The ideal component is calculated from the weighted sum of the (ideal) components molar enthalpies.

Component Molar Entropy by Phase

Component molar entropies by phase are calculated using the pure component methods provided by the users in the property package configuration arguments.

Component Fugacity by Phase

Fugacity is calculated from the system pressure and fugacity coefficients as follows:

$$f_{i,p} = \phi_{i,p} P$$

Component Fugacity Coefficient by Phase

The fugacity coefficient is calculated from the departure function of the cubic equation of state as shown below:

$$\ln \phi_i = \frac{b_i}{b_m} (Z - 1) - \ln (Z - B) + \frac{A}{B\sqrt{u^2 - 4w}} \left(\frac{b_i}{b_m} - \delta_i \right) \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right)$$
$$\delta_i = \frac{2a_i^{1/2}}{a_m} \sum_j x_j a_j^{1/2} (1 - k_{ij})$$

Molar Gibbs Energy by Phase

For both liquid and vapor phases, the molar Gibbs energy is calculated as the weighted sum of the component molar Gibbs energies for the given phase:

$$g_{mol,p} = \sum_j x_{p,j} \times g_{mol,p,j}$$

where $x_{p,j}$ is the mole fraction of component j in the phase p .

Component Gibbs Energy by Phase

Component molar Gibbs energies are calculated using the definition of Gibbs energy:

$$g_{mol,p,j} = h_{mol,p,j} - s_{mol,p,j} \times T$$

Phase-Specific Parameter

In some cases, a property package may include parameters which are specific to a given phase. In these cases, these parameters are stored as part of the associated *Phase* object and the values of these set using the *parameter_data* argument when declaring the phase. This is done in the same fashion as for *component specific parameters*.

Phases with Partial Component Lists

In many applications a mixture will contain species that only appear in a single phase (either by nature or assumption). Common examples include crystalline solids and non-condensable gases. The IDAES Generic Property Package Framework provides support for these behaviors and allows users to specify phase-specific component lists (i.e. a list of components which appear in a given phase).

This is done by providing a phase with a *component_list* argument, which provides a *list* of component names which appear in the phase. The framework automatically validates the *component_list* argument to ensure that it is a subset of the master component list for the property package, and will inform the user if an unrecognized component is included. If a phase is not provided with a *component_list* argument it is assumed that all components defined in the master component list may be present in the phase.

State Definition

Defining State Variables

An important part of defining a set of property calculations is choosing the set of variables which will describe the state of the material. The set of state variables needs to include information on the extensive flow, composition and thermodynamic state of the material. However, there are many ways in which this information can be described, and the best choice of state variables depends on many factors.

Within the IDAES Generic Property Package Framework, the definition of state variables is done using sub-modules which create the necessary variables supporting information for the property package. A state definition sub-module may define any set of state variables the user feel appropriate, but must define the following components as either state variables or functions of the state variables:

- *temperature* (must be a Pyomo Var)
- *pressure*
- *mole_frac_phase_comp*
- *phase_frac*

The IDAES Generic Property Package Framework has a library of prebuilt state definition sub-modules for users to use which are listed below.

State Definition Libraries

FTP_x

Contents

- *FTP_x*
 - *State Definition*
 - *Application*
 - *Bounds*
 - *Supporting Variables and Constraints*
 - *Default Balance Types and Flow Basis*

State Definition

This approach describes the material state in terms of total flow (F : *flow_mol*), overall (mixture) mole fractions (x_j : *mole_frac_comp*), temperature (T : *temperature*) and pressure (P : *pressure*). As such, there are $3 + N_{components}$ state variables, however only $2 + N_{components}$ are independent as the mole fraction must sum to 1.

Application

This is the simplest approach to fully defining the state of a material, and one of the most easily accessible to the user as it is defined in terms of variables that are easily measured and understood. However, this approach has a number of limitations which the user should be aware of:

- If the property package is set up for multiphase flow, an equilibrium calculation is required at the inlet of each unit, as the state definition does not contain information on multiphase flow. This increases the number of complex equilibrium calculations that must be performed, which could be avoided by using a different state definition.
- State becomes ill-defined when only one component is present and multiphase behavior can occur, as temperature and pressure are insufficient to fully define the thermodynamic state under these conditions.

Bounds

The FTPx module supports bounding of the following variables through the *state_bounds* configuration argument:

- *flow_mol*
- *temperature*
- *pressure*

Note that mole fractions are automatically assigned a lower bound of 0, but the upper bound is left free as this is implicitly defined by the sum of mole fractions constraint.

Supporting Variables and Constraints

In addition to the state variables, this definition of state creates a number of supporting variables and constraints.

Variables

- *flow_mol_phase* ($F_{mol,p}$)
- *mole_frac_phase_comp* ($x_{p,j}$)
- *phase_frac* (ψ_p)

Constraints

In all cases, a constraint is written for the sum of the overall mole fractions.

$$\sum_j x_j = 1$$

Note: The sum of mole fractions constraint is not written at inlet states, as all mole fractions should be defined in the inlet stream.

If the property package supports only one phase:

$$F_{mol,p} = F_{mol}$$

$$x_{p,j} = x_j \text{ for all } j$$

$$\psi_p = 1$$

If the property package supports only two phases, the Rachford-Rice formulation is used:

$$\sum_p F_{mol,p} = F_{mol}$$

$$F_{mol} \times x_j = \sum_p F_{mol,p} \times x_{p,j} \text{ for all } j$$

$$\sum_j x_{\text{phase } 1,j} - \sum_j x_{\text{phase } 2,j} = 0$$

$$\psi_p \times F_{mol} = F_{mol,p} \text{ for all } p$$

If the property package supports more than two phases, the following general formulation is used:

$$F_{mol} \times x_j = \sum_p F_{mol,p} \times x_{p,j} \text{ for all } j$$

$$\sum_j x_{p,j} = 1 \text{ for all } p$$

$$\psi_p \times F_{mol} = F_{mol,p} \text{ for all } p$$

Default Balance Types and Flow Basis

The following defaults are specified for Unit Models using this state definition:

- Material balances: total component balances
- Material flow basis: molar flow
- Energy balances: total enthalpy

FCTP

Contents

- *FCTP*
 - *State Definition*
 - *Application*
 - *Bounds*
 - *Supporting Variables and Constraints*
 - *Default Balance Types and Flow Basis*

State Definition

This approach describes the material state in terms of total component flow (F : *flow_mol_comp*), temperature (T : *temperature*) and pressure (P : *pressure*). As such, there are $2 + N_{components}$ state variables.

Application

This approach is similar to using total flow rate and mole fractions as state variable, and is generally accessible to the user as the state variables are easily measured and understood. Compared to using total flow rate and mole fractions, this approach changes how the bilinear terms (flow rate times mole fraction) appear in the problem structure, and may result in improved performance for some applications. However, this approach has a number of limitations which the user should be aware of:

- If the property package is set up for multiphase flow, an equilibrium calculation is required at the inlet of each unit, as the state definition does not contain information on multiphase flow. This increases the number of complex equilibrium calculations that must be performed, which could be avoided by using a different state definition.
- State becomes ill-defined when only one component is present and multiphase behavior can occur, as temperature and pressure are insufficient to fully define the thermodynamic state under these conditions.

Bounds

The FcTP module supports bounding of the following variables through the *state_bounds* configuration argument:

- *flow_mol_comp*
- *temperature*
- *pressure*

Supporting Variables and Constraints

In addition to the state variables, this definition of state creates a number of supporting variables and constraints.

Variables

- *flow_mol_phase* ($F_{mol,p}$)
- *mole_frac_comp* (x_j)
- *mole_frac_phase_comp* ($x_{p,j}$)
- *phase_frac* (ψ_p)

Expressions

An Expression is created for the total flowrate such that $F = \sum F_j$

Constraints

In all cases, a constraint is created to calculate component mole fractions from the component flow rates.

$$F_j = x_j \times \sum F_j$$

Note: If only one component is present in the property package, this is simplified to $x_j = 1$.

If the property package supports only one phase:

$$F_{mol,p} = F_{mol}$$

$$x_{p,j} = x_j \text{ for all } j$$

$$\psi_p = 1$$

If the property package supports only two phases, the Rachford-Rice formulation is used:

$$\sum_p F_{mol,p} = F_{mol}$$

$$F_{mol,j} = \sum_p (F_{mol,p} \times x_{p,j}) \text{ for all } j$$

$$\sum_j x_{\text{phase } 1,j} - \sum_j x_{\text{phase } 2,j} = 0$$

$$\psi_p \times F_{mol} = F_{mol,p} \text{ for all } p$$

If the property package supports more than two phases, the following general formulation is used:

$$F_{mol,j} = \sum_p (F_{mol,p} \times x_{p,j}) \text{ for all } j$$

$$\sum_j x_{p,j} = 1 \text{ for all } p$$

$$\psi_p \times F_{mol} = F_{mol,p} \text{ for all } p$$

Default Balance Types and Flow Basis

The following defaults are specified for Unit Models using this state definition:

- Material balances: total component balances
- Material flow basis: molar flow
- Energy balances: total enthalpy

FPhx**Contents**

- *FPhx*
 - *State Definition*
 - *Application*
 - *Bounds*
 - *Supporting Variables and Constraints*
 - *Default Balance Types and Flow Basis*

State Definition

This approach describes the material state in terms of total flow (F : *flow_mol*), overall (mixture) mole fractions (x_j : *mole_frac_comp*), total molar enthalpy (h : *enth_mol*) and pressure (P : *pressure*). As such, there are $3 + N_{components}$ state variables, however only $2 + N_{components}$ are independent as the mole fraction must sum to 1.

Application

This approach is commonly used by other process simulation tools as it avoids the issues associated with using temperature and pressure as state variables in single component systems. However, as the user generally does not know the specific enthalpy of their feed streams, this approach requires some method to calculate this for feed streams. This can generally be done by specifying temperature of the feed, and then solving for the specific enthalpy.

This approach suffers from the following limitation which the user should be aware of:

- If the property package is set up for multiphase flow, an equilibrium calculation is required at the inlet of each unit, as the state definition does not contain information on multiphase flow. This increases the number of complex equilibrium calculations that must be performed, which could be avoided by using a different state definition.

Bounds

The FPhx module supports bounding of the following variables through the *state_bounds* configuration argument:

- *flow_mol*
- *enth_mol*
- *pressure*
- *temperature*

Supplying bounds for temperature is supported as these are often known to greater accuracy than the enthalpy bounds, and specifying these can help the solver find a feasible solution.

Note that mole fractions are automatically assigned a lower bound of 0, but the upper bound is left free as this is implicitly defined by the sum of mole fractions constraint.

Supporting Variables and Constraints

In addition to the state variables, this definition of state creates a number of supporting variables and constraints.

Variables

- *flow_mol_phase* ($F_{mol,p}$)
- *mole_frac_phase_comp* ($x_{p,j}$)
- *temperature* (T)
- *phase_frac* (ψ_p)

Constraints

In all cases, a constraint is written for the sum of the overall mole fractions.

$$\sum_j x_j = 1$$

Note: The sum of mole fractions constraint is not written at inlet states, as all mole fractions should be defined in the inlet stream.

Additionally, a constraint relating the total specific enthalpy to the specific enthalpy of each phase is written.

$$h_{mol} = \sum_j \psi_p \times h_{mol,p}$$

If the property package supports only one phase:

$$F_{mol,p} = F_{mol}$$

$$x_{p,j} = x_j \text{ for all } j$$

$$\psi_p = 1$$

If the property package supports only two phases, the Rachford-Rice formulation is used:

$$\sum_p F_{mol,p} = F_{mol}$$

$$F_{mol} \times x_j = \sum_p F_{mol,p} \times x_{p,j} \text{ for all } j$$

$$\sum_j x_{\text{phase } 1,j} - \sum_j x_{\text{phase } 2,j} = 0$$

$$\psi_p \times F_{mol} = F_{mol,p} \text{ for all } p$$

If the property package supports more than two phases, the following general formulation is used:

$$F_{mol} \times x_j = \sum_p F_{mol,p} \times x_{p,j} \text{ for all } j$$

$$\sum_j x_{p,j} = 1 \text{ for all } p$$

$$\psi_p \times F_{mol} = F_{mol,p} \text{ for all } p$$

Default Balance Types and Flow Basis

The following defaults are specified for Unit Models using this state definition:

- Material balances: total component balances
- Material flow basis: molar flow
- Energy balances: total enthalpy

FcPh

Contents

- *FcPh*
 - *State Definition*
 - *Application*
 - *Bounds*
 - *Supporting Variables and Constraints*
 - *Default Balance Types and Flow Basis*

State Definition

This approach describes the material state in terms of total component flow (F : *flow_mol_comp*), total specific enthalpy (h : *enth_mol*) and pressure (P : *pressure*). As such, there are $2 + N_{components}$ state variables.

Application

This approach is similar to the FPhx formulation used by many process simulators, with the exception that component flow rates are used in place of total flow and mole fractions. This changes where the bilinear terms (flow rate time mole fractions) appear in the problem structure and may improve robustness in some cases. The use of pressure and enthalpy as state variables avoids the issues related to using temperature and pressure as state variables for single component systems. However, as the user generally does not know the specific enthalpy of their feed streams, this approach requires some method to calculate this for feed streams. This can generally be done by specifying temperature of the feed, and then solving for the specific enthalpy.

This approach suffers from the following limitation which the user should be aware of:

- If the property package is set up for multiphase flow, an equilibrium calculation is required at the inlet of each unit, as the state definition does not contain information on multiphase flow. This increases the number of complex equilibrium calculations that must be performed, which could be avoided by using a different state definition.

Bounds

The FcPh module supports bounding of the following variables through the *state_bounds* configuration argument:

- *flow_mol_comp*
- *enth_mol*
- *pressure*
- *temperature*

Supplying bounds for temperature is supported as these are often known to greater accuracy than the enthalpy bounds, and specifying these can help the solver find a feasible solution.

Supporting Variables and Constraints

In addition to the state variables, this definition of state creates a number of supporting variables and constraints.

Variables

- *flow_mol_phase* ($F_{mol,p}$)
- *mole_frac_comp* (x_j)
- *mole_frac_phase_comp* ($x_{p,j}$)
- *temperature* (T)
- *phase_frac* (ψ_p)

Expressions

An Expression is created for the total flowrate such that $F = \sum F_j$

Constraints

In all cases, a constraint is created to calculate component mole fractions from the component flow rates.

$$F_j = x_j \times \sum F_j$$

Note: If only one component is present in the property package, this is simplified to $x_j = 1$.

If the property package supports only one phase:

$$F_{mol,p} = F_{mol}$$

$$x_{p,j} = x_j \text{ for all } j$$

$$\psi_p = 1$$

If the property package supports only two phases, the Rachford-Rice formulation is used:

$$\begin{aligned}\sum_p F_{mol,p} &= F_{mol} \\ F_{mol,j} &= \sum_p (F_{mol,p} \times x_{p,j}) \text{ for all } j \\ \sum_j x_{\text{phase } 1,j} - \sum_j x_{\text{phase } 2,j} &= 0 \\ \psi_p \times F_{mol} &= F_{mol,p} \text{ for all } p\end{aligned}$$

If the property package supports more than two phases, the following general formulation is used:

$$\begin{aligned}F_{mol,j} &= \sum_p (F_{mol,p} \times x_{p,j}) \text{ for all } j \\ \sum_j x_{p,j} &= 1 \text{ for all } p \\ \psi_p \times F_{mol} &= F_{mol,p} \text{ for all } p\end{aligned}$$

Default Balance Types and Flow Basis

The following defaults are specified for Unit Models using this state definition:

- Material balances: total component balances
- Material flow basis: molar flow
- Energy balances: total enthalpy

FpcTP

Contents

- *FpcTP*
 - *State Definition*
 - *Application*
 - *Bounds*
 - *Supporting Variables and Constraints*
 - *Default Balance Types and Flow Basis*

State Definition

This approach describes the material state in terms of phase-component flow ($F_{p,j}$: *flow_mol_phase_comp*), temperature (T : *temperature*) and pressure (P : *pressure*). As such, there are $2 + \text{phases} * N_{\text{components}}$ state variables.

Application

This approach required knowledge of the phase-equilibrium of the material in order to define the state variables. Compared to using total flow rate and mole fractions, this approach contains full information on the phase equilibria within the state variables, and thus avoids the needs for flash calculations in many cases. This can greatly reduce the complexity of the problem, and results can significantly affect the tractability of the problem. However, this approach has a number of limitations which the user should be aware of:

- Users must have knowledge of, or calculate, the phase-component flows of all inlet streams. For single phase flows this is often known, but for streams with potential two-phase behaviour this can require a set of flash calculations for the feed stream (users can make use of Feed blocks to assist with this).
- State becomes ill-defined when only one component is present and multiphase behavior can occur, as temperature and pressure are insufficient to fully define the thermodynamic state under these conditions.

Bounds

The FpcTP module supports bounding of the following variables through the *state_bounds* configuration argument:

- *flow_mol_phase_comp*
- *temperature*
- *pressure*

Supporting Variables and Constraints

In addition to the state variables, this definition of state creates a number of supporting variables and constraints.

Variables

- *mole_frac_phase_comp* ($x_{p,j}$)

Expressions

- *flow_mol* ($F = \sum F_{p,j}$)
- *flow_mol_phase* ($F_p = \sum F_{p,j}$)
- *flow_mol_comp* ($F_j = \sum F_{p,j}$)
- *mole_frac_comp* ($x_j = \frac{\sum F_{p,j}}{F}$)
- *phase_frac* ($\psi_p = \frac{F_p}{F}$ or $\psi_p = 1$ if only single phase)

Constraints

A set of constraints is created to calculate phase-component mole fractions from the phase-component flow rates.

$$F_j = x_{p,j} \times \sum F_p = F_{p,j}$$

Default Balance Types and Flow Basis

The following defaults are specified for Unit Models using this state definition:

- Material balances: total component balances
- Material flow basis: molar flow
- Energy balances: total enthalpy

Setting Bounds on State Variables

For optimization applications, it is important to specify a good initial guess and bounds on the state variables in order to improve the robustness of the problem. Further, due to the empirical nature of most thermophysical correlations these correlations are only valid in specific range of states. Users should set the *state_bounds* configuration argument to define the bounds on the state variables of their property package.

The *state_bounds* configuration argument should be a *dict* where the keys are the names of the state variables (using the standard naming convention) and the values should be a tuple with the form (*lower*, *nominal*, *upper*, *units*). The *lower* and *upper* values are used to set the lower and upper bounds respectively, whilst the *nominal* value is used to set the initial value for the state variable. The *units* value is optional, and is used to specify the units of measurement for the values provided, which will be used to automatically convert these values to the base set of units defined for the property package if required. If the *units* value is omitted, it is assumed that the values provided are in the base unit set for the property package.

Note: Some state definitions allow for setting on additional variables beyond the chosen state variables (temperature is a common example). See the documentation for your state definition for more information on what bounds can be set using the *state_bounds* argument.

Reference State

Many thermophysical properties are relative quantities, and require the definition of a thermodynamic reference state. Whilst some simpler models and correlations forego this or define the reference state implicitly, the IDAES Generic Property Package requires the user to specify the thermodynamic reference state (even if it is not used explicitly).

As such, users must provide the following two configuration arguments:

- *pressure_ref* - pressure at reference state
- *temperature_ref* - temperature at reference state

Defining Phase Equilibria

Phase equilibrium and separation is a key part of almost all chemical processes, and also represent some of the most complex and non-linear constraints in a model, especially when dealing with systems which may cross phase boundaries. Systems may also include multiple interacting phases with equilibrium, which further complicates the problem. As such, good formulations of these constraints is key to a robust and tractable model.

The IDAES Generic Property Package framework supports a range of phase equilibrium behaviors, including multiple phases in equilibrium and different formulations for describing the equilibria. These are all optional, and users do not need to define phase equilibria if it is not required for their system.

Setting up phase equilibrium within the framework is done using three configuration arguments as discussed below. However, users should be aware that some of these options require definition of further properties, such as bubble and dew point calculations.

Define Phases in Equilibrium

The first step in setting up phase equilibrium in the framework is to describe which phases are in equilibrium with each other. In general, for phases to be in equilibrium with each other, the following conditions need to be met:

1. Phases must be in direct contact with each other, and
2. At least one component must appear in both phases (equilibria involving chemical reactions are handled by *ReactionBlocks*).

In order to describe which phases are in equilibrium, the user needs to set the *phase_in_equilibrium* construction argument, which should be a list of 2-tuples where each tuple describes a pair of phases which are in equilibrium. Any component which appears in both phases in a pair is assumed to be in equilibrium.

A simple example for a VLE system is shown below.

```
"phases_in_equilibrium" = [ ("Vap", "Liq") ]
```

Note: Users should take care not to over define their system. For example, in a VLSE system a user could potentially write three sets of equilibrium constraints (VL, LS and VS). However, this would result in an over defined system, as only two of these three are independent. For most situations, a user would consider only the VL and LS equilibria, with the VS being implicitly defined.

Define Equilibrium State Formulation

Next, for each pair of phases in equilibrium, the user must define a formulation for the equilibrium state. To handle the complexities of disappearing phases, the IDAES Generic Property Package Framework allows for phase equilibrium to be solved at a separate equilibrium state rather than the actual state of the material. This allows for formulations which avoid disappearing phases by limiting the equilibrium state to exist within the valid two-phase region, whilst returning a negligible amount of any phase which is not valid at the actual material state.

The equilibrium state formulation is set using the *phase_equilibrium_state* configuration argument. This should be a *dict* where the keys are 2-tuples of phases in equilibrium (matching those defined in the *phases_in_equilibrium* argument) and values are a phase equilibrium state formulation method. The IDAES Generic Property Package Framework contains a library of methods for the formulation of the phase equilibrium state, which is shown below.

Phase Equilibrium State Libraries

Smooth Vapor-Liquid Equilibrium Formulation (*smooth_VLE*)

Contents

- *Smooth Vapor-Liquid Equilibrium Formulation (smooth_VLE)*
 - *Source*
 - *Introduction*
 - *Formulation*

Source

Burgard, A.P., Eason, J.P., Eslick, J.C., Ghouse, J.H., Lee, A., Biegler, L.T., Miller, D.C., 2018, A Smooth, Square Flash Formulation for Equation-Oriented Flowsheet Optimization. Proceedings of the 13th International Symposium on Process Systems Engineering – PSE 2018, July 1-5, 2018, San Diego.

Introduction

Typically, equilibrium calculations are only used when the user knows the current state is within the two-phase envelope. For simulation only studies, the user may know a priori the condition of the stream but when the same set of equations are used for optimization, there is a high probability that the specifications can transcend the phase envelope. In these situations, the equilibrium calculations become trivial, thus it is necessary to find a formulation that has non-trivial solutions at all states.

To address this, the smooth vapor-liquid equilibrium (VLE) formulation always solves the equilibrium calculations at a condition where a valid two-phase solution exists. In situations where only a single phase is present, the phase equilibrium is solved at the either the bubble or dew point, where the non-existent phase exists but in negligible amounts. In this way, a non-trivial solution is guaranteed but still gives near-zero material in the non-existent phase in the single phase regions.

Formulation

The approach used by the smooth VLE formulation is to define an “equilibrium temperature” (T_{eq}) at which the equilibrium calculations will be performed. The equilibrium temperature is computed as follows:

$$T_1 = \max(T_{bubble}, T)$$

$$T_{eq} = \min(T_1, T_{dew})$$

where T is the actual stream temperature, T_1 is an intermediate temperature variable and T_{bubble} and T_{dew} are the bubble and dew point temperature of mixture. In order to express the maximum and minimum operators in a tractable form, these equations are reformulated using the IDAES *smooth_max* and *smooth_min* operators which results in the following equations:

$$T_1 = 0.5 \left[T + T_{bubble} + \sqrt{(T - T_{bubble})^2 + \epsilon_1^2} \right]$$

$$T_{eq} = 0.5 \left[T_1 + T_{dew} - \sqrt{(T - T_{dew})^2 + \epsilon_2^2} \right]$$

where ϵ_1 and ϵ_2 are smoothing parameters (mutable *Params* named *eps_1* and *eps_2*). The default values are 0.01 and 0.0005 respectively, and it is recommended that $\epsilon_1 > \epsilon_2$. It can be seen that if the stream temperature is less than that of the bubble point temperature, the VLE calculations will be computed at the bubble point. Similarly, if the stream temperature is greater than the dew point temperature, then the VLE calculations are computed at the dew point temperature. For all other conditions, the equilibrium calculations will be computed at the actual temperature.

Finally, the phase equilibrium is expressed using the following equation:

$$\Phi_{\text{vap},j}(T_{eq}) = \Phi_{\text{liq},j}(T_{eq})$$

where $\Phi_{p,j}(T_{eq})$ is the fugacity of component j in the phase p calculated at T_{eq} . The fugacities are calculated using methods defined by the equation of state chosen by the user for each phase.

Necessary Properties

Next, any component which is involved in a phase equilibrium interaction (i.e, appears in both phases of an interacting pair) must define a form for the required equilibrium constraint. There are a number of ways these constraints can be written depending on the equation of state and scaling of the problem. This is set using the *phase_equilibrium_form* configuration argument in the *Component* objects, and takes the form of a *dict* where the keys are 2-tuples of interacting phases and the value is the formulation to use for the current component across the given phase pair. For example:

```
parameters.component_1.config.phase_equilibrium_form = {(phase_1, phase_2): ↪ formulation}
```

A library of common forms for equilibrium constraints is available, and is shown below.

Library of Common Equilibrium Forms

The IDAES Generic Property Package Framework contains a library of common forms for phase equilibrium conditions.

Contents

- *Library of Common Equilibrium Forms*
 - *Fugacity (fugacity)*

Fugacity (fugacity)

Fugacity of each component must be equal between interacting phases

$$x_{p1,j} \times f_{p1,j} = x_{p2,j} \times f_{p2,j}$$

Bubble and Dew Point Calculations

Bubble and dew points are often of interest to process engineers for designing process equipment, and appear in some calculations of other thermodynamic properties. They are also useful in getting initial guesses for states in phase equilibrium problems, and some equilibrium state formulations rely on these properties.

Whilst calculation of the saturation pressure for single components is relatively simple, calculating the bubble and dew points of mixtures is more challenging due to the non-linear nature of the equations. Calculation of these properties is generally done through calculations based on the equations of state for the liquid and vapor phases, however these calculations can be greatly simplified if ideal behavior is assumed for both phases (i.e. ideal gas and Raoult's law). To allow for both cases, the IDAES Generic Property Package Framework provides a library of different formulations for the bubble and dew point calculations, which can be set using the following arguments:

- *bubble_dew_method*

A list of available methods is given below:

Bubble and Dew Point Methods

Contents

- *Bubble and Dew Point Methods*
 - *Ideal Assumptions (IdealBubbleDew)*
 - * *Ideal Bubble Pressure*
 - * *Ideal Bubble Temperature*
 - * *Ideal Dew Pressure*
 - * *Ideal Dew Temperature*
 - *Equal Fugacity (log form) (LogBubbleDew)*
 - * *Bubble Pressure (log form)*
 - * *Bubble Temperature (log form)*
 - * *Dew Pressure (log form)*
 - * *Dew Temperature (log form)*

Ideal Assumptions (IdealBubbleDew)

In the case where ideal behavior can be assumed, i.e. ideal gas assumption and Raoult's Law holds, the bubble and dew points can be calculated directly from the saturation pressure using the following equations.

Ideal Bubble Pressure

$$P_{bub} = \sum_j x_j \times P_{sat,j}(T)$$

$$x_j(P_{bub}) \times P_{bub} = x_j \times P_{sat,j}(T)$$

where P_{bub} is the bubble pressure of the mixture, $P_{sat,j}(T)$ is the saturation pressure of component j at the system temperature, T , x_j is the overall mixture mole fraction and $x_j(P_{bub})$ is the mole fraction of the vapor phase at the bubble pressure.

Ideal Bubble Temperature

$$\sum_j (x_j \times P_{sat,j}(T_{bub})) - P = 0$$

$$x_j(T_{bub}) \times P = x_j \times P_{sat,j}(T_{bub})$$

where P is the system pressure, $P_{sat,j}(T_{bub})$ is the saturation pressure of component j at the bubble temperature, T_{bub} , x_j is the overall mixture mole fraction and $x_j(T_{bub})$ is the mole fraction of the vapor phase at the bubble temperature.

Ideal Dew Pressure

$$0 = 1 - P_{dew} \times \sum_j x_j \times P_{sat,j}(T)$$

$$x_j(P_{dew}) \times P_{sat,j}(T) = x_j \times P_{dew}$$

where P_{dew} is the dew pressure of the mixture, $P_{sat,j}(T)$ is the saturation pressure of component j at the system temperature, T , x_j is the overall mixture mole fraction and $x_j(P_{dew})$ is the mole fraction of the liquid phase at the dew pressure.

Ideal Dew Temperature

$$P \times \sum_j (x_j \times P_{sat,j}(T_{dew})) - 1 = 0$$

$$x_j(T_{dew}) \times P_{sat,j}(T_{dew}) = x_j \times P$$

where P is the system pressure, $P_{sat,j}(T_{dew})$ is the saturation pressure of component j at the dew temperature, T_{dew} , x_j is the overall mixture mole fraction and $x_j(T_{dew})$ is the mole fraction of the liquid phase at the dew temperature.

Equal Fugacity (log form) (LogBubbleDew)

For cases where ideal behavior is insufficient, it is necessary to calculate the fugacity of each component at the relevant transition point and enforce equality of the fugacity in each phase. As such, this methods depends upon the definition of fugacity for each phase and component. In this formulation, the logarithm of the phase equilibrium constraint is used.

Bubble Pressure (log form)

$$\ln(x_j) + \ln(f_{liquid,j}(P_{bub})) = \ln(x_j(P_{bub})) + \ln(f_{vapor,j}(P_{bub}))$$
$$1 = \sum_j x_j(P_{bub})$$

where P_{bub} is the bubble pressure of the mixture, $f_{p,j}(P_{bub})$ is the fugacity of component j in phase p at P_{bub} , x_j is the overall mixture mole fraction and $x_j(P_{bub})$ is the mole fraction of the vapor phase at the bubble pressure.

Bubble Temperature (log form)

$$\ln(x_j) + \ln(f_{liquid,j}(T_{bub})) = \ln(x_j(T_{bub})) + \ln(f_{vapor,j}(T_{bub}))$$
$$1 = \sum_j x_j(T_{bub})$$

where T_{bub} is the bubble temperature of the mixture, $f_{p,j}(T_{bub})$ is the fugacity of component j in phase p at T_{bub} , x_j is the overall mixture mole fraction and $x_j(T_{bub})$ is the mole fraction of the vapor phase at the bubble temperature.

Dew Pressure (log form)

$$\ln(x_j(P_{dew})) + \ln(f_{liquid,j}(P_{dew})) = \ln(x_j) + \ln(f_{vapor,j}(P_{dew}))$$
$$1 = \sum_j x_j(P_{dew})$$

where P_{dew} is the dew pressure of the mixture, $f_{p,j}(P_{dew})$ is the fugacity of component j in phase p at P_{dew} , x_j is the overall mixture mole fraction and $x_j(P_{dew})$ is the mole fraction of the vapor phase at the dew pressure.

Dew Temperature (log form)

$$\ln(x_j(T_{dew})) + \ln(f_{liquid,j}(T_{dew})) = \ln(x_j) + \ln(f_{vapor,j}(T_{dew}))$$
$$1 = \sum_j x_j(T_{dew})$$

where T_{dew} is the dew temperature of the mixture, $f_{p,j}(T_{dew})$ is the fugacity of component j in phase p at T_{dew} , x_j is the overall mixture mole fraction and $x_j(T_{dew})$ is the mole fraction of the vapor phase at the dew temperature.

Global Options

The IDAES Generic Properties Framework also includes a number of general options for further customizing how certain properties are calculated. These options are listed below. Note that these are generally considered to be global options and will affect all phases and components in the system. Global configuration arguments can be declared at the top level of the property package configuration dict.

Contents

- *Global Options*
 - *Enthalpy of Formation*

Enthalpy of Formation

In most process applications, it is necessary to include calculation of the enthalpy of formation for each component in the system in order to account for latent heat due to chemical reactions or phase changes. However, in certain circumstances, such as systems with no reactions or phase equilibria or cases where the user wishes to specify latent heats as separate parameters, it may be desirable to exclude the heat of formation from the specific enthalpy calculations.

Users can define whether or not the heat of formation should be included in the calculation of the specific enthalpy using the *include_enthalpy_of_formation* configuration argument. If this is set to *True* (the default behavior) then heat of formation will be included in the calculation of the specific enthalpy, if set to *False* then heat of formation will be excluded.

Developing New Property Libraries

Information on how to develop new components for the IDAES Generic Property Package Framework are given in the following sections.

Contents

Developing Pure Component Methods

Contents

- *Developing Pure Component Methods*
 - *Naming Methods*
 - *Method Arguments*
 - *Method Parameters*
 - *Method Body*
 - *Example*

The most common task developers of new property packages will need to do is writing methods for new pure component property calculations. Most equation of state type approaches rely on a set of calculations for pure components

under ideal conditions which are then modified to account for mixing and deviations from ideality. These pure component property calculations tend to be empirical correlations based on experimental data (generally as functions of temperature) and due to their empirical nature a wide range of forms have been used in literature.

In order to support different forms for these calculations, the IDAES Generic Property Package Framework uses Python methods to define the form of pure component property calculations. This allows developers and users to easily enter the form they wish to use for their application with a minimum amount of code.

Naming Methods

The IDAES Generic Property Package Framework supports two ways of providing pure component property methods:

1. Providing the method directly - users may directly provide their method of choice as a config argument (*config.property_name*) in the *PropertyParameterBlock*, in which case the method can use any name the user desires.
2. Providing a library module - alternatively, users can provide a module containing a library of methods as the config argument (*config.property_name*), in which case the framework searches the module for a method with the same name as the property (and the config argument). E.g., for the property *enth_mol_phase_comp* the method name would be *enth_mol_phase_comp* (as would the associated config argument).

Method Arguments

Note: Currently, the IDAES Generic Property Package Framework assumes pure component property calculations will be a function of only temperature. If additional functionality is required, please contact the IDAES Developers.

Currently, all pure component property methods in the IDAES Generic Property Package Framework take three arguments:

1. A reference to the *StateBlock* where the method will be used (generally *self*),
2. An element of a component list,
3. A pointer to the *temperature* variable to be used in the calculation. By using a pointer rather than an absolute reference (i.e. *self.temperature*), this allows the method to be applied at different temperatures as necessary (e.g. the reference temperature).

Method Parameters

Pure component property methods all depend on a number of parameters, often derived from empirical data. In order to avoid duplication of parameters and facilitate parameter estimation studies, all property parameters are stored in the *PropertyParameterBlock* and each *StateBlock* contains a reference to its associated parameter block (*self.params*).

For pure component property methods, parameter names are defined in the associated methods thus developers can choose any name they desire. However, the IDAES standard is to use the name of the property appended with *_coeff* and developers are encouraged to follow this convention.

Method Body

The body of the pure component property method should assemble an expression describing the specified quantity for the component given in the method arguments. This expression should involve Pyomo components from the *StateBlock* (i.e. *self*), the associated *PropertyParameterBlock* (*self.params*) and be returned in the final step of the method.

Example

Below is an example of a pure component property method for the molar heat capacity of a component in the (ideal) gas phase with the form $c_{p,ig,j} = A + B \times T$.

```
def cp_mol_ig_comp(self, component, temperature):
    # Method named using standard naming convention
    # Arguments are self, a component and temperature

    # Return an expression involving temperature and parameters
    return (self.params.cp_mol_ig_comp_coeff[component, "A"] +
            self.params.cp_mol_ig_comp_coeff[component, "B"]*temperature)
```

Note that the method only returns an expression representing the R.H.S. of the correlation.

Developing Equation of State Modules

Contents

- *Developing Equation of State Modules*
 - *Equations of State and Multiple Phases*
 - *General Structure*
 - *Phase Equilibrium*
 - *Accessing Pure Component Property Methods*
 - *Common Methods*
 - *Mixture Property Methods*
 - *Example*

The central part of any property package are the equations of state or equivalent models which describe how the mixture behaves under the conditions of interest. For systems with multiple phases and phase equilibrium, each phase must have its own equation of state (or equivalent), which must provide information on phase equilibrium which is compatible with the other phases in the system.

Equations of State and Multiple Phases

The IDAES Generic Property Package Framework requires users to assign an equation of state module for each phase in their system, thus equations of state can be written for specific phases (e.g. an ideal gas equation of state). In some cases, developers may wish to write equations of state for multiple phases, and the generic framework supports this by indexing all properties by phase.

Developers are encouraged to add checks to their methods to ensure their equations of state are only applied to phases where they are appropriate (e.g. an ideal gas equation of state should raise an exception if the phase argument is not “Vap”).

General Structure

Equation of State Modules in the IDAES Generic Property Package Framework are files (modules) containing a number of methods which describe the behavior of the material. These methods define how each of the properties associated with a given phase should be calculated, and the list of properties supported for a given phase is limited by the methods provided by the developer of the equation of state.

Phase Equilibrium

When calculating phase equilibrium, the IDAES Generic Property Package Framework uses the general form $\Phi_{\text{phase } 1, j}^e = \Phi_{\text{phase } 2, j}^e$ where $\Phi_{p, j}^e$ is the fugacity of component j in phase p calculated at the equilibrium temperature (T_{eq} , variable name *self.teq*). The equilibrium temperature is calculated using the users’ choice of phase equilibrium formulation and determines how the property package will handle phase transitions.

All equation of state methods should contain a method for calculating fugacity if they are to support phase equilibrium calculations.

Accessing Pure Component Property Methods

In most cases, property calculations in the equation of state methods will require calculations of the pure component properties for the system. These can be accessed using *get_method* (imported from *idaes.property_models.core.generic.generic_property*) using the form *get_method(self, “property_name”)*. This will return the **method** defined by the user in the *PropertyParameterBlock* for the named property, which can then be used in the equation of state methods (note that users will need to call the method and provide it with the required arguments - generally *self*, component and a pointer to temperature).

Common Methods

For equations of state that support multiple phases, there may be certain calculations and/or variables that are common to all phases. To support this (and avoid duplication of these), equation of state methods should contain a method named *common* which implements any component which are common to multiple phases. This method should also contain checks to ensure that these components have not already been created for another phase in the system (to avoid duplication). In cases where there are no common components, this method can *pass*.

Mixture Property Methods

The main part of an equation of state method are a set of methods which describe properties of the mixture for a given phase. Any mixture property that the property package needs to support must be defined as a method in the equation of state module, which returns an expression for the given property (construction of the actual Pyomo component will be handled by the core framework code).

Mixture properties can be defined in any way the developer desires, and can cross-link and reference other mixture properties as required. Developers should recall that the State Definition method should have defined the following properties which can be used in mixture property correlations:

- pressure
- temperature
- mole_frac_phase_comp
- phase_frac

Other state variables **may** have been defined by the user's choice of State Definition, however this cannot be guaranteed. Developers may choose to assume that certain state variables will be present, but this will limit the application of their equation of state module to certain state definitions which should be clearly documented.

Example

Below is an example method for a method in an equation of state module for calculating molar density that supports both liquid and vapor phases.

```
def dens_mol_phase(b, phase):
    if phase == "Vap":
        return b.pressure/(b.params.gas_const*b.temperature)
    elif phase == "Liq":
        return sum(b.mole_frac_phase_comp[phase, j] *
                   get_method(b, "dens_mol_liq_comp")(b, j, b.temperature)
                   for j in b.params.component_list)
    else:
        raise PropertyNotSupportedError("Phase not supported")
```

Developing State Definitions

Contents

- *Developing State Definitions*
 - define_state(self)
 - * *State Variables*
 - * define_state_vars
 - * *Auxiliary Variables*
 - * *Supporting Constraints*
 - * always_flash
 - * get_material_flow_terms(phase, comp)

```

* get_enthalpy_flow_terms(phase)
* get_material_density_terms(phase, component)
* get_energy_density_terms(phase)
* get_material_flow_basis()
* default_material_balance_type()
* default_energy_balance_type()
* define_port_members()
* define_display_vars()
- state_initialization(self)
- self.do_not_initialize

```

The primary purpose of the State Definition method is to define the state variables which will be used to describe the state of the mixture in the property package. However, a number of other key aspects of the property package definition are tied to the choice of state variables and must be declared here as well.

State definitions are defined as Python modules with two methods and one list, which are describe below.

define_state(self)

The first method in a State Definition module is the *define_state* method. This method is used to define the state variables and associated components and methods. The *define_state* method must define the following things:

State Variables

The most important part of a State Definition module is the definition of the state variables that should be used in the resulting property package. The choice of state variables is up to the module developer, however the set of variables selected must contain sufficient information to fully define the extensive and intensive state of the material. That is, if all the state variables are fixed, the resulting set of variables and constraints should form a square problem (i.e. 0 degrees of freedom). Beyond this requirement however, developers may choose any combination of state variables they wish.

State variables should be defined as Pyomo *Vars* with names drawn from the IDAES naming standard, and should include initial values and bounds. The Generic Property Package Framework includes an optional user input of bounds for the state variables (*config.state_bounds*) which developers are encouraged to make use of when setting bounds and initializing variables.

define_state_vars

In order to inform the IDAES Process Modeling Framework of which variables should be considered state variable, developers are required to define a method named *define_state_vars*. This method should return a *dict* where the keys are a string identifier for each state variable and the values being pointers to the associated *Var* component. For example:

```

def define_state_vars_state_definition():
    return {"flow_mol": self.flow_mol,
            "mole_frac_comp": self.mole_frac_comp,
            "pressure": self.pressure,

```

(continues on next page)

(continued from previous page)

```

        "temperature": self.temperature,}
self.define_state_vars = define_state_vars_state_definition

```

Auxiliary Variables

Whilst the developer is free to choose any set of state variable they wish to define their system, there are certain properties/quantities associated with material state that are frequently used in process models. For example, most property calculation methods drawn upon empirical correlations for pure component properties which are most commonly expressed as functions of temperature (and sometimes pressure). Additionally, multiphase systems often require knowledge of the volume fractions of each phase present.

To ensure that these properties/quantities are available when required, it is required that State Definition modules define the following quantities if they are not already one of the state variables chosen:

- *temperature* - the temperature of the mixture,
- *pressure* - the pressure of the mixture,
- *mole_frac_phase_comp* - mole fraction of the mixture by phase and component (even if only one phase is present),
- *phase_frac* - volume fractions of each phase (even if only one phase is present).

These quantities can be defined as either Pyomo *Vars* with associated *Constraints*, or as Pyomo *Expressions* as the developer desires. Developers may choose to include additional auxiliary variables as required by their needs (e.g. different forms of flow rates).

Supporting Constraints

Depending upon the choice of state and auxiliary variables, developers may need to include a number of supporting constraints in their State Definitions. Common examples include constraints for the sum of mole fractions in the system, and relationships between different types of flow rates. Any number of constraints can be included by the developer to suit their needs, subject to the limitations of degrees of freedom.

However, developers need to be aware of the difference between inlet and outlet states and how this affects which constraints can be written. In the case of inlet states, all state variables are defined by the upstream process and thus no constraint can be written that involves only state variables (e.g. sum of mole fractions). For outlet (and intermediate) states however, it is often necessary to include these types of constraints to fully define the system. The IDAES Process Modeling Framework uses the *config.defined_state* configuration argument to indicate situations where the state variables should be considered fully defined (e.g. inlets) which can be used in *if* statements to determine whether a constraint should be included.

always_flash

Whilst the set of state variables chosen must be sufficient for fully defining the state of the material, depending on the set of state variables chosen information of the phase separation (if applicable) may or may not be explicitly included. For example, using total flow rate and composition along with pressure and specific enthalpy is sufficient to define the state of the material, however it does not explicitly describe the phase fractions of the system. In these cases, it is necessary to perform a flash calculation at every state in the system to determine the phase fractions. However, If the state is defined in terms of flow rates by phase and component along with pressure and specific enthalpy, information on the phase separation is already included in the state definition and flash calculations are not required where the state is fully defined (i.e. *config.state_defined* is True).

To inform the Generic Property Package Framework of whether phase equilibrium calculations should be included when *config.state_defined* is True, all State Definitions are required to include a component named *always_flash* which is a boolean indicating whether equilibrium calculations should always be included (True) or only included when the state is not fully defined (False).

get_material_flow_terms(phase, comp)

In order to automate the construction of the material balance equations, the IDAES Process Modeling Framework expects property packages to provide expressions for the flow terms in these equations. This is done via the *get_material_flow_terms* method which should return an expression involving variables in the StateBlock which should be used as the flow term in the material balances.

There are many forms this expression can take depending upon the state variables chosen and how the developer wishes to formulate the material balance equations, and the framework endeavors to support as many of these as possible. Material flow terms are defined on a phase-component basis (i.e. a separate expression for each component in each phase). An example of a *get_material_flow_term* using flow rate and mole fractions by phase is shown below.

```
def get_material_flow_terms_definition(phase, component):  
    return self.flow_mol_phase[phase] * self.mole_frac_phase_comp[phase, component]  
self.get_material_flow_terms = get_material_flow_terms_definition
```

get_enthalpy_flow_terms(phase)

In the same way that *get_material_flow_terms* is used to automate construction of the material balance equations, automating the construction of the energy balance equations requires a *get_enthalpy_flow_terms* method. This method should return an expression for the enthalpy flow terms involving variables in the StateBlock.

There are many forms for the enthalpy flow terms as well, and developers may choose whichever best suits their needs. Enthalpy flow terms are defined on a phase basis, and an example is shown below using flow rate and specific enthalpy by phase.

```
def get_enthalpy_flow_terms_definition(phase):  
    return self.flow_mol_phase[phase] * self.enth_mol_phase[phase]  
self.get_enthalpy_flow_terms = get_enthalpy_flow_terms_definition
```

get_material_density_terms(phase, component)

For dynamic system, calculation of the material holdups also requires a material density term which is defined using the *get_material_density_terms* method. This method is defined in a similar fashion to the *get_material_flow_terms* method and is also defined on a phase-component basis.

get_energy_density_terms(phase)

For dynamic system, calculation of the energy holdups also requires an energy density term which is defined using the *get_energy_density_terms* method. This method is defined in a similar fashion to the *get_enthalpy_flow_terms* method and is also defined on a phase basis. Note however that the energy density term should only include internal energy contributions, and not the full enthalpy density (i.e. excluding the PV term).

get_material_flow_basis()

To automate generation of some terms in the balance equations, the IDAES Process Modeling Framework needs to know the basis (mass, mole or other) of the flow terms. This is defined in the State Definition by providing a *get_material_flow_basis* method which returns a *MaterialFlowBasis Enum* (importable from *idaes.core*). E.g.:

```
def get_material_flow_basis_definition():
    return MaterialFlowBasis.molar
self.get_material_flow_basis = get_material_flow_basis_definition
```

default_material_balance_type()

The IDAES Process Modeling Framework allows property packages to specify a default form for the material balance equations to be used if the modeler does not specify a form. Whilst not strictly required, developers are strongly encouraged to define a default form for the material balance equations.

To set the default material balance type, the State Definition must implement a method which returns a *MaterialBalanceType Enum* (importable from *idaes.core*). E.g.:

```
def default_material_balance_type_definition():
    return MaterialBalanceType.componentTotal
self.default_material_balance_type = default_material_balance_type_definition
```

default_energy_balance_type()

The IDAES Process Modeling Framework allows property packages to specify a default form for the energy balance equations to be used if the modeler does not specify a form. Whilst not strictly required, developers are strongly encouraged to define a default form for the energy balance equations.

To set the default energy balance type, the State Definition must implement a method which returns an *EnergyBalanceType Enum* (importable from *idaes.core*). For an example, see *default_material_balance_type* above.

define_port_members()

In some situations, it is desirable to pass additional information between unit operations in a model beyond just the state variables. In these circumstance, the developer may define a *define_port_members* method which describes the information to be passed in *Ports* connecting units. This method should return a *dict* with a form similar to that of *define_state_vars*. Note that developers must also ensure that any additional information passed in *Ports* does not result in an over-specified problem, generally by excluding certain constraints in *StateBlocks* where *config.defined_state* is *True*.

If this method is not defined, *Ports* will default to using the variables described in *define_state_vars* instead.

define_display_vars()

Developers may also define a *define_display_vars* method which is used by the IDAES *report* methods to determine what information should be displayed for each state. The *define_display_vars* method should return a *dict* containing the information to display with the keys being the display name for the information and value being the quantity to display (similar to the *define_state_Vars* method). If this method is not defined then the *define_state_vars* method is used by the *report* methods instead.

state_initialization(self)

The *state_initialization* method is called as part of the Generic Property Package Framework *initialize* method and is expected to set initial guesses for any auxiliary variables defined by the State Definition based on the current values of the state variables. Note that the state variables will have been provided with initial guesses for the current state of the material from the process models, and thus will likely not be at their pre-defined initial conditions.

self.do_not_initialize

The *do_not_initialize* component is a list containing a list of *Constraint* names which should remain deactivated during initialization of the StateBlock and only reactivated during the final step on initialization. Common examples of these are those constraints that are only written for outlet Blocks (i.e. those when *config.defined_state* is False), such as overall sum of mole fraction constraints.

Developing Phase Equilibrium Methods

Contents

- *Developing Phase Equilibrium Methods*
 - *phase_equil(self)*
 - *phase_equil_initialization(self)*

Handling phase equilibrium and phase transitions within an equation oriented framework can be challenging as it is necessary to ensure that all constraints and variables has feasible solution at all states. When dealing with disappearing phases and correlations that can become ill-defined or singular outside of the two phase envelope, it is necessary to either bound the problem to the two-phase region or reformulate the problem.

The IDAES Generic Property Package Framework provides support for reformulating the problem by defining an “equilibrium temperature” (*self.teq*) at which all phase equilibrium calculations are performed. Issues surrounding phase transitions can be avoided by providing a definition for the equilibrium temperature that satisfies the following constraints:

$$T_{\text{bubble}} \leq T_{eq} \leq T_{\text{dew}}$$

The Phase Equilibrium module allows users to provide a definition for the equilibrium temperature, along with any necessary instructions on how to initialize the components associated with this definition.

A Phase Equilibrium module consists of two methods , which are described below.

phase_equil(self)

The *phase_equil* method is responsible for defining the variables and constraints necessary for calculating the equilibrium temperature, and at a minimum must contain one constraint relating the equilibrium temperature (*self._teq*) to the system temperature (*self.temperature*).

phase_equil_initialization(self)

This method is called by the Generic Property Package Framework initialization routine and should initialize the constraints associated with the phase equilibrium definition.

Note that the Generic Property Package Framework beings by deactivating all constraints in the *StateBlock* so the first step in the *phase_equil_initialization* method should be to activate any constraints defined in *phase_equil*. Additionally, this method may calculate initial values for any supporting variables defined in *phase_equil* based on variables that have already been initialized (primarily *temperature* and bubble and dew points if used). Developers should be careful however to fully understand the initialization sequence of the Generic Property Package Framework to understand which variables may have been initialized at this point.

Introduction

Note: The generic property package framework is still under development. Whilst the current framework is functional, features are still being developed and added.

The generic property package framework builds upon the existing framework for implementing property packages within IDAES, and will not prevent the use of custom written property packages in the future. Due to the complex nature of thermophysical property calculations, the generic property framework cannot support all possible materials and applications. Whilst it is hoped that the generic framework will be able to handle most common applications, users with more unusual systems or those solving computationally intensive problems may need to write custom property packages for their cases.

Property packages represent the core of any process model, and having a suitable property package is key to successfully modeling any process system. However, developing property packages is a significant challenge even for experienced modelers as they involve large numbers of tightly coupled constraints and parameters. The goal of the IDAES Generic Property Package Framework is to provide a flexible platform on which users can build property packages for common types of systems by calling upon libraries of modular sub-models to build up complex property calculations with the least effort possible.

The Generic Property Package Framework breaks down property packages into a number of components which can be assembled in a modular fashion. Users need only provide those components which they require for their system of interest, and components can be drawn from libraries of existing components or provided by the user as custom code. Details on how to set up the definition of a property package using the generic framework are given [here](#).

The components which make up a generic property package are as follows:

1. Choose a base set of *units of measurement* for the property package.
2. Define the *components* which make up the material of interest, including methods for calculating the pure component properties of interest in the system.
3. Define the *phases of interest* for the application, including equations of state and other phase specific decisions.
4. Choose the set of *state variables* you wish to use and a reference state for the system.
5. (Optional) Define any *phase equilibria* which occurs in the system and methods associated with calculating this.

6. (Optional) A number of *global options* are available for further customizing behavior of certain property calculations.

The following sections will describe how to define a property package using the Generic Property Package Framework along with the libraries of sub-models currently available. Finally, the *developers* section describes how to go about defining your own custom components to use when creating custom property packages.

Note: Within most IDAES models “parameters” are in fact defined as Pyomo ‘Vars’ (i.e. variables) which are fixed at their defined values. Whilst *Params* would seem to be the logical choice for these, parameter estimation problems require the parameters being estimated to be defined as *Vars* so that the solver is free to vary them.

Generic Reaction Package Framework

Contents

Defining Reaction Packages

Contents

- *Defining Reaction Packages*
 - *Introduction*
 - *Units of Measurement*
 - *Config Dictionary*
 - *Class Definition*

Introduction

In order to create and use a property package using the IDAES Generic Reaction Package Framework, users must provide a definition for the system they wish to model. The framework supports two approaches for defining the property package, which are described below, both of which are equivalent in practice.

Units of Measurement

As with generic thermophysical property packages, when defining a reaction package using the generic framework users must define the base units for the reaction package (see [link](#)). The approach for setting the base units and units for all parameters is the same as for thermophysical property packages and depends on the approach used to define the reaction package.

Config Dictionary

The most common way to use the Generic Reaction Package Framework is to create an instance of the *GenericReactionParameterBlock* component and provide it with a dictionary of configuration arguments, as shown below:

```
m = ConcreteModel()

m.fs = FlowsheetBlock()

m.fs.thermo_properties = PhysicalParameterBlock()

m.fs.reaction_properties = GenericReactionParameterBlock(default={"property_package":_
↳m.fs.thermo_properties, config_dict})
```

In the above example, the *PhysicalParameterBlock* object can be from any thermophysical property package suitable for the user's application.

Users need to populate *config_dict* with the desired options for their system as described in the other parts of this documentation. An example of a configuration dictionary can be found later on this page. For details on each configuration option, please see the relevant documentation.

Using this approach, units of measurement are defined using the *base_units* option in the configuration dictionary. Users must provide units for the 5 core quantities, and may also provide units for the other 2 SI base quantities (if required). For details on other configuration options, please see the relevant documentation.

Linking to a Thermophysical Property Package

As state information is defined by thermophysical property packages in IDAES, each reaction package must be linked to an appropriate thermophysical property package. This linkage is used by the reaction package to find the state information required to calculate the reaction properties, and thus the thermophysical property package must support all the properties required by the reaction package.

Setting Reaction Basis

Many reaction properties (e.g. reaction rates) can be defined on different bases, such as a mass or molar basis. All properties within a package must use the same basis, which can be set using the "reaction_basis" configuration argument (see below). This must be done using the *MaterialFlowBasis Enum*, which can be imported from *idaes.core*.

Configuration Example

```
from pyomo.environ import units as pyunits

from idaes.core import MaterialFlowBasis

config_dict = {
    "base_units": {"time": pyunits.s,
                  "length": pyunits.m,
                  "mass": pyunits.kg,
                  "amount": pyunits.mol,
                  "temperature": pyunits.K},
    "rate_reactions": {
        "R1": {"stoichiometry": {("Liq", "A"): -1,
```

(continues on next page)

(continued from previous page)

```

        ("Liq", "B"): -1,
        ("Liq", "C"): 2},
    "heat_of_reaction": constant_dh_rxn,
    "rate_constant": arrhenius,
    "rate_form": power_law_rate,
    "concentration_form": ConcentrationForm.moleFraction,
    "parameter_data": {
        "dh_rxn_ref": (-10000, pyunits.J/pyunits.mol),
        "arrhenius_const": (1, pyunits.mol/pyunits.m**3/pyunits.s),
        "energy_activation": (1000, pyunits.J/pyunits.mol)}},
    "equilibrium_reactions": {
        "R2": {"stoichiometry": {("Liq", "B"): -1,
                                ("Liq", "C"): -1,
                                ("Liq", "D"): 1},
            "heat_of_reaction": constant_dh_rxn,
            "equilibrium_constant": van_t_hoff,
            "equilibrium_form": power_law_equil,
            "concentration_form": ConcentrationForm.moleFraction,
            "parameter_data": {
                "dh_rxn_ref": (-20000, pyunits.J/pyunits.mol),
                "k_eq_ref": (100, None),
                "T_eq_ref": (350, pyunits.K)}}}}

```

Class Definition

Alternatively, the IDAES Generic Reaction Package Framework supports defining classes derived from the IDAES *GenericReactionParameterData* class with methods for defining configuration options and parameters.

Users can define two methods which are called automatically when an instance of the property package is created:

1. *configure*, which defines the users selection of sub-models, and
2. *parameters*, which defines the parameters necessary for the selected property methods.

A basic outline of a user defined Reaction Parameter Block is shown below.

```

@declare_process_block_class("UserReactionParameterBlock")
class UserReactionParameterData(GenericReactionParameterData):
    def configure(self):
        # Set configuration options
        self.config.option_1 = value

    def parameters(self):
        # Define parameters
        self.param_1 = Var(index_set, initialize=value)

```

Users should populate the *configure* and *parameters* methods as discussed below.

Configure

The ‘configure’ method is used to assign values to the configuration arguments, using the format *self.config.option_name = value*. Users will also need to set the units of measurement in the property package meta-data.

Parameters

The *parameters* method is used to construct all the parameters associated with the property calculations and to specify values for these. The list of necessary parameters is based on the configuration options and the selected methods. Each method lists their necessary parameters in their documentation. Users need only define those parameters required by the options they have chosen.

Defining Rate-Based Reactions

The *rate_reactions* Argument

Each *GenericReactionParameterBlock* has a configuration argument named *rate_reactions* which is used to define rate-based reactions and specify how to calculate properties associated with these. The *rate_reactions* configuration argument is expected to be a dict-of-dicts, where the keys are the names for the rate-based reactions, and the values are a dict of configuration arguments for that reaction. Note that reaction names must be unique across both rate-based and equilibrium reactions, as all reactions are indexed by name.

```
"rate_reactions": {
    "reaction_1": {options},
    "reaction_2": {options}}
```

Configuration Arguments

The configuration arguments for each rate-based reaction are used to define methods for calculating reaction properties and defining the parameters associated with these. A full list of the supported configuration arguments is given below:

- stoichiometry (required)
- rate_form (required)
- concentration_form
- heat_of_reaction
- rate_constant

Stoichiometry

The *stoichiometry* configuration argument is used to define which components take part in a reaction, and is a required argument. The *stoichiometry* argument should be a dict where the keys are phase-component pairs and the values are the stoichiometric coefficient for that pair. Users need only provide values for those components that take part in the reaction - all undeclared phase-component pairs will be assumed to have a value of 0. An example of defining the reaction stoichiometry is given below, where in phase_1 component_1 is converted to component_2 in a 1:1 ratio:

```
"stoichiometry": {  
    ("phase_1", "component_1"): -1,  
    ("phase_1", "component_2"): 1}
```

Concentration Form

Many common rate forms can be written using a number of different bases, such as molarity, molality or partial pressure. The *concentration_form* configuration argument is used in these cases to determine what basis to use for the concentration terms in the rate form and automatically write the correct expression (and determine units for the associated parameters. The *concentration_form* configuration argument must be an instance of a *ConcentrationForm Enum* (imported from `idaes.generic_models.properties.core.generic.utility`), and the following forms are currently available:

- molarity: `ConcentrationForm.molarity`
- activity: `ConcentrationForm.activity`
- molality: `ConcentrationForm.molality`
- mole fractions: `ConcentrationForm.moleFraction`
- mass fractions: `ConcentrationForm.massFraction`
- partial pressure: `ConcentrationForm.partialPressure`

Other Reaction Properties

The remaining configuration arguments are used to define how different properties should be calculated for each reaction. The *rate_form* argument is required, however all other properties need only be defined if needed for the user's application. These arguments should be provided as either Python functions or classes;

- functions are used for self-contained correlations with hard-coded parameters,
- classes are used for more generic correlations which require associated parameters.

A list of the libraries of methods available in the IDAES Framework can be found [here](#).

Defining Equilibrium Reactions

The *equilibrium_reactions* Argument

Each *GenericReactionParameterBlock* has a configuration argument named *equilibrium_reactions* which is used to define equilibrium reactions and specify how to calculate properties associated with these. The *equilibrium_reactions* configuration argument is expected to be a dict-of-dicts, where the keys are the names for the equilibrium reactions, and the values are a dict of configuration arguments for that reaction. Note that reaction names must be unique across both rate-based and equilibrium reactions, as all reactions are indexed by name.

```
"equilibrium_reactions": {  
    "reaction_1": {options},  
    "reaction_2": {options}}
```

Configuration Arguments

The configuration arguments for each equilibrium reaction are used to define methods for calculating reaction properties and defining the parameters associated with these. A full list of the supported configuration arguments is given below:

- stoichiometry (required)
- equilibrium_form (required)
- concentration_form
- heat_of_reaction
- equilibrium_constant

Stoichiometry

The *stoichiometry* configuration argument is used to define which components take part in a reaction, and is a required argument. The *stoichiometry* argument should be a dict where the keys are phase-component pairs and the values are the stoichiometric coefficient for that pair. Users need only provide values for those components that take part in the reaction - all undeclared phase-component pairs will be assumed to have a value of 0. An example of defining the reaction stoichiometry is given below, where in phase_1 component_1 is converted to component_2 in a 1:1 ratio:

```
"stoichiometry": {
    ("phase_1", "component_1"): -1,
    ("phase_1", "component_2"): 1}
```

Concentration Form

See [rate reaction](#) documentation.

Other Reaction Properties

The remaining configuration arguments are used to define how different properties should be calculated for each reaction. The *equilibrium_form* argument is required, however all other properties need only be defined if needed for the user's application. These arguments should be provided as either Python functions or classes;

- functions are used for self-contained correlations with hard-coded parameters,
- classes are used for more generic correlations which require associated parameters.

A list of the libraries of methods available in the IDAES Framework can be found [here](#).

Reaction Module Libraries

The following libraries are available for defining reaction parameters as part of the Generic Reaction Package Framework.

Rate Constant Forms

Contents

- *Rate Constant Forms*
 - *Arrhenius Equation (arrhenius)*

Arrhenius Equation (arrhenius)

The method uses the Arrhenius equation to calculate the rate constant.

$$k_{rxn} = Ae^{\frac{-E_A}{RT}}$$

Parameters

Symbol	Parameter Name	Units	Description
A	arrhenius_const	Varies, based on reaction form	Pre-exponential factor
E_A	energy_activation	Base units	Activation energy

Rate-Based Reaction Forms

Contents

- *Rate-Based Reaction Forms*
 - *Power Law (power_law_rate)*

Power Law (power_law_rate)

The method uses a power law form using the concentration form provided to calculate the reaction rate.

$$r_{rxn} = k_{rxn} \prod_{(p,j)} C_{(p,j)}^{O_{(p,j)}}$$

Parameters

Symbol	Parameter Name	Indices	Description
O	reaction_order	phase, component	Reaction order

Providing a *reaction_order* dict is optional. If one is not provided, it will be assumed that this is an elementary reaction and that the reaction order is equal to the stoichiometric coefficient for the products (i.e. for all phase-component pairs with a *negative* stoichiometric coefficient, the reaction order is equal to the absolute value of the stoichiometric coefficient).

Equilibrium Constant Forms

Contents

- *Equilibrium Constant Forms*
 - *van 't Hoff Equation (van_t_hoff)*

van 't Hoff Equation (van_t_hoff)

The method uses the van 't Hoff equation to calculate the equilibrium constant.

$$k_{eq} = k_{eq,ref} e^{-\left(\frac{\Delta H_{rxn}}{R}\right)\left(\frac{1}{T} - \frac{1}{T_{ref,eq}}\right)}$$

Parameters

Sym-bol	Parameter Name	Units	Description
$k_{eq,ref}$	k_eq_ref	Varies, depends on equilibrium form	Equilibrium constant at reference temperature
$T_{ref,eq}$	tempera-ture_eq_ref	Base units	Reference temperature for calculating equilibrium constant

Equilibrium Reaction Forms

Contents

- *Equilibrium Reaction Forms*
 - *Power Law (power_law_equil)*

Power Law (power_law_equil)

The method uses a power law form using the concentration form provided to calculate the reaction rate.

$$k_{eq} = \prod_{(p,j)} x_{(p,j)}^{O_{(p,j)}}$$

Parameters

Symbol	Parameter Name	Indices	Description
O	reaction_order	phase, component	Reaction order

Providing a *reaction_order* dict is optional. If one is not provided, it will be assumed that this is an elementary reaction and that the reaction order is equal to the stoichiometric coefficient for all component in non-solid phases (the contribution of solid phases is assumed to be constant and included in the equilibrium constant, thus an order of zero is assumed).

Heat of Reaction Forms

Contents

- *Heat of Reaction Forms*
 - *Constant Heat of Reaction (`constant_dh_rxn`)*

Constant Heat of Reaction (`constant_dh_rxn`)

The simplest form for calculating the heat of reaction, this method assumes a constant value provided by the user.

$$\Delta h_{rxn} = \Delta h_{rxn,ref}$$

Parameters

Symbol	Parameter Name	Indices	Description
$\Delta h_{rxn,ref}$	dh_rxn_ref		Heat of reaction

Introduction

Note: The generic reaction package framework is still under development. Whilst the current framework is functional, features are still being developed and added.

The generic reaction package framework builds upon the existing framework for implementing reaction packages within IDAES, and will not prevent the use of custom written reaction packages in the future. Whilst it is hoped that the generic framework will be able to handle most common applications, users with more unusual systems or those solving computationally intensive problems may need to write custom reaction packages for their cases.

The Generic Reaction Package Framework breaks down reaction packages into a number of components which can be assembled in a modular fashion. Users need only provide those components which they require for their system of interest, and components can be drawn from libraries of existing components or provided by the user as custom code. Details on how to set up the definition of a reaction package using the generic framework are given [here](#).

The components which make up a generic reaction package are as follows:

1. Choose a base set of *units of measurement* for the property package.
2. *Associate* the reaction package with an appropriate thermodynamic property package. The thermodynamic property package must use the same set of base units of measurement,
3. Define the *basis of the reaction terms* for the reaction package.
4. Define the *rate-based reactions* of interest in the system.
5. Define the *equilibrium-based reactions* of interest in the system. Note that phase equilibrium is generally handled in the thermodynamic property package.

The following sections will describe how to define a reaction package using the Generic Reaction Package Framework along with the libraries of sub-models currently available. Finally, the *developers* section describes how to go about defining your own custom components to use when creating custom property packages.

Note: Within most IDAES models “parameters” are in fact defined as Pyomo ‘Vars’ (i.e. variables) which are fixed at their defined values. Whilst *Params* would seem to be the logical choice for these, parameter estimation problems require the parameters being estimated to be defined as *Vars* so that the solver is free to vary them.

Property packages provide the relationships and parameters necessary to determine the properties of process streams. They may be general in purpose, such as ideal gas equations, or specific to a certain application. Property packages are separated into two categories:

- physical and transport properties
- chemical reaction properties

While several standard property packages are provided in the IDAES model libraries, many process modeling applications will require specific property packages. Information on developing custom property packages is provided in the [advanced user guide](#).

Since the effort to develop a custom property package is substantial, the IDAES modeling framework provides a [Generic Property Package Framework](#) and [Generic Reaction Package Framework](#) to make it easier to create a package for common property and reaction models.

Units of Measurement

One of the most important roles property packages play within the modeling framework is to define the units of measurement that will be used for those models which use the property packages. Any variable which is created in a unit model will derive its units of measurement from those defined in the associated property package in order to ensure consistency of units.

Defining units of measurement in property packages is [discussed here](#).

Physical properties

Almost all process models depend on physical properties to some extent, such as calculation of specific enthalpy or internal energy for energy balances. These properties only depend on the material being considered and are independent of the unit operations in which they are used. As such, physical property calculations can be separated from the unit model calculations and treated as a separate submodel which is called by the unit model. Each unit model can then create instances of these submodels as required to calculate those properties required by each unit.

Within IDAES, this is handled by StateBlock objects – these are self-contained submodels containing the calculations for all necessary thermophysical properties for a given material at a given point in space and time. IDAES UnitModels create instances of these StateBlocks wherever they need to calculate physical properties and link to variables within the StateBlock within the unit model constraints.

However, physical property calculations depend on a set of parameters which are specific to a given material or mixture. Thus, each instance of a StateBlock for a material use the same set of parameters. To avoid duplicating these parameters in every instance of a StateBlock for a given material, these parameters are instead grouped in a PhysicalParameterBlock for that material which the StateBlocks link to. In this way, there is a single common location for all parameters.

In summary, physical property packages consist of two parts:

- [PhysicalParameterBlocks](#), which contain a set of parameters associated with the specific material(s) being modeled
- [StateBlocks](#), which contain the actual calculations of the state variables and functions

Reaction properties

Reaction property packages represent a collection of calculations necessary to determine the reaction behavior of a mixture at a given state. Reaction properties depend upon the state and physical properties of the material, and thus must be linked to a StateBlock which provides the necessary state and physical property information.

Reaction property packages consist of two parts:

- *ReactionParameterBlocks*, which contain a set of parameters associated with the specific reaction(s) being modeled, and
- *ReactionBlocks*, which contain the actual calculations of the reaction behavior.

Component and Phase Objects

Property packages also rely on component and phase objects.

Component Objects are used to identify the chemical species of interest in a property package and to contain information describing the behavior of that component (such as properties of that component).

Phase Objects are used to identify the thermodynamic phases of interest in a property package and to contain information describing the behavior of that phase (for example the equation of state which describes that phase).

As Needed Properties

Process flow sheets often require a large number of properties to be calculate, but not all of these are required in every unit operation. Calculating additional properties that are not required is undesirable, as it leads to larger problem sizes and unnecessary complexity of the resulting model.

To address this, IDAES supports “as needed” construction of properties, where the variables and constraints required to calculate a given quantity are not added to a model unless the model calls for this quantity. To designate a property as an “as needed” quantity, a method can be declared in the associated property BlockData class (StateBlockData or ReactionBlockData) which contains the instructions for constructing the variables and constraints associated with the quantity (rather than declaring these within the BlockData’s build method). The name of this method can then be associated with the property via the add_properties metadata in the property packages ParameterBlock, which indicates that when this property is called for, the associated method should be run.

The add_properties metadata can also indicate that a property should always be present (i.e. constructed in the BlockData’s build method) by setting the method to *None*, or that it is not supported by setting the method to *False*.

Generic Property Package Framework

Property packages represent the core of any process model, and having a suitable property package is key to successfully modeling any process system. However, developing property packages is a significant challenge even for experienced modelers as they involve large numbers of tightly coupled constraints and parameters. The *Generic Property Package Framework* was designed to help users build property packages with the least effort possible by leveraging libraries of modular sub-models that include common types of property calculations.

Generic Reaction Package Framework

Similar to the Generic Property Package Framework, the *Generic Reaction Package Framework* helps users create reaction property packages for common systems.

Unit Model

Control Volume

- *Overview*
- *Common Control Volume Tasks*
- *Setting up the time domain*
- *Getting Property Package Information*
- *Collecting Indexing Sets for Property Package*
- *ControlVolume and ControlVolumeBlockData Classes*

Overview

Control volumes serve as the fundamental building block of all unit operations. Control Volumes represent a single, well-defined volume of material over which material, energy and/or momentum balances will be performed.

The IDAES ControlVolume classes are designed to facilitate the construction of these balance equations by providing the model developer with a set of pre-built methods to perform the most common tasks in developing models of unit operations. The ControlVolume classes contain methods for creating and linking the necessary property calculations and writing common forms of the balance equations so that the model developer can focus their time on the aspects that make each unit model unique.

The IDAES process modeling framework currently supports two types of ControlVolumes:

- *ControlVolume0DBlock* represents a single well-mixed volume of material with a single inlet and a single outlet. This type of control volume is sufficient to model most inlet-outlet type unit operations which do not require spatial discretization.
- *ControlVolume1DBlock* represents a volume with spatial variation in one dimension parallel to the material flow. This type of control volume is useful for representing flow in pipes and simple 1D flow reactors.

Common Control Volume Tasks

All of the IDAES ControlVolume classes are built on a common core ControlVolumeBlockData which defines a set of common tasks required for all Control Volumes. The more specific ControlVolume classes then build upon these common tasks to provide tools appropriate for their specific application.

All ControlVolume classes begin with the following tasks:

- Determine if the ControlVolume should be steady-state or dynamic.
- Get the time domain.
- Determine whether material and energy holdups should be calculated.
- Collect information necessary for creating StateBlocks and ReactionBlocks.

- Create references to `phase_list` and `component_list` Sets in the `PhysicalParameterBlock`

Setting up the time domain

The first common task the `ControlVolumeBlock` performs is to determine if it should be dynamic or steady-state and to collect the time domain from the `UnitModel`. `ControlVolumeBlocks` have an argument `dynamic` which can be provided during construction which specifies if the Control Volume should be dynamic (`dynamic=True`) or steady-state (`dynamic=False`). If the argument is not provided, the `ControlVolumeBlock` will inherit this argument from its parent `Unit` model.

Finally, the `ControlVolume` checks that the `has_holdup` argument is consistent with the `dynamic` argument, and raises a `ConfigurationError` if it is not.

Getting Property Package Information

If a reference to a property package was not provided by the `UnitModel` as an argument, the `Control Volume` first checks to see if the unit model has a `property_package` argument set, and uses this if present. Otherwise, the `ControlVolumeBlock` begins searching up the model tree looking for an argument named `default_property_package` and uses the first of these that it finds. If no `default_property_package` is found, a `ConfigurationError` is returned.

Collecting Indexing Sets for Property Package

The final common step for all `ControlVolumes` is to collect any required indexing sets from the physical property package (for example component and phase lists). These are used by the `Control Volume` for determining what balance equations need to be written, and what terms to create.

The indexing sets the `ControlVolume` looks for are:

- `component_list` - used to determine what components are present, and thus what material balances are required
- `phase_list` - used to determine what phases are present, and thus what balance equations are required

ControlVolume and ControlVolumeBlockData Classes

A key purpose of `ControlVolumes` is to automate as much of the task of writing a unit model as possible. For this purpose, `ControlVolumes` support a number of methods for common tasks model developers may want to perform. The specifics of these methods will be different between different types of `ControlVolumes`, and certain methods may not be applicable to some types of `Control Volumes` (in which case a `NotImplementedError` will be returned). A full list of potential methods is provided here, however users should check the documentation for the specific `Control Volume` they are using for more details on what methods are supported in that specific `Control Volume`.

A key feature of the IDAES Core Modeling Framework is the use of `ControlVolumeBlocks`. `ControlVolumes` represent a volume of material over which material, energy and/or momentum balances can be performed. `ControlVolumeBlocks` contain methods to automate the task of writing common forms of these balance equations. `ControlVolumeBlocks` can also automate the creation of `StateBlocks` and `ReactionBlocks` associated with the control volume.

`Unit` models represent pieces of equipment and their processes. These models contain the unit performance constraints and associated variables for the equipment, such as:

- constraints relating balance terms to physical phenomena or properties (e.g. relating extent of reaction to reaction rate and volume)
- constraints describing flow of material into or out of unit (e.g. pressure driven flow constraints)

- unit level efficiency constraints (e.g. relating mechanical work to fluid work)

IDAES includes libraries of UnitModel classes. These models are composed of the following components:

1. *ControlVolumeBlocks*, which represent volume of material over which we wish to perform material, energy and/or momentum balances
2. *StateBlocks* and *ReactionBlocks*, which represent the thermophysical, transport and reaction properties of the material at a specific point in space and time
3. Inlets and Outlets, which allow UnitModels to connect to other UnitModels

Data Management Framework

DMF Overview

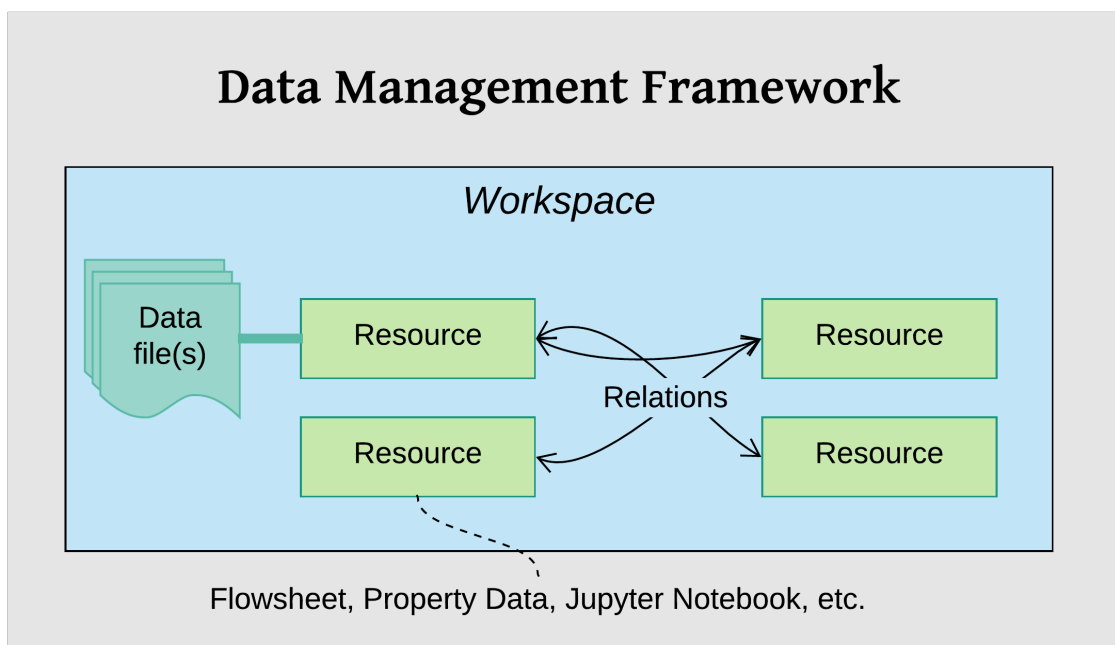
The Data Management Framework (DMF) is used to manage all the data needed by the IDAES framework, including flowsheets, models, and results. It stores metadata and data in persistent storage. It does not require that the user run a server or connect to a remote service. The DMF can be accessed through its *API* or *command-line interfaces*.

- *Concepts*
- *Configuration*
- *Jupyter notebook usage*
- *Sharing*
- *Reference*

Concepts

The DMF is designed to allow multiple separate threads of work. These are organized in *workspaces*. Inside a given workspace, all the information is represented by containers called *resources*. A resource describes some data in the system in a standard way, so it can be searched and manipulated by the rest of the IDAES framework. Resources can be connected to each other with *relations* such as “derived”, “contains”, “uses”, and “version”.

Below is an illustration of these components.



Configuration

The DMF is configured with an optional global configuration file and a required per-workspace configuration file. By default the global file is looked for as `.dmf` in the user's home directory. Its main function at the moment is to set the default workspace directory with the `workspace` keyword. For example:

```
# global DMF configuration
workspace: ~/data/workspaces/workspace1
```

The per-workspace configuration has more options. See the documentation in the `Workspace` class for details. The configuration file is in YAML (or JSON) format. Here is an example file, with some description in comments:

```
settings:                                     # Global settings
  workspace: /home/myuser/ws                 # Path to current workspace
workspace:                                   # Per-workspace settings
  location: /home/myuser/ws                  # Path to this workspace
  name: myws                                # Name of this workspace
  description: my workspace                  # Description (if any) of this workspace
  created: 2019-04-09 12:55:05               # Date workspace was created
  modified: 2019-04-09 12:55:05             # Date workspace was modified
  files:                                     # Basic information about data files
    count: 3                                # How many files
    total_size: 1.3 MB                       # Total size of the files
  html_documentation_paths:                 # List of paths for HTML documentation
    -: /home/myuser/idaes/docs/build
  logging:                                  # Logging configuration
    idaes.dmf:                               # Name of the logger
      level: DEBUG                           # Log level (Python logging constant)
      output: /tmp/debug.log                 # File path or "_stdout_" or "_stderr_"
```

This configuration file is used whether you use the DMF from the command-line, Jupyter notebook, or in a Python

program. For details see the DMF package documentation.

Jupyter notebook usage

In the Jupyter Notebook, there are some “magics” defined that make initializing the DMF pretty easy. For example:

```
from idaes.dmf import magics
%dmf init path/to/workspace
```

The code above loads the “%dmf” *line magic* in the first line, then uses it to initialize the DMF with the workspace at “path/to/workspace”.

From there, other “line magics” will operate in the context of that DMF workspace.

- `%dmf help` - Provide help on IDAES objects and classes. See [dmf-help](#).
- `%dmf info` - Provide information about DMF current state for whatever ‘topics’ are provided
- `%dmf list` - List resources in the current workspace
- `%dmf workspaces` - List DMF workspaces; you can do this *before* `%dmf init`

DMF help

The IDAES Python interfaces are documented with [Sphinx](#). This includes automatic translation of the comments and structure of the code into formatted and hyperlinked HTML pages. The `%dmf help` command lets you easily pull up this documentation for an IDAES module, class, or object. Below are a couple of examples:

```
# Initialize the DMF first
from idaes.dmf import magics
%dmf init path/to/workspace create

# Get help on a module (imported)
from idaes.core import control_volume0d
%dmf help control_volume0d

# Get help on a module (by name, no import)
%dmf help idaes.core.control_volume0d

# Get help on a class
from idaes.core.control_volume0d import ControlVolume0DBlock
%dmf help ControlVolume0DBlock

# Get help on a class (by name, no import)
%dmf help idaes.core.control_volume0d.ControlVolume0DBlock

# Get help on an object (will show help for the object's class)
# This will end up showing the same help as the previous two examples
obj = control_volume0d.ControlVolume0DBlock()
%dmf help obj
```

The help pages will open in a new window. The location of the built documentation that they use is configured in the per-workspace DMF configuration under the `htmldocs` keyword (a default value is filled in when the DMF is first initialized).

Sharing

The contents of a DMF workspace can be shared quite simply because the data is all contained within a directory in the local file system. So, some ways to share (with one or many people) include:

- Put the workspace directory in a cloud/shared drive like [Dropbox](#) , [Box](#) , [Google Drive](#) , or [OneDrive](#) .
- Put the workspace directory under version control like [Git](#) and share that versioned data using Git commands and a service like [Github](#) , [BitBucket](#) or [Gitlab](#).
- Package up the directory with a standard archiving utility like “zip” or “tar” and share it like any other file (e.g. attach it to an email).

Note: These modes of sharing allow users to see the same data, but are not designed for real-time collaboration (reading and writing) of the same data. That mode of operation requires a proper database server to mediate operations on the same data. This is in the roadmap for the DMF, but not currently implemented.

Reference

See the `idaes.dmf` package documentation that is generated automatically from the source code.

DMF Command-line Interface

This page lists the commands and options for the DMF command-line interface, which is a Python program called *dmf*. There are several usage examples for each sub-command. These examples assume the UNIX *bash* shell.

- *dmf*
 - *dmf options*
 - *dmf usage*
 - *dmf subcommands*
 - *usage overview*
- *dmf find*
 - *dmf find options*
 - *dmf find usage*
- *dmf info*
 - *dmf info options*
 - *dmf info usage*
- *dmf init*
 - *dmf init options*
 - *dmf init usage*
- *dmf ls*
 - *dmf ls options*

- *dmf ls usage*
- *dmf register*
 - *dmf register options*
 - *dmf register usage*
- *dmf related*
 - *dmf related options*
 - *dmf related usage*
- *dmf rm*
 - *dmf rm options*
 - *dmf rm usage*
- *dmf status*
 - *dmf status options*
 - *dmf status usage*

dmf

Data management framework command wrapper. This base command has some options for verbosity that can be applied to any sub-command.

dmf options

-v

--verbose

Increase verbosity. Show warnings if given once, then info, and then debugging messages.

-q

--quiet

Increase quietness. If given once, only show critical messages. If given twice, show no messages.

dmf usage

Run sub-command with logging at level “error”:

```
$ dmf <sub-command>
```

Run sub-command and log warnings:

```
$ dmf <sub-command>
```

Run sub-command and log informational / warning messages:

```
$ dmf -vv <sub-command>
```

Run sub-command only logging fatal errors:

```
$ dmf -q <sub-command>
```

Run sub-command with no logging at all:

```
$ dmf -qq <sub-command>
```

dmf subcommands

The subcommands are listed alphabetically below. For each, keep in mind that any unique prefix of that command will be accepted. For example, for `dmf init`, the user may also type `dmf ini`. However, `dmf in` will not work because that would also be a valid prefix for `dmf info`.

In addition, there are some aliases for some of the sub-commands:

- `dmf info` => *dmf resource* or *dmf show*
- `dmf ls` => *dmf list*
- `dmf register` => *dmf add*
- `dmf related` => *dmf graph*
- `dmf rm` => *dmf delete*
- `dmf status` => *dmf describe*

usage overview

To give a feel for the context in which you might actually run these commands, below is a simple example that uses each command:

```
# create a new workspace
$ dmf init ws --name workspace --desc "my workspace" --create
Configuration in '/home/dang/src/idaes/dangunter/idaes-dev/docs/ws/config.yaml

# view status of the workspace
$ dmf status
settings:
  workspace: /home/myuser/ws
workspace:
  location: /home/myuser/ws
  name: workspace
  description: my workspace
  created: 2019-04-20 08:32:59
  modified: 2019-04-20 08:32:59

# add some resources from files
$ echo "one" > oldfile ; echo "two" > newfile
$ dmf register oldfile --version 0.0.1
2792c0ceb0734ed4b302c44884f2d404
$ dmf register newfile --version 0.0.2 --prev 2792c0ceb0734ed4b302c44884f2d404
6ddee9bb2bb3420ab10aaf4c74d186f6

# list the current workspace contents
$ dmf ls
```

(continues on next page)

(continued from previous page)

```

id    type desc    modified
2792  data oldfile 2019-04-20 15:33:11
6dde  data newfile 2019-04-20 15:33:23

# look at one one resource (newfile)
$ dmf info 6dde
                                Resource 6ddee9bb2bb3420ab10aaf4c74d186f6

created
  '2019-04-20 15:33:23'
creator
  name: dang
datafiles
  - desc: newfile
    is_copy: true
    path: newfile
    sha1: 7bbef45b3bc70855010e02460717643125c3beca
datafiles_dir
  /home/myuser/ws/files/8027bf92628f41a0b146a5167d147e9d
desc
  newfile
doc_id
  2
id_
  6ddee9bb2bb3420ab10aaf4c74d186f6
modified
  '2019-04-20 15:33:23'
relations
  - 2792c0ceb0734ed4b302c44884f2d404 --[version]--> ME
type
  data
version
  0.0.2 @ 2019-04-20 15:33:23

# see relations
$ dmf related 2792
2792 data
├──version└─ 6dde data -

# remove the "old" file
$ dmf rm 2792
id                                type desc    modified
2792c0ceb0734ed4b302c44884f2d404 data oldfile 2019-04-20 15:33:11
Remove this resource [y/N]? y
resource removed

$ dmf ls
id    type desc    modified
6dde  data newfile 2019-04-20 15:33:23

```

dmf find

Search for resources by a combination of their fields. Several convenient fields are provided. At this time, a comprehensive capability to search on any field is not available.

dmf find options

In addition to the options below, this command also accepts all the *dmf ls options*, although the `--color/`
`--no-color` option is ignored for JSON output.

--output value

Output style/format. Possible values:

list (Default) Show results as a listing, as from the *ls* subcommand.

info Show results as individual records, as from the *info* subcommand.

json Show results are JSON objects

--by value

Look for “value” in the value of the *creator.name* field.

--created value

Use “value” as a date or date range and filter on records that have a *created* date in that range. Dates should be in the form:

`YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]]][+HH:MM[:SS[.fffff]]]`

To indicate a date range, separate two dates with a “..”.

- 2012-03-19: On March 19, 2012
- 2012-03-19..2012-03-22: From March 19 to March 22, 2012
- 2012-03-19..: After March 19, 2012
- ..2012-03-19: Before March 19, 2012

Note that times may also be part of the date strings.

--file value

Look for “value” in the value of the *desc* field in one of the *datafiles*.

--modified value

Use “value” as a date or date range and filter on records that have a *modified* date in that range. See `--created` for details on the date format.

--name value

Look for “value” as one of the values of the *alias* field.

--type value

Look for “value” as the value of the *type* field.

dmf find usage

By default, find will essentially provide a filtered listing of resources. If used without options, it is basically an alias for *ls*.

```
$ dmf ls
id  type desc      modified
2517 data file1.txt 2019-04-29 17:29:00
344c data file2.txt 2019-04-29 17:29:01
5d98 data A        2019-04-29 17:28:41
602a data B        2019-04-29 17:28:56
8c55 data C        2019-04-29 17:28:58
9cbe data D        2019-04-29 17:28:59
$ dmf find
id  type desc      modified
2517 data file1.txt 2019-04-29 17:29:00
344c data file2.txt 2019-04-29 17:29:01
5d98 data A        2019-04-29 17:28:41
602a data B        2019-04-29 17:28:56
8c55 data C        2019-04-29 17:28:58
9cbe data D        2019-04-29 17:28:59
```

The find-specific options add filters. In the example below, the find filters for files that were modified after the given date and time.

```
$ dmf find --modified 2019-04-29T17:29:00..
id  type desc      modified
2517 data file1.txt 2019-04-29 17:29:00
344c data file2.txt 2019-04-29 17:29:01
```

dmf info

Show detailed information about a resource. This command may also be referred to as `dmf show`.

dmf info options

identifier

Identifier, or unique prefix thereof, of the resource. Any unique prefix of the identifier will work, but if that prefix matches multiple identifiers, you need to add `--multiple` to allow multiple records in the output.

--multiple

Allow multiple records in the output (see *identifier*)

-f, --format value

Output format. Accepts the following values:

term Terminal output (colored, if the terminal supports it), with values that are empty left out and some values simplified for easy reading.

json Raw JSON value for the resource, with newlines and indents for readability.

jsonc Raw JSON value for the resource, “compact” version with no extra whitespace added.

dmf info usage

The default is to show, with some terminal colors, a summary of the resource:

```
$ dmf info 0b62

Resource 0b62d999f0c44b678980d6a5e4f5d37d
created
  '2019-03-23 17:49:35'
creator
  name: dang
datafiles
  - desc: foo13
    is_copy: true
    path: foo13
    sha1: feee44ad365b6b1ec75c5621a0ad067371102854
datafiles_dir
  /home/dang/src/idaes/dangunter/idaes-dev/ws2/files/
  ↪ 71d101327d224302aa8875802ed2af52
desc
  foo13
doc_id
  4
id_
  0b62d999f0c44b678980d6a5e4f5d37d
modified
  '2019-03-23 17:49:35'
relations
  - 1e41e6ae882b4622ba9043f4135f2143 --[derived]--> ME
type
  data
version
  0.0.0 @ 2019-03-23 17:49:35
```

The same resource in JSON format:

```
$ dmf info --format json 0b62
{
  "id_": "0b62d999f0c44b678980d6a5e4f5d37d",
  "type": "data",
  "aliases": [],
  "codes": [],
  "collaborators": [],
  "created": 1553363375.817961,
  "modified": 1553363375.817961,
  "creator": {
    "name": "dang"
  },
  "data": {},
  "datafiles": [
    {
      "desc": "foo13",
      "path": "foo13",
      "sha1": "feee44ad365b6b1ec75c5621a0ad067371102854",
      "is_copy": true
    }
  ],
}
```

(continues on next page)

(continued from previous page)

```

    "datafiles_dir": "/home/dang/src/idaes/dangunter/idaes-dev/ws2/files/
↪ 71d101327d224302aa8875802ed2af52",
    "desc": "foo13",
    "relations": [
      {
        "predicate": "derived",
        "identifier": "1e41e6ae882b4622ba9043f4135f2143",
        "role": "object"
      }
    ],
    "sources": [],
    "tags": [],
    "version_info": {
      "created": 1553363375.817961,
      "version": [
        0,
        0,
        0,
        ""
      ],
      "name": ""
    },
    "doc_id": 4
  }

```

And one more time, in “compact” JSON:

```

$ dmf info --format jsonc 0b62
{"id_": "0b62d999f0c44b678980d6a5e4f5d37d", "type": "data", "aliases": [], "codes": ↪
↪ [], "collaborators": [], "created": 1553363375.817961, "modified": 1553363375.
↪ 817961, "creator": {"name": "dang"}, "data": {}, "datafiles": [{"desc": "foo13",
↪ "path": "foo13", "sha1": "feee44ad365b6blec75c5621a0ad067371102854", "is_copy": ↪
↪ true}], "datafiles_dir": "/home/dang/src/idaes/dangunter/idaes-dev/ws2/files/
↪ 71d101327d224302aa8875802ed2af52", "desc": "foo13", "relations": [{"predicate":
↪ "derived", "identifier": "1e41e6ae882b4622ba9043f4135f2143", "role": "object"}],
↪ "sources": [], "tags": [], "version_info": {"created": 1553363375.817961, "version
↪ ": [0, 0, 0, ""], "name": ""}, "doc_id": 4}

```

dmf init

Initialize the current workspace. Optionally, create a new workspace.

dmf init options

path

Use the provided *path* as the workspace path. This is required.

--create

Create a new workspace at location provided by *path*. Use the *--name* and *--desc* options to set the workspace name and description, respectively. If these are not given, they will be prompted for interactively.

--name

Workspace name, used by *--create*

--desc

Workspace description, used by `--create`

dmf init usage

Note: In the following examples, the current working directory is set to `/home/myuser`.

This command sets a value in the user-global configuration file in `.dmf`, in the user's home directory, so that all other `dmf` commands know which workspace to use. With the `--create` option, a new empty workspace can be created.

Create new workspace in sub-directory `ws`, with given name and description:

```
$ dmf init ws --create --name "foo" --desc "foo workspace description"
Configuration in '/home/myuser/ws/config.yaml'
```

Create new workspace in sub-directory `ws`, providing the name and description interactively:

```
$ dmf init ws --create
New workspace name: foo
New workspace description: foo workspace description
Configuration in '/home/myuser/ws/config.yaml'
```

Switch to workspace `ws2`:

```
$ dmf init ws2
```

If you try to switch to a non-existent workspace, you will get an error message:

```
$ dmf init doesnotexist
Existing workspace not found at path='doesnotexist'
Add --create flag to create a workspace.
$ mkdir some_random_directory
$ dmf init some_random_directory
Workspace configuration not found at path='some_random_directory/'
```

If the workspace exists, you cannot create it:

```
$ dmf init ws --create --name "foo" --desc "foo workspace description"
Configuration in '/home/myuser/ws/config.yaml'
$ dmf init ws --create
Cannot create workspace: path 'ws' already exists
```

And, of course, you can't create workspaces anywhere you don't have permissions to create directories:

```
$ mkdir forbidden
$ chmod 000 forbidden
$ dmf init forbidden/ws --create
Cannot create workspace: path 'forbidden/ws' not accessible
```


dmf ls

This command lists resources in the current workspace.

dmf ls options

--color

Allow (if terminal supports it) colored terminal output. This is the default.

--no-color

Disallow, even if terminal supports it, colored terminal output.

-s, --show

Pick field to show in output table. This option can be repeated to show any known subset of fields. Also the option value can have commas in it to hold multiple fields. Default fields, if this option is not specified at all, are “type”, “desc”, and “modified”. The resource identifier field is always shown first.

codes List name of code(s) in resource. May be shortened with ellipses.

created Date created.

desc Description of resource.

files List names of file(s) in resource. May be shortened with ellipses.

modified Date modified.

type Name of the type of resource.

version Resource version.

You can specify other fields from the schema, as long as they are not arrays of objects, i.e. you can say `--show tags` or `--show version_info.version`, but `--show sources` is too complicated for a tabular listing. To see detailed values in a record use the *dmf info* command.

-S, --sort

Sort by given field; if repeated, combine to make a compound sort key. These fields are a subset of those in `-s, --show`, with the addition of `id` for sorting by the identifier: “id”, “type”, “desc”, “created”, “modified”, and/or “version”.

--no-prefix

By default, shown identifier is the shortest unique prefix, but if you don’t want the identifier shortened, this option will force showing it in full.

-r, --reverse

Reverse the order of the sorting given by (or implied by absence of) the `-S, --sort` option.

dmf ls usage

Note: In the following examples, the current working directory is set to `/home/myuser` and the workspace is named `ws`.

Without arguments, show the resources in an arbitrary (though consistent) order:

```
$ dmf ls
id  type desc  modified
0b62 data foo13 2019-03-23 17:49:35
1e41 data foo10 2019-03-23 17:47:53
6c9a data foo14 2019-03-23 17:51:59
d3d5 data bar1 2019-03-26 13:07:02
e780 data foo11 2019-03-23 17:48:11
eb60 data foo12 2019-03-23 17:49:08
```

Add a sort key to sort by, e.g. modified date

```
$ dmf ls -S modified
id  type desc  modified
1e41 data foo10 2019-03-23 17:47:53
e780 data foo11 2019-03-23 17:48:11
eb60 data foo12 2019-03-23 17:49:08
0b62 data foo13 2019-03-23 17:49:35
6c9a data foo14 2019-03-23 17:51:59
d3d5 data bar1 2019-03-26 13:07:02
```

Especially for resources of type “data”, showing the first (possibly only) file that is referred to by the resource is useful:

```
$ dmf ls -S modified -s type -s modified -s files
id  type modified      files
1e41 data 2019-03-23 17:47:53 foo10
e780 data 2019-03-23 17:48:11 foo11
eb60 data 2019-03-23 17:49:08 foo12
0b62 data 2019-03-23 17:49:35 foo13
6c9a data 2019-03-23 17:51:59 foo14
d3d5 data 2019-03-26 13:07:02 bar1
```

Note that you don’t actually have to show a field to sort by it (compare sort order with results from command above):

```
$ dmf ls -S modified -s type -s files
id  type files
1e41 data foo10
e780 data foo11
eb60 data foo12
0b62 data foo13
6c9a data foo14
d3d5 data bar1
```

Add `--no-prefix` to show the full identifier:

```
$ dmf ls -S modified -s type -s files --no-prefix
id                                type files
1e41e6ae882b4622ba9043f4135f2143 data foo10
e7809d25b390453487998e1f1ef0e937 data foo11
```

(continues on next page)

(continued from previous page)

```
eb606172dde74aa79eea027e7eb6a1b6 data foo12
0b62d999f0c44b678980d6a5e4f5d37d data foo13
6c9a85629cb24e9796a2d123e9b03601 data foo14
d3d5981106ce4d9d8cccd4e86c2cd184 data bar1
```

dmf register

Register a new resource with the DMF, using a file as an input. An alias for this command is `dmf add`.

dmf register options

--no-copy

Do not copy the file, instead remember path to current location. Default is to copy the file under the workspace directory.

-t, --type

Explicitly specify the type of resource. If this is not given, then try to infer the resource type from the file. The default will be 'data'. The full list of resource types is in `idaes.dmf.resource.RESOURCE_TYPES`

--strict

If inferring the type fails, report an error. With `--no-strict`, or no option, if inferring the type fails, fall back to importing as a generic file.

--no-unique

Allow duplicate files. The default is `--unique`, which will stop and print an error if another resource has a file matching this file's name and contents.

--contained resource

Add a 'contained in' relation to the given resource.

--derived resource

Add a 'derived from' relation to the given resource.

--used resource

Add a 'used by' relation to the given resource.

--prev resource

Add a 'version of previous' relation to the given resource.

--is-subject

If given, reverse the sense of any relation(s) added to the resource so that the newly created resource is the subject and the existing resource is the object. Otherwise, the new resource is the object of the relation.

--version

Set the semantic version of the resource. From 1 to 4 part semantic versions are allowed, e.g.

- 1
- 1.0
- 1.0.1

- 1.0.1-alpha

See <http://semver.org> and the function `idaes.dmf.resource.version_list()` for more details.

dmf register usage

Note: In the following examples, the current working directory is set to `/home/myuser` and the workspace is named `ws`.

Register a new file, which is a CSV data file, and use the `--info` option to show the created resource.

```
$ printf "index,time,value\n1,0.1,1.0\n2,0.2,1.3\n" > file.csv
$ dmf reg file.csv --info
Resource 117a42287aec4c5ca333e0ff3ac89639
created
  '2019-04-11 03:58:52'
creator
  name: dang
datafiles
  - desc: file.csv
    is_copy: true
    path: file.csv
    sha1: f1171a6442bd6ce22a718a0e6127866740c9b52c
datafiles_dir
  /home/myuser/ws/files/4db42d92baf3431ab31d4f91ab1a673b
desc
  file.csv
doc_id
  1
id_
  117a42287aec4c5ca333e0ff3ac89639
modified
  '2019-04-11 03:58:52'
type
  data
version
  0.0.0 @ 2019-04-11 03:58:52
```

If you try to register (add) the same file twice, it will be an error by default. You need to add the `--no-unique` option to allow it.

```
$ printf "index,time,value\n1,0.1,1.0\n2,0.2,1.3\n" > timeseries.csv
$ dmf add timeseries.csv
2315bea239c147e4bc6d2e1838e4101f
$ dmf add timeseries.csv
This file is already in 1 resource(s): 2315bea239c147e4bc6d2e1838e4101f
$ dmf add --no-unique timeseries.csv
3f95851e4931491b995726f410998491
```

If you register a file ending in “.json”, it will be parsed (unless it is over 1MB) and, if it passes, registered as type JSON. If the parse fails, it will be registered as a generic file *unless* the `--strict` option is given (with this option, failure to parse will be an error):

```
$ echo "totally bogus" > notreally.json
$ dmf reg notreally.json
```

(continues on next page)

(continued from previous page)

```

2019-04-12 06:06:47,003 [WARNING] idaes.dmf.resource: File ending in '.json' is not
↳valid JSON: treating as generic file
d22727c678a1499ab2c5224e2d83d9df
$ dmf reg --strict notreally.json
Failed to infer resource: File ending in '.json' is not valid JSON

```

You can explicitly specify the type of the resource with the `-t`, `--type` option. In that case, any failure to validate will be an error. For example, if you say the resource is a Jupyter Notebook file, and it is not, it will fail. But the same file with type “data” will be fine:

```

$ echo "Ceci n'est pas une notebook" > my.ipynb
$ dmf reg -t notebook my.ipynb
Failed to load resource: resource type 'notebook': not valid JSON
$ dmf reg -t data my.ipynb
0197a82abab44ecf980d6e42e299b258

```

You can add links to existing resources with the options `--contained`, `--derived`, `--used`, and `--prev`. For all of these, the new resource being registered is the target of the relation and the option argument is the identifier of an existing resource that is the subject of the relation.

For example, here we add a “shoebox” resource and then some “shoes” that are contained in it:

```

$ touch shoebox.txt shoes.txt closet.txt
$ dmf add shoebox.txt
755374b6503a47a09870dfbdc572e561
$ dmf add shoes.txt --contained 755374b6503a47a09870dfbdc572e561
dba0a5dc7d194040ac646bf18ab5eb50
$ dmf info 7553 # the "shoebox" contains the "shoes"
Resource 755374b6503a47a09870dfbdc572e561

created
  '2019-04-11 20:16:50'
creator
  name: dang
datafiles
  - desc: shoebox.txt
    is_copy: true
    path: shoebox.txt
    sha1: da39a3ee5e6b4b0d3255bfef95601890afd80709
datafiles_dir
  /home/dang/src/idaes/dangunter/idaes-dev/docs/ws/files/
↳7f3ff820676b41689bb32bc325fd2d1b
desc
  shoebox.txt
doc_id
  9
id_
  755374b6503a47a09870dfbdc572e561
modified
  '2019-04-11 20:16:50'
relations
  - dba0a5dc7d194040ac646bf18ab5eb50 <--[contains]-- ME
type
  data
version
  0.0.0 @ 2019-04-11 20:16:50

$ dmf info dba0 # the "shoes" are in the "shoebox"

```

(continues on next page)

(continued from previous page)

```

Resource dba0a5dc7d194040ac646bf18ab5eb50
created
  '2019-04-11 20:17:28'
creator
  name: dang
datafiles
  - desc: shoes.txt
    is_copy: true
    path: shoes.txt
    sha1: da39a3ee5e6b4b0d3255bfef95601890afd80709
datafiles_dir
  /home/dang/src/idaes/dangunter/idaes-dev/docs/ws/files/
→ a27f98c24d1848eabalb26e5ef87be88
desc
  shoes.txt
doc_id
  10
id_
  dba0a5dc7d194040ac646bf18ab5eb50
modified
  '2019-04-11 20:17:28'
relations
  - 755374b6503a47a09870dfbdc572e561 --[contains]--> ME
type
  data
version
  0.0.0 @ 2019-04-11 20:17:28

```

To reverse the sense of the relation, add the `--is-subject` flag. For example, we now add a “closet” resource that contains the existing “shoebox”. This means the shoebox now has two different “contains” type of relations.

```

$ dmf add closet.txt --is-subject --contained 755374b6503a47a09870dfbdc572e561
22ace0f8ed914fa3ac3e7582748924e4
$ dmf info 7553

```

```

Resource 755374b6503a47a09870dfbdc572e561
created
  '2019-04-11 20:16:50'
creator
  name: dang
datafiles
  - desc: shoebox.txt
    is_copy: true
    path: shoebox.txt
    sha1: da39a3ee5e6b4b0d3255bfef95601890afd80709
datafiles_dir
  /home/dang/src/idaes/dangunter/idaes-dev/docs/ws/files/
→ 7f3ff820676b41689bb32bc325fd2d1b
desc
  shoebox.txt
doc_id
  9
id_
  755374b6503a47a09870dfbdc572e561
modified
  '2019-04-11 20:16:50'
relations

```

(continues on next page)

(continued from previous page)

```

- dba0a5dc7d194040ac646bf18ab5eb50 <--[contains]-- ME
- 22ace0f8ed914fa3ac3e7582748924e4 --[contains]--> ME
type
  data
version
  0.0.0 @ 2019-04-11 20:16:50

```

You can give your new resource a version with the `--version` option. You can use this together with the `--prev` option to link between multiple versions of the same underlying data:

```

# note: following command stores the output of "dmf reg", which is the
#       id of the new resource, in the shell variable "oldid"
$ oldid=$( dmf reg oldfile.py --type code --version 0.0.1 )
$ dmf reg newfile.py --type code --version 0.0.2 --prev $oldid
ef2d801ca29a4a0a8c6f79ee71d3fe07
$ dmf ls --show type --show version --show codes --sort version
id   type version codes
44e7 code 0.0.1   oldfile.py
ef2d code 0.0.2   newfile.py
$ dmf related $oldid
44e7 code
    |
    |_version|_ ef2d code -

```

dmf related

This command shows resources related to a given resource.

dmf related options

-d, --direction

Direction of relationships to show / follow. The possible values are:

in Show incoming connection/relationship edges. Since all relations have a bi-directional counterpart, this effectively only shows the immediate neighbors of the root resource. For example, if the root resource is “A”, and “A” *contains* “B” and “B” *contains* “C”, then this option shows the incoming edge from “B” to “A” but not the edge from “C” to “B”.

out (Default) Show the outgoing connection/relationship edges. This will continue until there are no more connections to show, avoiding cycles. For example, if the root resource is “A”, and “A” *contains* “B” and “B” *contains* “C”, then this option shows the outgoing edge from “A” to “B” and also from “B” to “C”.

The default value is `out`.

--color

Allow (if terminal supports it) colored terminal output. This is the default.

--no-color

Disallow, even if terminal supports it, colored terminal output.

--unicode

Allow unicode drawing characters in the output. This is the default.

--no-unicode

Use only ASCII characters in the output.

dmf related usage

In the following examples, we work with 4 resources arranged as a fully connected square (A, B, C, D). This is not currently possible just with the command-line, but the following Python code does the job:

```
from idaes.dmf import DMF, resource
dmf = DMF()
rlist = [resource.Resource(value={"desc": ltr, "aliases": [ltr],
                                "tags": ["graph"]})
          for ltr in "ABCD"]
relation = resource.PR_USES
for r in rlist:
    for r2 in rlist:
        if r is r2:
            continue
        resource.create_relation_args(r, relation, r2)
for r in rlist:
    dmf.add(r)
```

If you save that script as *r4.py*, then the following command-line actions will run it and verify that everything is created.

```
$ python r4.py
$ dmf ls
id  type  desc  modified
1e7f other B    2019-04-20 15:43:49
3bc5 other D    2019-04-20 15:43:49
ba67 other A    2019-04-20 15:43:49
f7e9 other C    2019-04-20 15:43:49
```

You can then see the connections by looking at any one of the four resource (e.g., A):

```
$ dmf rel ba67
ba67 other A
  |
  |—uses| 3bc5 other D
  |
  |—uses| f7e9 other C
  |
  |—uses| 1e7f other B
  |
  |—uses| ba67 other A
  |
  |—uses| f7e9 other C
  |
  |—uses| 3bc5 other D
  |
  |—uses| 1e7f other B
  |
  |—uses| ba67 other A
  |
  |—uses| 1e7f other B
  |
```

(continues on next page)

(continued from previous page)

```

|—uses| 3bc5 other D
|—uses| f7e9 other C
|—uses| ba67 other A

```

If you change the direction of relations, you will get much the same result, but with the arrows reversed.

dmf rm

Remove one or more resources. This also removes relations (links) to other resources.

dmf rm options

identifier

The identifier, or identifier prefix, of the resource(s) to remove

--list, --no-list

With the `--list` option, which is the default, the resources to remove, or removed, will be listed as if by the `dmf ls` command. With `--no-list`, then do not produce this output.

-y, --yes

If given, do not confirm removal of the resource(s) with a prompt. This is useful for scripts that do not want to bother with input, or people with lots of confidence.

--multiple

If given, allow multiple resources to be selected by an identifier prefix. Otherwise, if the given identifier matches more than one resource, the program will print a message and stop.

dmf rm usage

Note: In the following examples, there are 5 text files named “file1.txt”, “file2.txt”, ..., “file5.txt”, in the workspace. The identifiers for these files may be different in each example.

Remove one resource, by its full identifier:

```

$ dmf ls --no-prefix
id                                     type desc      modified
096aa2491e234c4b941f32b537dd3017 data file5.txt 2019-04-16 02:51:30
821fc8f8e54e4c65b481f483be7f5a2d data file4.txt 2019-04-16 02:51:29
c20f3a6e338a40ee8a3a4972544adb74 data file1.txt 2019-04-16 02:51:25
c8f2b5cb80824e649008c414db5287f7 data file3.txt 2019-04-16 02:51:28
cd62e3bcb9a4459c9f2f5405ca442961 data file2.txt 2019-04-16 02:51:26
$ dmf rm c20f3a6e338a40ee8a3a4972544adb74
id                                     type desc      modified
c20f3a6e338a40ee8a3a4972544adb74 data file1.txt 2019-04-16 02:51:25
Remove this resource [y/N]? y
resource removed
[dmfcli-167 !?]idaes-dev$ dmf ls --no-prefix

```

(continues on next page)

(continued from previous page)

id	type	desc	modified
096aa2491e234c4b941f32b537dd3017	data	file5.txt	2019-04-16 02:51:30
821fc8f8e54e4c65b481f483be7f5a2d	data	file4.txt	2019-04-16 02:51:29
c8f2b5cb80824e649008c414db5287f7	data	file3.txt	2019-04-16 02:51:28
cd62e3bcb9a4459c9f2f5405ca442961	data	file2.txt	2019-04-16 02:51:26

Remove a single resource by its prefix:

```
$ dmf ls
id      type desc      modified
6dd5 data file2.txt 2019-04-16 18:51:10
7953 data file3.txt 2019-04-16 18:51:12
7a06 data file4.txt 2019-04-16 18:51:13
e5d7 data file1.txt 2019-04-16 18:51:08
fe0c data file5.txt 2019-04-16 18:51:15
$ dmf rm 6d
id      type desc      modified
6dd57ecc50a24efb824a66109dda0956 data file2.txt 2019-04-16 18:51:10
Remove this resource [y/N]? y
resource removed
$ dmf ls
id      type desc      modified
7953 data file3.txt 2019-04-16 18:51:12
7a06 data file4.txt 2019-04-16 18:51:13
e5d7 data file1.txt 2019-04-16 18:51:08
fe0c data file5.txt 2019-04-16 18:51:15
```

Remove multiple resources that share a common prefix. In this case, use the `-y`, `--yes` option to remove without prompting.

```
$ dmf ls
id      type desc      modified
7953 data file3.txt 2019-04-16 18:51:12
7a06 data file4.txt 2019-04-16 18:51:13
e5d7 data file1.txt 2019-04-16 18:51:08
fe0c data file5.txt 2019-04-16 18:51:15
$ dmf rm --multiple --yes 7
id      type desc      modified
7953e67db4a543419b9988c52c820b68 data file3.txt 2019-04-16 18:51:12
7a06435c39b54890a3d01a9eab114314 data file4.txt 2019-04-16 18:51:13
2 resources removed
$ dmf ls
id      type desc      modified
e5d7 data file1.txt 2019-04-16 18:51:08
fe0c data file5.txt 2019-04-16 18:51:15
```

dmf status

This command shows basic information about the current active workspace and, optionally, some additional details. It does not (yet) give any way to modify the workspace configuration. To do that, you need to edit the `config.yaml` file in the workspace root directory. See [Configuration](#).

dmf status options

--color

Allow (if terminal supports it) colored terminal output. This is the default.

--no-color

Disallow, even if terminal supports it, colored terminal output. UNIX output streams to pipes should be detected and have color disabled, but this option can force that behavior if detection is failing.

-s, --show info

Show one of the following types of information:

files Count and total size of files in workspace

htmldocs Configured paths to the HTML documentation (for “%dmf help” magic in the Jupyter Notebook)

logging Configuration for logging

all Show all items above

-a, --all

This option is just an alias for “--show all”.

dmf status usage

Note: In the following examples, the current working directory is set to `/home/myuser` and the workspace is named `ws`.

Also note that the output shown below is plain (black) text. This is due to our limited understanding of how to do colored text in our documentation tool (Sphinx). In a color-capable terminal, the output will be more colorful.

Show basic workspace status:

```
$ dmf status
settings:
  workspace: /home/myuser/ws
workspace:
  location: /home/myuser/ws
  name: myws
  description: my workspace
  created: 2019-04-09 12:46:40
  modified: 2019-04-09 12:46:40
```

Add the file information:

```
$ dmf status --show files
settings:
  workspace: /home/myuser/ws
workspace:
  location: /home/myuser/ws
  name: myws
  description: my workspace
  created: 2019-04-09 12:52:49
  modified: 2019-04-09 12:52:49
  files:
    count: 3
    total_size: 1.3 MB
```

You can repeat the `-s`, `--show` option to add more things:

```
$ dmf status --show files --show htmldocs
settings:
  workspace: /home/myuser/ws
workspace:
  location: /home/myuser/ws
  name: myws
  description: my workspace
  created: 2019-04-09 12:54:10
  modified: 2019-04-09 12:54:10
  files:
    count: 3
    total_size: 1.3 MB
  html_documentation_paths:
    -: /home/myuser/idaes/docs/build
```

However, showing everything is less typing, and not overwhelming:

```
$ dmf status -a
settings:
  workspace: /home/myuser/ws
workspace:
  location: /home/myuser/ws
  name: myws
  description: my workspace
  created: 2019-04-09 12:55:05
  modified: 2019-04-09 12:55:05
  files:
    count: 3
    total_size: 1.3 MB
  html_documentation_paths:
    -: /home/myuser/idaes/docs/build
  logging:
    not configured
```

DMF Application Programming Interface (API)

This page describes how to use the DMF when you create and save your models. For information on performing some DMF functions from the command-line, see *DMF Command-line Interface*. All the modules referenced here are in the `idaes.dmf` subpackage.

- *Initialization*
- *Adding data*
- *Adding relations*

Initialization

You can create a new `dmfbase.DMF` instance quite simply:

```
from idaes.dmf import DMF
dmf = DMF() # new DMF instance
```

When initialized this way, the DMF will use the *configuration* it finds in a file called *.dmf* in the user’s home directory. You can specify another configuration to use. The configuration of the DMF specifies where the workspace is located, which can be retrieved through the attribute *workspace_path*.

Adding data

The data in the DMF is broken down into “resources”. When adding data to the DMF with the Python API, you create resources and add them to the DMF instance. A resource describes one dataset, and contains:

- *metadata* about the creator, creation and modification time, version, names, and description
- *provenance* about the sources of the information
- *data*, as a references to files, embedded structured (JSON) data, or both
- *codes*, as references to code file locations or module paths, and optionally specific sections of that file or module
- *relations*, i.e. labeled connections to and from other resources. The following

To add a dataset, you first create a “resource”, which is an instance of `resource.Resource` (in module *resource*). This class provides some convenience methods for manipulating the underlying structure of the resource, which is contained in a Python dictionary (in an attribute called *v*, for “values”) and described, using JSON Schema syntax. The schema is contained in the module variable `resource.RESOURCE_SCHEMA`. An example of creating a new Resource object:

```
from idaes.dmf.resource import Resource

r = Resource()
r.v["version_info"]["version"] = test_version
r.v["collaborators"] = [
    {"name": "Clark Kent", "email": "ckent@dailyplanet.com"},
    {"name": "Superman", "email": "sman@fortress.solitude.org"},
]
r.v["sources"].append(
    {
        "isbn": "978-0201616224",
```

(continues on next page)

(continued from previous page)

```

        "source": 'Hunt, A. and Thomas, D., "The Pragmatic Programmer", '
        "Addison-Wesley, 1999",
        "date": "1999-01-01",
    }
)
r.v["codes"].append(
    {
        "type": "function",
        "name": "test_resource_full",
        "desc": "The test function",
        "location": "test_files.py",
        "version": test_version,
    }
)
r.v["datafiles"].append({"path": "/etc/passwd"})
r.v["aliases"] = ["test_resource_full"]
r.v["tags"] = ["test", "resource"]
r.data = {"arbitrary": {"values": [1, 2, 3]}}
return r

```

You can also create a resource that describes a file by using the `resource.Resource.from_file()` method. This will fill in the *datafiles* section of the resource object.

Once you have the resource object populated, you can add it to the DMF instance (and, thus, its workspace) with the `dmfbase.DMF.add()` method:

```

from idaes.dmf import DMF
from idaes.dmf.resource import Resource

r = Resource()
# ... create resource ...
dmf = DMF()
dmf.add(r)

```

You can create a resource and add it to the DMF in a single step with the `dmfbase.DMF.new()` method:

```

from idaes.dmf import DMF

dmf = DMF()
r = dmf.new(file="/path/to/breaking_news.doc",
            author={"name": "Clark Kent", "email": "ckent@dailyplanet.com"})

```

Once a resource is added to a DMF instance, you can still modify its content, but you need to call `dmfbase.DMF.update()` to synchronize those changes with the stored values. This is necessary for adding *relations* between two resources, which you simply cannot do until both of them are created. But it can also be used to do things like add a description:

```

from idaes.dmf import DMF

dmf = DMF()
# create and add resource
r = dmf.new(file="/path/to/breaking_news.doc")
# add a description to the resource
r.v["description"] = get_description()
# sync the description to the stored value
dmf.update()

```

Adding relations

One of the main functions of the DMF is to track the relationships, or relations, between its resources. In the lingo of graphs of objects, and in particular the Resource Description Framework (RDF) that is used as the foundation for many provenance systems, these relations are directed edges between objects, labeled by “predicates”. In this terminology, the resource from which the directed edge starts is called the “subject” of the relation, and the resource from which the directed edge ends is the “object”. The DMF defines the following predicates (associated module string constants are shown in parentheses):

- (resource.PR_DERIVED) derived: object is derived from subject
- (resource.PR_CONTAINS) contains: object contains the subject
- (resource.PR_USES) uses: object uses the subject
- (resource.PR_VERSION) version: object is a (new) version of the subject

Adding a relation between two resources is pretty straightforward. You create both resources and add them to the DMF, then create a “triple” to describe the connection between them (with the “predicate” that labels that connection), with the `resource.create_relation()` function. Then you call the `dmfbase.DMF.update()` function on the DMF instance to save the relation:

```
from idaes.dmf import DMF
from idaes.dmf.resource import Triple, PR_DERIVED
from idaes.dmf.resource import create_relation_args

dmf = DMF()
# create and add resources
r1 = dmf.new(file="/path/to/breaking_news.doc")
r2 = dmf.new(file="/path/to/interview_notes.txt")
# create relation (news --was derived from--> notes)
create_relation(r1, PR_DERIVED, r2)
# sync the relation to the DMF
dmf.update()
```

Flowsheet

Flowsheet models are the top level of the modeling heirachy. Flowsheet models represent traditional process flowsheets, containing a number of unit models connected together into a flow network and the property packages.

Property Package

Property packages are a collection of related models that represent the physical, thermodynamic, and reactive properties of the process streams.

Unit Model

Unit models represent individual pieces of equipment and their processes.

Data Management Framework

The *Data Management Framework* is used to manage all the data needed by the platform, including flowsheets, models, and results. It stores metadata and data in persistent storage.

4.2.4 Conventions

- *Units of Measurement and Reference States*
- *Standard Variable Names*
 - *Standard Naming Format*
 - *Constants*
 - *Thermophysical and Transport Properties*
 - *Reaction Properties*
 - *Solid Properties*
 - *Naming Examples*

Units of Measurement and Reference States

Due to the flexibility provided by the IDAES Integrated Platform, there is no standard set of units of measurement or standard reference state that should be used in models. Units of measurement are defined by the modeler for the 7 base quantities (time, length, mass, amount, temperature, current and luminous intensity) in each property package, and the platform makes use of this and Pyomo's Units container to automatically determine the units of all variables and expressions within a model. Thus, all components within a model using a given property package must use units based on the units chosen for the base quantities (to ensure consistency of units). However, flowsheets may contain property packages which use different sets of base units, however users should be careful to ensure units are converted correctly where property packages interact. For more detail on defining units of measurement see [Defining Units of Measurement](#).

Pyomo also provides convenient tools for converting between different units of measurement and checking for unit consistency, of which a few are highlighted below:

- `units.convert(var, to_units=units)` - returns a Pyomo expression including the variable *var* and the necessary conversion factors to convert it to the desired set of units (*units*). This method will return an *Exception* if the units of *var* are not consistent with those requested by the user.
- `units.assert_units_consistent(object)` - checks for consistency of units in *object* and raises an *AssertionError* if they are not. *object* may be a *Block*, *Constraint* or *Expression*.

The IDAES developers have generally used SI units without prefixes (i.e. Pa, not kPa) within models developed by the institute, with a default thermodynamic reference state of 298.15 K and 101325 Pa. Supercritical fluids have been considered to be part of the liquid phase, as they will be handled via pumps rather than compressors.

Standard Variable Names

In order for different models to communicate information effectively, it is necessary to have a standard naming convention for any variable that may need to be shared between different models. Within IDAES, this occurs most frequently when information regarding the state and properties of the material, which is calculated in specialized PropertyBlocks, is used in other parts of the model.

Standard Naming Format

There are a wide range of different variables which may be of interest to modelers, and a number of different ways in which these quantities can be expressed. In order to facilitate communication between different parts of models, a naming convention has been established to standardize the naming of variables across models. Variable names within IDAES follow to the format below:

```
{property_name}_{basis}_{state}_{condition}
```

Here, `property_name` is the name of the quantity in question, and should be drawn from the list of standard variable names given later in this document. If a particular quantity is not included in the list of standard names, users are encouraged to contact the IDAES developers so that it can be included in a future release. This is followed by a number of qualifiers which further indicate the specific conditions under which the quantity is being calculated. These qualifiers are described below, and some examples are given at the end of this document.

Basis Qualifier

Many properties of interest to modelers are most conveniently represented on an intensive basis, that is quantity per unit amount of material. There are a number of different bases that can be used when expressing intensive quantities, and a list of standard basis qualifiers are given below.

Basis	Standard Name
Mass Basis	mass
Molar Basis	mol
Volume Basis	vol

State Qualifier

Many quantities can be calculated either for the whole or a part of a mixture. In these cases, a qualifier is added to the quantity to indicate which part of the mixture the quantity applies to. In these cases, quantities may also be indexed by a Pyomo Set.

Basis	Standard Name	Comments
Component	comp	Indexed by component list
Phase	phase	Indexed by phase list
Phase & Component	phase_comp	Indexed by phase and component list
Total Mixture		No state qualifier

Phase	Standard Name
Supercritical Fluid	liq
Ionic Species	ion
Liquid Phase	liq
Solid Phase	sol
Vapor Phase	vap
Multiple Phases	e.g. liq1

Condition Qualifier

There are also cases where a modeler may want to calculate a quantity at some state other than the actual state of the system (e.g. at the critical point, or at equilibrium).

Basis	Standard Name
Critical Point	crit
Equilibrium State	equil
Ideal Gas	ideal
Reduced Properties	red
Reference State	ref

Constants

IDAES contains a library of common physical constants of use in process systems engineering models, which can be imported from `idaes.core.util.constants`. Below is a list of these constants with their standard names and values (SI units).

Note: It is important to note that these constants are represented as Pyomo *expressions* in order to include units of measurement. As such, they can be directly included in other *expressions* within a model. However, if the user desires to use their value directly (e.g. to initialize a variable), the `value()` method must be used to extract the value of the constant from the *expression*.

Constant	Standard Name	Value	Units
Acceleration due to Gravity	acceleration_gravity	9.80665	ms^{-2}
Avogadro's Number	avogadro_number	6.02214076e23	mol^{-1}
Boltzmann Constant	boltzmann_constant	1.38064900e-23	JK^{-1}
Elementary Charge	elementary_charge	1.602176634e-19	C
Faraday's Constant	faraday_constant	96485.33212	$Cmol^{-1}$
Gas Constant	gas_constant	8.314462618	$Jmol^{-1}K^{-1}$
Newtonian Constant of Gravitation	gravitational_constant	6.67430e-11	$m^3kg^{-1}s^{-2}$
Mass of an Electron	mass_electron	9.1093837015e-31	kg
Pi (Archimedes' Constant)	pi	3.141592 [1]	
Planck Constant	planck_constant	6.62607015e-34	Js
Stefan-Boltzmann Constant	stefan_constant	5.67037442e-8	$Wm^{-2}K^{-4}$
Speed of Light in a Vacuum	speed_light	299792458	ms^{-1}

[1] pi imported from the Python *math* library and is available to machine precision.

Values for fundamental constants and derived constants are drawn from the definitions of SI units (<https://www.bipm.org/utis/common/pdf/si-brochure/SI-Brochure-9.pdf>) and are generally defined to 9 significant figures.

Acceleration due to gravity, gravitational constant and electron mass are sourced from NIST (<https://physics.nist.gov>) and used the significant figures reported there.

Thermophysical and Transport Properties

Below is a list of all the thermophysical properties which have standardized names.

Variable	Standard Name
Activity	act
Activity Coefficient	act_coeff
Bubble Pressure	pressure_bubble
Bubble Temperature	temperature_bubble
Compressibility Factor	compress_fact
Concentration	conc
Density	dens
Dew Pressure	pressure_dew
Dew Temperature	temperature_dew
Diffusivity	diffus
Diffusion Coefficient (binary)	diffus_binary
Enthalpy	enth
Entropy	entr
Fugacity	fug
Fugacity Coefficient	fug_coeff
Gibbs Energy	energy_gibbs
Heat Capacity (const. P)	cp
Heat Capacity (const. V)	cv
Heat Capacity Ratio	heat_capacity_ratio
Helmholtz Energy	energy_helmholtz
Henry's Constant	henry
Internal Energy	energy_internal
Mass Fraction	mass_frac
Material Flow	flow
Molality	molality
Molecular Weight	mw
Mole Fraction	mole_frac
pH	pH
Pressure	pressure
Speed of Sound	speed_sound
Surface Tension	surf_tens
Temperature	temperature
Thermal Conductivity	therm_cond
Vapor Pressure	pressure_sat
Viscosity (dynamic)	visc_d
Viscosity (kinematic)	visc_k
Vapor Fraction	vap_frac
Volume Fraction	vol_frac

Reaction Properties

Below is a list of all the reaction properties which have standardized names.

Variable	Standard Name
Activation Energy	energy_activation
Arrhenius Coefficient	arrhenius
Heat of Reaction	dh_rxn
Entropy of Reaction	ds_rxn
Equilibrium Constant	k_eq
Reaction Rate	reaction_rate
Rate constant	k_rxn
Solubility Constant	k_sol

Solid Properties

Below is a list of all the properties of solid materials which have standardized names.

Variable	Standard Name
Min. Fluidization Velocity	velocity_mf
Min. Fluidization Voidage	voidage_mf
Particle Size	particle_dia
Pore Size	pore_dia
Porosity	particle_porosity
Specific Surface Area	area_{basis}
Sphericity	sphericity
Tortuosity	tort
Voidage	bulk_voidage

Naming Examples

Below are some examples of the IDAES naming convention in use.

Variable Name	Meaning
enth	Specific enthalpy of the entire mixture (across all phases)
flow_comp["H2O"]	Total flow of H2O (across all phases)
entr_phase["liq"]	Specific entropy of the liquid phase mixture
conc_phase_comp["liq", "H2O"]	Concentration of H2O in the liquid phase
temperature_red	Reduced temperature
pressure_crit	Critical pressure

4.2.5 Workflow

The section describes the recommended workflows for constructing and working with models on the IDAES Integrated Platform. Below is the list of the documented workflows.

General Workflow

While IDAES offers significant freedom in how users write their models, they are encouraged to follow this general workflow in order to make it easier for others to follow their code.

This workflow is used throughout the tutorials and examples on the [examples online documentation page](#).

Note: It is important to note that IDAES models are constructed upon execution of each line of code, and that most user defined options are only processed on model construction. This means that if the user wishes to make changes to any model construction option, it is necessary to rebuild the model from the beginning. Users should not be put off by this however, as model construction is generally very quick.

The general workflow for working with a model in IDAES is shown below:

- *1. Importing Modules*
- *2. Building a Model*
 - *2.1 Create a Model Object*
 - *2.2 Add a Flowsheet to the Model*
 - *2.3 Add Property Packages to Flowsheet*
 - *2.4 Add Unit Models to Flowsheet*
 - *2.5 Define Unit Model Connectivity*
 - *2.6 Expand Arcs*
 - *2.7 Add Variables, Constraints and Objectives*
- *3. Scaling the Model*
- *4. Specifying the Model*
- *5. Initializing the Model*
- *6. Solving the Model*
- *7. Optimizing the Model*
- *8. Analyzing and Visualizing the Results*

1. Importing Modules

IDAES is built upon a modular, object-oriented platform using Python, which requires users to import the components from the appropriate model libraries. The necessary components and libraries will vary from application to application, and were discussed earlier in this User Guide, however some common components users will need include:

- Pyomo environment components (e.g. `ConcreteModel`, `SolverFactory`, `TransformationFactory`, `Var`, `Constraint`, `objective`) imported from `pyomo.environ`
- Pyomo network components (e.g. `Arc`, `expand_arcs`) from `pyomo.network`
- IDAES FlowsheetBlock, from `idaes.core`
- *Property packages* for materials of interest
- Unit models for process equipment, drawn from either the IDAES model libraries and/or user-defined models
- Data visualization and analysis tools. Common tools include degrees of freedom and scaling, a full list is provided [here](#).
- External packages of interest to the user. Being built upon Python, users have access to the full range of Python libraries for working with and analyzing their models.

2. Building a Model

The next step in the workflow is to create a model object which represents the problem to be solved. The steps involved in this may vary depending on the problem being solved, but the general procedure is as follows:

2.1 Create a Model Object

The foundation of any model in IDAES is a Pyomo *ConcreteModel* object, which is created as follows:

```
m = ConcreteModel()
```

Note: IDAES does not support the use of Pyomo *AbstractModels*

2.2 Add a Flowsheet to the Model

The foundation of a process model within IDAES is the *FlowsheetBlock*, which forms the canvas upon which the process will be constructed. A key aspect of the *FlowsheetBlock* is to define whether the model will be steady-state or dynamic, and to define the time domain as appropriate.

```
m.fs = FlowsheetBlock(default={"dynamic": False})
```

Note: IDAES supports nested flowsheets to allow complex processes to be broken down into smaller sub-processes.

2.3 Add Property Packages to Flowsheet

All process models depend on calculations of thermophysical and chemical reaction properties, which are represented in IDAES using property packages. Users need to add the property packages they intend to use to the flowsheet.

```
m.fs.properties_1 = MyPropertyPackage.PhysicalParameterBlock()
```

Note: Users can add as many property packages as they need to a flowsheet, and can determine which property package will be used for each unit operation as it is created.

2.4 Add Unit Models to Flowsheet

Next, the user can add Unit Models to their flowsheet to represent each unit operation in the process.

```
m.fs.unit01 = UnitModel(default={"property_package": m.fs.properties_1})
```

2.5 Define Unit Model Connectivity

In order to describe the flow of material between unit operations, users must declare *Arcs* (or streams) which connect the outlet of each unit operation to the inlet of the next.

```
m.fs.arc_1 = Arc(source=m.fs.unit01.outlet, destination=m.fs.unit02.inlet)
```

2.6 Expand Arcs

It is important to note that *Arcs* only define the connectivity between unit operations, but do not create the actual model constraints needed to describe this. Once all *Arcs* in a flowsheet have been defined, it is necessary to expand these *Arcs* using the Pyomo *TransformationFactory*.

```
TransformationFactory("network.expand_arcs").apply_to(m)
```

Note: Pyomo provides a number of other Transformations and tools that may be useful to the user depending on the application. Examples include the *gdp* and *dae* transformations.

2.7 Add Variables, Constraints and Objectives

Finally, users can add any additional variables, constraints and objectives to their model. These could include the objective function for which they wish to optimize, additional constraints that provide limits on process performance, or simply additional quantities that the user wishes to use in analyzing or visualizing the results.

3. Scaling the Model

Note: The IDAES scaling tools are currently under development.

Ensuring that a model is well scaled is important for increasing the efficiency and reliability of solvers, and users should consider model scaling as an integral part of the modeling process. IDAES provides a number of tool for assisting users with scaling their models, and details on these can be found [here](#).

4. Specifying the Model

Note: IDAES is in the process of developing a set of tools to assist users with working with units of measurement when fixing and displaying values.

The next step is to specify the model by fixing variables. which can be done using the form *variable_name.fix(value)*. The variables that need to be fixed are application dependent, but commonly include the feed state variables.

In order to prepare the model for initialization, it is necessary to fully specify the model, such that there are no degrees of freedom. IDAES provides a tools for counting and reporting the degrees of freedom in any model (or sub-model/block):

```
from idaes.core.util.model_statistics import degrees_of_freedom
print(degrees_of_freedom(m) )
```

Note: Whilst it is not always necessary to fully define a model before initialization, it is much safer to do so as it ensures the model is well-defined. Most IDAES initialization tools check that the model is well-defined before proceeding, and will raise an Exception if it is not.

Note: Depending on the solver to be used during initialization, it can be better to avoid putting bounds on variables and adding inequality constraints at this stage. For solving square problems (i.e. zero degrees of freedom), some solvers (e.g. IPOPT) perform better without bounds on the problem. These bounds and constraints can be added later when it comes time to optimize the problem.

5. Initializing the Model

The next step is to initialize the model. All IDAES models have established initialization methods that can be called using *model.initialize()* which can be expected to take a model from its initial state to a feasible solution for a set of initial guesses (within the models expected operating range).

IDEAS workflows generally use a sequential-modular approach to initialize flowsheets, where unit models are initialized sequentially, passing the outlet state from one unit as the initial state for the next. An automated sequential-modular tool is available through Pyomo and demonstrated in the tutorials.

6. Solving the Model

Important: The sequential-modular approach initializes each unit model individually, thus it is important to do a final solve of the overall flowsheet/model in order to complete the initialization process. In most cases, this final solve should only take a few iterations, as the state of each unit model should be at or near the final solution already.

In order to solve the model, it is necessary to create a solve object and set any desired solver options (such as tolerances, iteration limits etc.).

```
solver = SolverFactory('solver_name')
solver.options = {'tol': 1d-6}

results = solver.solve(m)
```

Users should check the output from the solver to ensure a feasible solution was found using the following:

```
print(results.solver.termination_condition)
```

Different problems will require different solvers, and users will need to experiment to find those that work best for their problems. The default solver for most IDAES applications is IPOPT, which can be downloaded using the `idaes get-extensions` command line.

7. Optimizing the Model

Once an initial solution has been found, users can proceed to solving the optimization problem of interest. This procedure will vary by application but generally involves the following steps:

- 7.1) Unfix some degrees of freedom to provide the problem with decision variables, `variable_name.unfix()`.
- 7.2) Add bounds to variables and inequality constraints to constrain solution space, `variable_name.setlb(value)` and `var_name.setub(value)`
- 7.3) Call a solver and check the termination conditions, see step 6 Solving the Model.

Note: Users may wish/need to use different solvers for initialization and optimization. IDAES and Pyomo support the use of multiple solvers as part of the same workflow for solving different types of problems.

8. Analyzing and Visualizing the Results

One of the benefits of the IDAES Integrated Platform is that it operates in a fully featured programming language, which provides users a high degree of flexibility in analyzing their models. For example, users can automate the simulation of the model across multiple objectives or a range of parameters, store and save results from one or multiple solutions. Users also have access to a wide range of tools for manipulating, plotting and visualizing the results.

Data Reconciliation and Parameter Estimation

This workflow generally describes features of the IDAES framework that are useful for data reconciliation and parameter estimation. Many of these features can be used for any task where plant data is to be used in conjunction with a process model. It is assumed that the user is familiar with the IDAES modeling platform and Pyomo. See the [General Workflow](#) for more information on how to set up a model.

This provides general information about IDAES functionality laid out in terms of typical use cases. See [Tutorials and Examples](#), for specific complete examples of data reconciliation and parameter estimation examples. Relevant tutorials can be found in `tutorials/advanced/data_recon_and_parameter_estimation`.

Data Management

IDAES has functions to read and manage process data. Data management functions are contained in the `idaes.dmf.model_data` module.

Reading Data

A set of process data can be stored in two files, a data file and a metadata file. The data file is a CSV file where the first column contains a data point index, usually a timestamp. The first row contains a header with the measurement tag names. The rest of the file contains measurement data. The data file format is shown in the table below.

index tag (optional)	tag 1	tag 2	...
index 1	data(1,1)	data(1,2)	...
index 2	data(2,1)	data(2,2)	...
...

Metadata is provided for each tag in a separate CSV file. The metadata file has no header row, and aside from the tag name, any column may be blank. In the metadata csv file, the first column is the measurement tag name, the second column is a string that maps the tag to a specific model quantity, the third column is a description of the tag, and the fourth column is a for units of measure. Any columns past the fourth column are ignored and can be used to store any additional information.

tag 1	model reference string 1	description 1	units of measure 1
tag 2	model reference string 2	description 2	units of measure 1
...

The unit strings should be interpretable by `pint`, with the additional unit strings given in [Unit String Information](#). The model reference string is the a string to reference a model quantity. In the reference string the top-level model is always represented by `m`. For example, the reference string for a heater block outlet temperature could be `m.flowsheet.heater.control_volume.properties_out[:].temperature`. This reference will be indexed by time.

Reading data can be done with the `idaes.dmf.model_data.read_data()` function.

```
idaes.dmf.model_data.read_data(csv_file, csv_file_metadata, model=None, re-
                               name_mapper=None, unit_system=None, ambi-
                               ent_pressure=1.0, ambient_pressure_unit='atm')
```

Read CSV data into a Pandas DataFrame.

The data should be in a form where the first row contains column headings where each column is labeled with a data tag, and the first column contains data point labels or time stamps. The metadata should be in a csv file where the first column is the tag name, the second column is the model reference (which can be empty), the third column is the tag description, and the fourth column is the unit of measure string. Any additional

information can be added to columns after the fourth column and will be ignored. The units of measure should be something that is recognized by pint, or in the aliases defined in this file. Any tags not listed in the metadata will be dropped.

The function returns two items a `pandas.DataFrame` containing process data, and a dictionary with tag metadata. The metadata dictionary keys are tag name, and the values are dictionaries with the keys: “reference_string”, “description”, “units”, and “reference”.

Parameters

- **csv_file** (*str*) – Path of file to read
- **csv_file_metadata** (*str*) – Path of csv file to read column metadata from
- **model** (*pyomo.environ.ConcreteModel*) – Optional model to map tags to
- **rename_mapper** (*Callable*) – Optional function to rename tags
- **unit_system** (*str*) – Optional system of units to attempt convert to
- **ambient_pressure** (*float, numpy.array, pandas.series, str*) – Optional pressure to use to convert gauge pressure to absolute. If a string is supplied, the corresponding data tag is assumed to be ambient pressure.
- **ambient_pressure_unit** (*str*) – Optional ambient pressure unit, should be a unit recognized by pint.

Returns (`pandas.DataFrame`, `dict`)

Binning Data

Process data can be divided into bins based on some criteria. This allows for estimating measurement uncertainty when no better information is available, and provides a way to look at how different process measurements vary for operating conditions that should be similar in some way. As an example, power plant data could be binned by power output, and assuming that operating procedures are standard, it could be assumed that measurements in each bin should be about the same. The variance of a measurement in a bin could be used as an approximation of uncertainty. Binning the data by load and time could show how process measurements change over time and be useful for things like fault detection and equipment degradation.

Adding bin information to a data frame is done with the `idaes.dmf.model_data.bin_data()` function.

```
idaes.dmf.model_data.bin_data(df, bin_by, bin_no, bin_nom, bin_size, min_value=None,
                               max_value=None)
```

Sort data into bins by a column value. If the min or max are given and the value in `bin_by` for a row is out of the range [min, max], the row is dropped from the data frame.

Parameters

- **df** (*pandas.DataFrame*) – Data frame to add bin information to
- **bin_by** (*str*) – A column for values to bin by
- **bin_no** (*str*) – A new column for bin number
- **bin_nom** (*str*) – A new column for the mid-point value of `bin_by`
- **bin_size** (*float*) – size of a bin
- **min_value** (*in {float, None}*) – Smallest value to keep or None for no lower
- **max_value** (*in {float, None}*) – Largest value to keep or None for no upper

Returns

returns the data frame, and a dictionary with the number of rows in each bin.

Return type (dict)

The `idaes.dmf.model_data.bin_stdev()` function can be used to calculate the standard deviation for each measurement in each bin.

```
idaes.dmf.model_data.bin_stdev(df, bin_no, min_data=4)
```

Calculate the standard deviation for each column in each bin.

Parameters

- **df** (*pandas.DataFrame*) – pandas data frame that is a bin number column
- **bin_no** (*str*) – Column to group by, usually contains bin number
- **min_data** (*int*) – Minimum number of data points required to calculate standard deviation for a bin (default=4)

Returns

key is the bin number and the value is a `pandas.Series` with column standard deviations

Return type dict

The function `idaes.dmf.model_data.data_plot_book()` creates a multipage PDF containing box plots for all the measurements based on the bins.

```
idaes.dmf.model_data.data_plot_book(df, bin_nom, file='data_plot_book.pdf',
                                     tmp_dir='tmp_plots', xlabel=None, metadata=None,
                                     cols=None, skip_cols=[])
```

Make box and whisker plots from process data based on bins from the `bin_data()` function.

Parameters

- **df** – data frame
- **bin_nom** – bin mid-point value column
- **file** – path for generated pdf
- **tmp_dir** – a directory to store temporary plots in
- **xlabel** – Label for x axis
- **metadata** – tag meta data dictionary

Returns None

To compare reconciled data to original measurements the `idaes.dmf.data_rec_plot_book()` is used. For each bin there are two box plots one for the original data and one for reconciled data.

```
idaes.dmf.model_data.data_rec_plot_book(df_data, df_rec, bin_nom,
                                         file='data_rec_plot_book.pdf',
                                         tmp_dir='tmp_plots', xlabel=None, meta-
                                         data=None, cols=None, skip_cols=[])
```

Make box and whisker plots from process data compared to data rec results based on bins from the `bin_data()` function. The `df_data` and `df_rec` data frames should have the same index set and the `df_data` data frame contains the bin data. This will plot the intersection of columns containing numerical data.

Parameters

- **df_data** – data frame with original data
- **df_rec** – data frame with reconciled data
- **bin_nom** – bin mid-point value column

- **file** – path for generated pdf
- **tmp_dir** – a directory to store temporary plots in
- **xlabel** – Label for x axis
- **metadata** – tag meta data dictionary
- **cols** – List of columns to plot, if None plot all
- **skip_cols** – List of columns not to plot, this overrides cols

Returns None

Tagging the Model

Mapping process data to a model is typically done by creating model tag dictionaries. Where the dictionary key is a measurement tag and the value is a reference to a model variable, expression, or parameter. The tags may be process measurement tags, or any other convenient string. IDAES has some utilities to help facilitate tagging of models.

If model reference strings were provided in the tag metadata file, the tag metadata from the `idaes.dmf.model_data.read_data()` will contain model references. These references can be accessed in the metadata dictionary as `metadata[tag]["reference"]`.

The reference strings are optional in the tag metadata file and can be added after reading the data. To add a reference string to the metadata, you can update the metadata dictionary like `metadata[tag]["reference_string"] = reference_string`. If you update the reference string, the `idaes.dmf.model_data.update_metadata_model_references()` function can be used to update the references in the tag metadata.

`idaes.dmf.model_data.update_metadata_model_references(model, metadata)`

Create model references from reference strings in the metadata. This updates the 'reference' field in the metadata.

Parameters

- **model** (*pyomo.environ.Block*) – Pyomo model
- **metadata** (*dict*) – Tag metadata dictionary

Returns None

An easy way to create a new tag dictionary from tag metadata is to use dictionary comprehension like so: `data_tags = {k:v["reference"][0] for k, v in metadata.items() if v["reference"] is not None}`

Often data reconciliation is performed using process data as the first step of parameter estimation or model validation. Data reconciliation can often fill in information for many unmeasured quantities. Most of this data can be associated with process streams. To make managing proliferation of data tags easier, it is often desirable to create a new set of tags based on stream (Arc) names that can be automatically obtained from the model.

The first step to creating a new set of tags based on streams is to get a dictionary of streams and their associated state blocks, with can be done with the `idaes.core.util.tables.arcs_to_stream_dict()` function.

`idaes.core.util.tables.arcs_to_stream_dict(blk, additional=None, descend_into=True, sort=False, prepend=None, s={})`

Creates a stream dictionary from the Arcs in a model, using the Arc names as keys. This can be used to automate the creation of the streams dictionary needed for the `create_stream_table_dataframe()` and `stream_states_dict()` functions.

Parameters

- **blk** (*pyomo.environ._BlockData*) – Pyomo model to search for Arcs

- **additional** (*dict*) – Additional states to add to the stream dictionary, which aren't represented by arcs in blk, for example feed or product streams without Arcs attached or states internal to a unit model.
- **descend_into** (*bool*) – If True, search subblocks for Arcs as well. The default is True.
- **sort** (*bool*) – If True sort keys and return an OrderedDict
- **prepend** (*str*) – Prepend a string to the arc name joined with a '.'. This can be useful to prevent conflicting names when sub blocks contain Arcs that have the same names when used in combination with descend_into=False.
- **s** (*dict*) – Add streams to an existing stream dict.

Returns Dictionary with Arc names as keys and the Arcs as values.

The stream dictionary can be converted to a corresponding dictionary of state at a specific time with the `idaes.core.util.tables.stream_states_dict()` function.

`idaes.core.util.tables.stream_states_dict(streams, time_point=0)`

Method to create a dictionary of state block representing stream states. This takes a dict with stream name keys and stream values.

Parameters

- **streams** – dict with name keys and stream values. Names will be used as display names for stream table, and streams may be Arcs, Ports or StateBlocks.
- **time_point** – point in the time domain at which to generate stream table (default = 0)

Returns A pandas DataFrame containing the stream table data.

With the dictionary of states, a tag dictionary can be created automatically with the `idaes.core.util.tables.stream_states_dict()` function.

`idaes.core.util.tables.stream_states_dict(streams, time_point=0)`

Method to create a dictionary of state block representing stream states. This takes a dict with stream name keys and stream values.

Parameters

- **streams** – dict with name keys and stream values. Names will be used as display names for stream table, and streams may be Arcs, Ports or StateBlocks.
- **time_point** – point in the time domain at which to generate stream table (default = 0)

Returns A pandas DataFrame containing the stream table data.

Objective Function

For either parameter estimation or data reconciliation the objective function is often written in the form:

$$\min \sum_i \frac{(x_{\text{data},i} - x_{\text{model},i})^2}{\sigma_i^2}$$

To add the data to be used in the objective standard practice has been to add a mutable parameter for data and standard deviation indexed by measurement tags. The parameter values can be set from the measurement data frame to a specific index.

The following code snippet exemplifies the use of data parameters in a model.

```

# df is from reading process data into a pandas.DataFrame. bin_stdev comes
# from binning the data and calculating the standard deviations, as described
# in the Data Management section.

# Add data parameters
m.data = pyo.Param(data_tags, mutable=True)
m.data_stdev = pyo.Param(data_tags, mutable=True)

# A function to set the data parameters from measurement data
def set_data(m, df, data_tags, index=None):
    m.bin_no = df.iloc[index]["bin_no"]
    for t in data_tags:
        m.data[t] = df.iloc[index][t]
        m.data_stdev[t] = bin_stdev[m.bin_no][t]

# Expressions for error in the objective function
@m.Expression(data_tags)
def err(m, i):
    return (m.data[i] - data_tags[i])/m.data_stdev[i]

```

Parameter Estimation

For parameter estimation and data reconciliation, it is recommended to use [Paramest](#). If more control is needed a user can also set up their own parameter estimation problem, by combining multiple process models into a larger parameter estimation model.

APPENDIX: Unit String Information

This section provides additional detail about units strings that can be used to read data with the `read_data()` function.

Temperature Differences

The unit conversion for temperatures with offsets (C and F) are deferent depending on whether a measurement is temperature or temperature difference. It is important to ensure the temperature units are correctly specified before reading data. The units “delta_degC and delta_degF” are defined to handle temperature differences.

Units Not Converted

The following units are not affected by unit conversion.

- percent
- ppm
- ppb
- pH
- VAR
- MVAR
- H2O

- percent open
- percent closed

Gauge Pressure

The table below shows a list of unit string that are taken to be gauge pressure when data is read. Gauge pressures are converted to absolute pressure in the unit conversion process.

Gauge Pressure Unit	Absolute Pressure Unit
psig	psi
in water gauge	in water
in hg gauge	in hg

Additional Unit Definitions

Some units are common enough in process data that the `read_data()` function will recognize them and convert them to standard Pint unit strings. Unit strings are case sensitive to handle things like milli (m) and mega (M) prefixes.

The table below shows the additional units.

Unit String	Pint Unit String
Pressure	
PSI	psi
PSIA	psi
psia	psi
PSIG	psig
INWC	in water
IN WC	in water
IN/WC	in water
" H2O	in water
INHG	in hg
IN HG	in hg
IN/HG	in hg
HGA	in hg
IN HGA	in hg
Fraction	
PCT	percent
pct	percent
PERCT	percent
PERCT.	percent
PCNT	percent
PPM	ppm
PPB	ppb
% OPEN	percent open
% CLSD	percent closed
% CLOSED	percent closed
Length	
IN	in
INS	in

continues on next page

Table 2 – continued from previous page

Unit String	Pint Unit String
INCHES	in
Inches	in
FT	ft
FEET	ft
FOOT	ft
Feet	ft
MILS	minch
Speed	
MPH	mile/hr
IPS	in/s
Volume	
KGAL	kgal
Vol Flow	
GPM	gal/min
gpm	gal/min
CFM	ft ³ /min
KCFM	ft ³ /mmin
SCFM	ft ³ /min
KSCFM	ft ³ /mmin
Angle	
DEG	deg
Angular Speed	
RPM	rpm
Frequency	
HZ	hz
Temperature	
DEG F	degF
Deg F	degF
deg F	degF
DEG C	degC
Deg C	degC
deg C	degC
DEGF	degF
DegF	degF
DEGC	degC
DegC	degC
Temperature Difference	
DELTA DEG F	delta_degF
DETLA Deg F	delta_degF
DETLA deg F	delta_degF
DETLA DEG C	delta_degC
DETLA Deg C	delta_degC
DELTA deg C	delta_degC
DELTA DEGF	delta_degF
DELTA DegF	delta_degF
DELTA degF	delta_degF
DELTA DEGC	delta_degC
DELTA DegC	delta_degC
DELTA degC	delta_degC

continues on next page

Table 2 – continued from previous page

Unit String	Pint Unit String
Delta DEG F	delta_degF
Delta Deg F	delta_degF
Delta deg F	delta_degF
Delta DEG C	delta_degC
Delta Deg C	delta_degC
Delta deg C	delta_degC
Delta DEGF	delta_degF
Delta DegF	delta_degF
Delta degF	delta_degF
Delta DEGC	delta_degC
Delta DegC	delta_degC
Delta degC	delta_degC
delta DEG F	delta_degF
delta Deg F	delta_degF
delta deg F	delta_degF
delta DEG C	delta_degC
delta Deg C	delta_degC
delta deg C	delta_degC
delta DEGF	delta_degF
delta DegF	delta_degF
delta degF	delta_degF
delta DEGC	delta_degC
delta DegC	delta_degC
delta degC	delta_degC
Energy	
MBTU	kbtu
Mass	
MLB	klb
K LB	klb
K LBS	klb
lb.	lb
Mass flow	
TPH	ton/hr
tph	ton/hr
KLB/HR	klb/hr
KPPH	klb/hr
Current	
AMP	amp
AMPS	amp
Amps	amp
Amp	amp
AMP AC	amp
pH	
PH	pH
VARs (volt-amp reactive)	
VARs	VAR
MVARs	MVAR

4.2.6 Command-line interface

The IDAES PSE Toolkit includes a command-line tool that can be invoked by typing *idaes* in a UNIX or Mac OSX shell, or Windows Powershell, that is in an installed IDAES environment. For the most part, this means that wherever you installed IDAES will have this command available.

This section of the documentation describes the capabilities of this command-line program.

idaes command

The base *idaes* command does not do anything by itself, besides set some shared configuration values. All the real work is done by one of the subcommands, each of which is described on a separate page below.

idaes bin-directory: Show IDAES executable file directory

This page lists the options for the *idaes* “bin-directory” bin-directory. This is invoked like:

```
idaes [general options] bin-directory [bin-directory options]
```

general options

The following general options from the *idaes* base command affect the bin-directory bin-directory. They should be placed *before* the “bin-directory” bin-directory, on the command-line.

- `-v/--verbose`
- `-q/--quiet`

See the [idaes command](#) for details.

idaes bin-directory

This subcommand shows the IDAES executable file directory.

options

--help

Show the help message and exit.

--exists

Show if the directory exists.

--create

Create the directory.

idaes copyright: Show IDAES copyright information

This page lists the options for the `idaes “copyright”` subcommand. This is invoked like:

```
idaes [general options] copyright [subcommand options]
```

general options

The following general options from the *idaes* base command affect the copyright subcommand. They should be placed *before* the “copyright” subcommand, on the command-line.

- `-v/--verbose`
- `-q/--quiet`

See the *idaes command* for details.

idaes copyright

This subcommand prints the IDAES copyright notice to standard output.

options

`--help`

Show the help message and exit.

idaes data-directory: Show IDAES data directory

This page lists the options for the `idaes “data-directory”` subcommand. This is invoked like:

```
idaes [general options] data-directory [subcommand options]
```

general options

The following general options from the *idaes* base command affect the data-directory subcommand. They should be placed *before* the “data-directory” subcommand, on the command-line.

- `-v/--verbose`
- `-q/--quiet`

See the *idaes command* for details.

idaes data-directory

This subcommand shows the IDAES data directory.

options

--help

Show the help message and exit.

--exists

Show if the directory exists.

--create

Create the directory.

idaes get-examples: Fetch example scripts and Jupyter Notebooks

This page lists the options for the `idaes “get-examples”` subcommand. This is invoked like:

```
idaes [general options] get-examples [subcommand options]
```

general options

The following general options from the *idaes* base command affect the `get-examples` subcommand. They should be placed *before* the “get-examples” subcommand, on the command-line.

- `-v/--verbose`
- `-q/--quiet`

See the *idaes command* for details.

idaes get-examples

This subcommand fetches example scripts and Jupyter Notebooks from a given release in [Github](#). and puts them in a directory of the users’ choosing. If the user does not specify a directory, the default is *examples*.

options

--help

Show the help message and exit.

-d, --dir TEXT

Select the installation target directory. See *example usage* for several examples of this option.

-I, --no-install

Do *not* install examples into ‘idaes_examples’ package. Examples are installed by default so they can be imported directly from Python. Not installing them might cause some tests, which import the examples, to fail.

-l, --list-releases

List all available released versions, and stop. This lets people browse the releases and select one. By default, the release that matches the version of the currently installed “idaes” package is used. See also the *–unstable* option.

–N, --no-download

Do not download anything. If the *–no-install* option is also given, this means the command will essentially do nothing. Or, looked at another way, this option means that only action will be the installation of the “idaes_examples” package from the selected directory.

–U, --unstable

Allow and list unstable/pre-release versions. This applies to both download and the *–list-releases* option. Unstable releases are marked with “rcN” or similar suffixes.

–V, --version TEXT

Version of examples to download. The default version, which is shown for the *–help* option, is the same as the version of the IDAES PSE toolkit for which the *idaes* command is installed. If the version to install is unstable (ends with “rcN”) then you will need to add the *–unstable* option to avoid errors.

example usage

idaes get-examples Download examples from release matching release for the *idaes* command, install them in the *examples* subdirectory of this directory, and install the modules found under *examples/src* as a package named *idaes_examples*. The *examples* directory must not exist, i.e. the program will refuse to overwrite the contents of an existing directory.

idaes get-examples -d /tmp/examples Same as above, but put downloaded code in */tmp/examples* instead.

idaes get-examples -d /tmp/examples -I Download to */tmp/examples*, but do not install.

idaes get-examples -d /tmp/examples -N Install the examples found under */tmp/examples*.

idaes get-examples –version 1.4.2-pre Download examples from release *1.4.2-pre*, install them in the *examples* subdirectory of this directory, and install the modules found under *examples/src* as a package named *idaes_examples*.

idaes get-examples –list-releases List available releases of the examples in Github repository, *idaes/examples-pse*. Do not attempt to download or install anything.

idaes get-examples –list-releases –unstable Same as above, but include non-stable releases.

idaes get-extensions: Get solvers and libraries

This page lists the options for the *idaes* “get-extensions” subcommand. This is invoked like:

```
idaes [general options] get-extensions [subcommand options]
```

general options

The following general options from the *idaes* base command affect the get-extensions subcommand. They should be placed *before* the “get-extensions” subcommand, on the command-line.

- -v/--verbose
- -q/--quiet

See *idaes command* for details.

idaes get-extensions

This subcommand gets the compiled solvers and libraries from a remote repository, and installs them locally.

options

--help

Show the help message and exit.

--url

URL from which to download the solvers/libraries.

idaes lib-directory: Show IDAES library file directory

This page lists the options for the *idaes* “lib-directory” subcommand. This is invoked like:

```
idaes [general options] lib-directory [subcommand options]
```

general options

The following general options from the *idaes* base command affect the lib-directory subcommand. They should be placed *before* the “lib-directory” subcommand, on the command-line.

- -v/--verbose
- -q/--quiet

See the *idaes command* for details.

idaes lib-directory

This subcommand shows the IDAES library file directory.

options

--help

Show the help message and exit.

--exists

Show if the directory exists.

--create

Create the directory.

IDAES Versioning

The IDAES Python package is versioned according to the general guidelines of [semantic versioning](#), following the recommendations of [PEP 440](#) with respect to extended versioning descriptors (alpha, beta, release candidate, etc.).

Basic usage

You can see the version of the package at any time interactively by printing out the `__version__` variable in the top-level package:

```
import idaes
print(idaes.__version__)
# prints a version like "1.2.3"
```

Advanced usage

This section describes the module's variables and classes.

Overview

The API in this module is mostly for internal use, e.g. from 'setup.py' to get the version of the package. But `Version` has been written to be usable as a general versioning interface.

Example of using the class directly:

```
>>> from idaes.ver import Version
>>> my_version = Version(1, 2, 3)
>>> print(my_version)
1.2.3
>>> tuple(my_version)
(1, 2, 3)
>>> my_version = Version(1, 2, 3, 'alpha')
>>> print(my_version)
1.2.3.a
>>> tuple(my_version)
(1, 2, 3, 'alpha')
>>> my_version = Version(1, 2, 3, 'candidate', 1)
>>> print(my_version)
1.2.3.rc1
>>> tuple(my_version)
(1, 2, 3, 'candidate', 1)
```


If you want to add a version to a class, e.g. a model, then simply inherit from `HasVersion` and initialize it with the same arguments you would give the `Version` constructor:

```
>>> from idaes.ver import HasVersion
>>> class MyClass(HasVersion):
...     def __init__(self):
...         super(MyClass, self).__init__(1, 2, 3, 'alpha')
...
>>> obj = MyClass()
>>> print(obj.version)
1.2.3.a
```

```
idaes.ver.__version__ = '1.9.0'
```

Package's version as a simple string

```
idaes.ver.package_version = <idaes.ver.Version object>
```

Package's version as an object

Version class

The versioning semantics are encapsulated in a class called *Version*.

```
class idaes.ver.Version(major, minor, micro, releaselevel='final', serial=None, label=None)
```

This class attempts to be compliant with a subset of [PEP 440](#).

Note: If you actually happen to read the PEP, you will notice that pre- and post- releases, as well as “release epochs”, are not supported.

```
__init__(major, minor, micro, releaselevel='final', serial=None, label=None)
```

Create new version object.

Provided arguments are stored in public class attributes by the same name.

Parameters

- **major** (*int*) – Major version
- **minor** (*int*) – Minor version
- **micro** (*int*) – Micro (aka patchlevel) version
- **releaselevel** (*str*) – Optional PEP 440 specifier
- **serial** (*int*) – Optional number associated with releaselevel
- **label** (*str*) – Optional local version label

```
__iter__()
```

Return version information as a sequence.

```
__str__()
```

Return version information as a string.

HasVersion class

For adding versions to other classes in a simple and standard way, you can use the *HasVersion* mixin class.

```
class idaes.ver.HasVersion(*args)
```

Interface for a versioned class.

```
    __init__(*args)
```

Constructor creates a *version* attribute that is an instance of *Version* initialized with the provided args.

Parameters **args* – Arguments to be passed to *Version* constructor.

shared configuration

--help

See a list of subcommands and options, or get help for a specific subcommand.

-v

--verbose

Increase verbosity. Show warnings if given once, then info, and then debugging messages.

-q

--quiet

Increase quietness. If given once, only show critical messages. If given twice, show no messages.

4.2.7 IDAES Flowsheet Visualizer

Contents

- *IDAES Flowsheet Visualizer*
 - *Introduction*
 - *Guide*
 - *Reference*

Introduction

The IDAES Flowsheet Visualizer, or IFV for short, is a web-based user interface (UI) that lets you:

- View any IDAES flowsheet as a process engineering diagram
- Export flowsheet diagrams as images ([SVG](#) format)
- View and export the “stream table” for the flowsheet
- Rearrange the flowsheet diagram to your taste and save the arrangement for next time
- Dynamically refresh the displayed values to reflect changes in the underlying IDAES model

To use the IFV, first *install IDAES*. The IFV can be invoked from a Jupyter Notebook or a Python script. It does not require that you run any other application. Currently the IFV is only for viewing the flowsheet on your own computer.

¹

Starting and stopping the IFV is fast and does not consume many resources.

¹ *But, since it is a web application, a shared service for viewing flowsheets stored remotely is definitely possible.*

Guide

This guide describes how to invoke (i.e., start) and use the IFV.

Invocation

The IFV visualizes *flowsheets*. To get started with creating a flowsheet with IDAES, see the *Flowsheet models* documentation page. Once you have created your flowsheet, simply call the *visualize* method on that object, passing some parameters to give it a name and optional file for saving changes:

```
# First, create your IDAES model, "m", which has an attribute ".fs" for the flowsheet
# Then, invoke the `visualize` method
m.fs.visualize("My Flowsheet", save_as="my_flowsheet.json")
```

The invocation of the *visualize* method will pop up a browser tab or window with the UI, displaying the flowsheet and, if the information is available, the stream table. In the notebook or script, you can continue to run more code and the UI will continue to work in the background. You can close the UI at any time. If you exit the script or notebook while the UI is running, you can still manipulate the diagram and stream table, but you will not be able to save or refresh, since these require communication with the Python process that no longer exists.

There are three ways to invoke the *visualize* functionality, which in the end do the same thing and have the same arguments.

1. Use the *visualize* method of a flowsheet (as above)
2. Call the *visualize* function from the package *idaes.ui.fsvis*, passing it a flowsheet object
3. Call the same *visualize* function from the module *idaes.ui.fsvis.fsvis*, passing it a flowsheet object

In all cases, the arguments and behavior are the same. See the *visualize function documentation* for details on parameters to this function.

Note: You can continue to modify and use your model and flowsheet after calling *visualize()*. This may update the visualization, but nothing you do in the IFV will affect the Python model; it is *read-only*.

User Interface

This section describes how to use the graphical web user interface. We start with a screenshot of the UI, with the main areas highlighted. Then we zoom in on each area and describe how to use it.

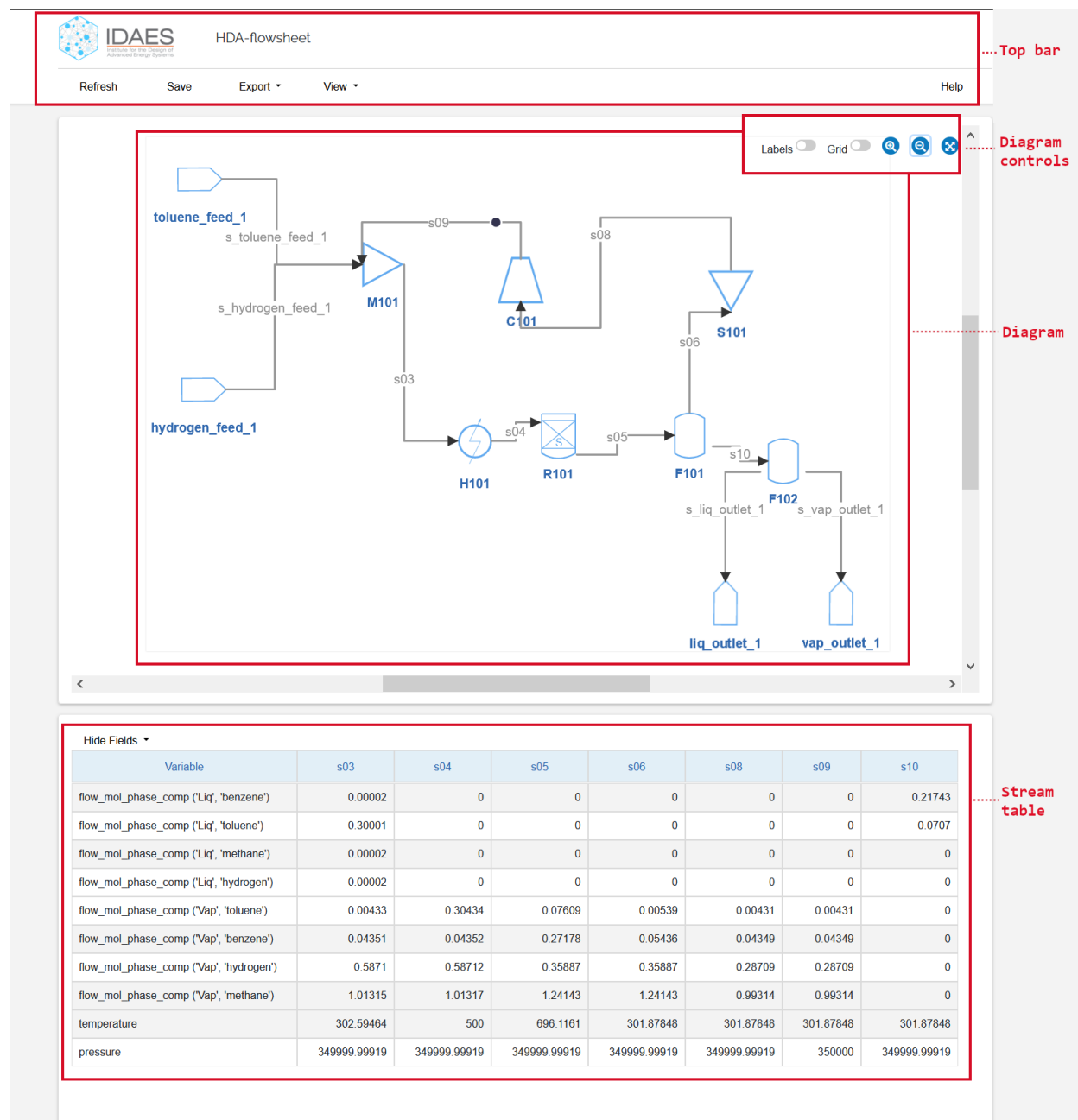


Fig. 1: Screenshot of the main window of the IFV UI

Top bar

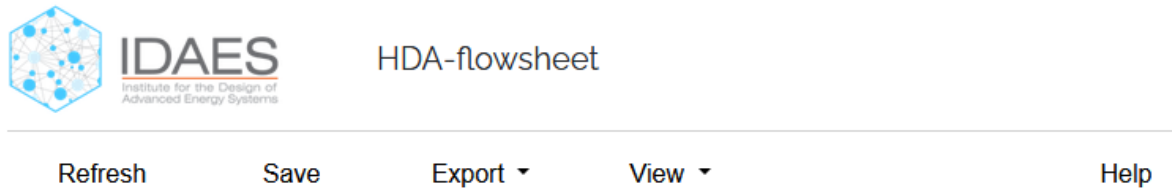


Fig. 2: Screenshot of the top bar of the IFV UI

The *top bar* has a title bar, which contains the IDAES logo and the name of the flowsheet being visualized, and a menu. The menu items are:

- **Refresh:** Update the view with any changes made to the flowsheet from the Python side. This also has the effect of saving the current layout.
- **Save:** Save the current layout to the data store that was specified with the visualization was launched. Note that this does *not* update with any changes made to the flowsheet in Python (use *Refresh* for that). Neither does it have any effect on the Python flowsheet values, as the IFV cannot modify the underlying flowsheet.
- **Export:** Save the flowsheet or stream table as a file.
 - **Flowsheet:** Save the flowsheet as a Scalable Vector Graphics (SVG) file, a common format for images that consist of “vector” elements like boxes, lines, and text. SVG files can be viewed like images by most programs that allow image viewing, and even edited with a program like [Inkscape](#). You will get a preview of the image and a “Download” button that will save in a file named for the flowsheet, with a “.svg” extension.
 - **Stream table:** Save the flowsheet as comma-separated values. The result will be a text file, called “export.csv”, that contains the data.
- **View:** Toggle the visibility of the flowsheet (diagram) area and the stream table area.
- **Help:** Load this documentation page.

[Back to main window screenshot](#)

Diagram

The *diagram* (or *flowsheet*) area lets you rearrange the flowsheet as you need and zoom in on particular sections. You can interact with the components on the diagram:

Shapes Geometric shapes on the flowsheet represent unit models, inlets and outlets, and other IDAES components. They are connected by lines, and each has a name. All shapes can be moved by clicking and dragging them. If you right-click on a shape, it will rotate 90 degrees.

Lines The lines connecting units can be manipulated by clicking and dragging. You can click on a line to create a new segment that can be used for routing the line around objects. You can eliminate a segment by clicking on the dot that appears as you hover over the line. There are also pill-shaped handles that appear on the lines for moving them. The endpoints of the lines are determined by the flowsheet and cannot be changed. For the same reason, you also cannot add or remove lines.

Labels Both the shapes and lines have associated values that can be shown, which pop up over the lines if you toggle the “Show labels” control. See the [Diagram Controls](#) section for details.

More details on mouse and keyboard actions for the diagram are available in the [documentation of the underlying Rappid toolkit](#).

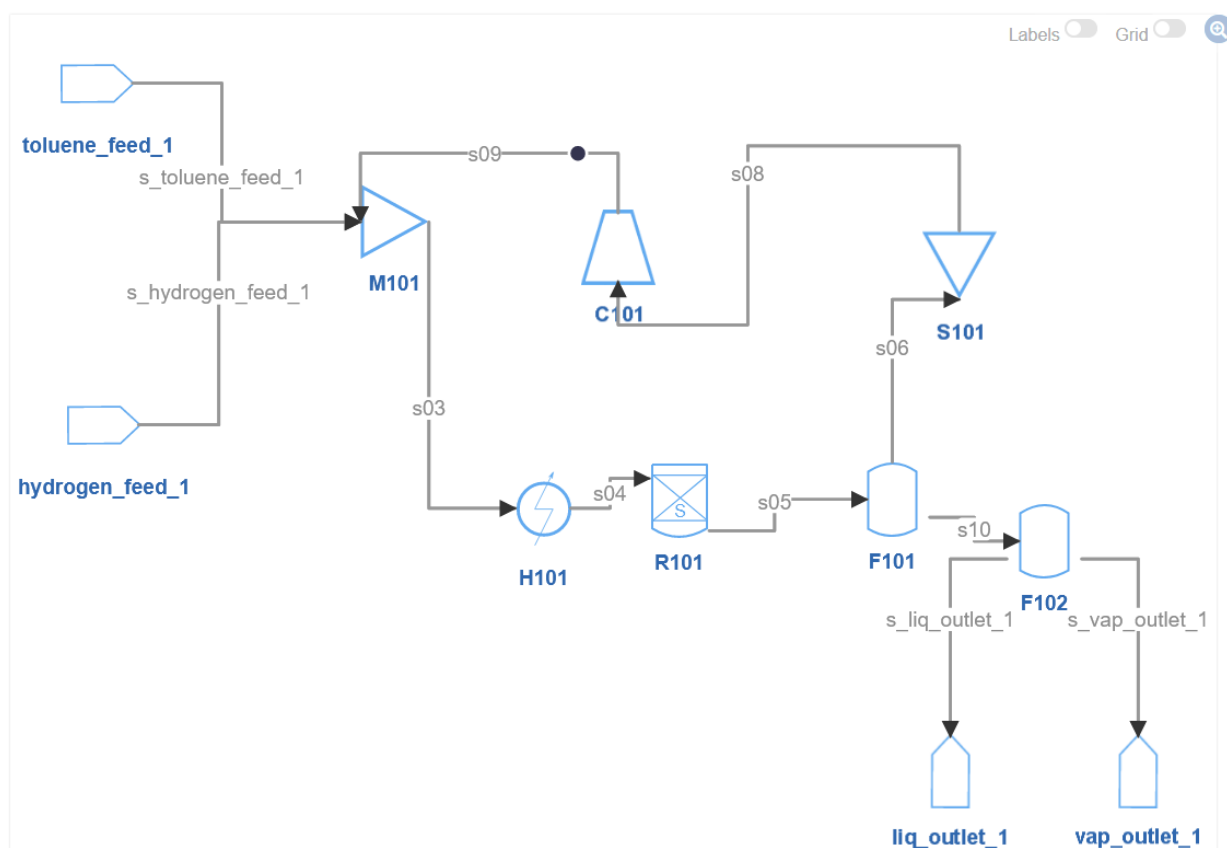


Fig. 3: Screenshot of the main *diagram* (or flowsheet) area of the IFV UI

[Back to main window screenshot](#)

Diagram Controls

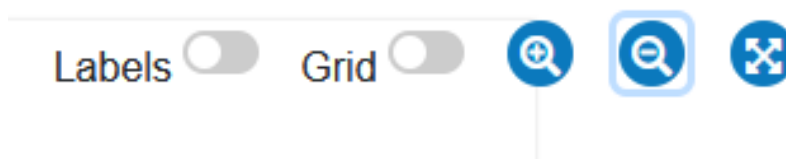





Fig. 4: Screenshot of the *diagram controls* area of the IFV UI

The *diagram controls* allow you to affect some global properties of the diagram/flowsheet area.

View actions

- Labels: Toggle visibility of the information (*labels*) shown for each stream. This is the same information that appears in the [Stream Table](#).
- Grid: Toggle a background “grid”
- : Zoom in by 25%
- : Zoom out by 25%
- : Fit the diagram into the current area

[Back to main window screenshot](#)

Stream Table

The IFV will show a stream table with variables defined for each stream in the flowsheet, if these values exist and the flowsheet adheres to the IDAES conventions for naming the inlet and outlet streams. An example of a stream table is shown below.

Hide Fields ▾

Variable	s03	s04	s05	s06	s08	s09	s10
flow_mol_phase_comp ('Liq', 'benzene')	0.00002	0	0	0	0	0	0.21743
flow_mol_phase_comp ('Liq', 'toluene')	0.30001	0	0	0	0	0	0.0707
flow_mol_phase_comp ('Liq', 'methane')	0.00002	0	0	0	0	0	0
flow_mol_phase_comp ('Liq', 'hydrogen')	0.00002	0	0	0	0	0	0
flow_mol_phase_comp ('Vap', 'toluene')	0.00433	0.30434	0.07609	0.00539	0.00431	0.00431	0
flow_mol_phase_comp ('Vap', 'benzene')	0.04351	0.04352	0.27178	0.05436	0.04349	0.04349	0
flow_mol_phase_comp ('Vap', 'hydrogen')	0.5871	0.58712	0.35887	0.35887	0.28709	0.28709	0
flow_mol_phase_comp ('Vap', 'methane')	1.01315	1.01317	1.24143	1.24143	0.99314	0.99314	0
temperature	302.59464	500	696.1161	301.87848	301.87848	301.87848	301.87848
pressure	349999.99919	349999.99919	349999.99919	349999.99919	349999.99919	350000	349999.99919

Fig. 5: Screenshot of an example stream table

There are a number of ways of manipulating this table:

- The “Hide Fields” pull-down menu provides a list of stream names. Select a name to hide/show that column in the table.
- Click on the column header and drag it left or right to change its order in the table.
- Resize a column by hovering over a column border until you see the mouse pointer change, then drag it to resize.

You can also export the entire table as a file of comma-separated values. See the [Export](#) documentation for details.

[Back to main window screenshot](#)

Reference

`idaes.ui.fsvis.fsvis.visualize` (*flowsheet*, *name*: *str* = 'flowsheet', *save_as*=None, *browser*: *bool* = True, *port*: *Optional[int]* = None, *log_level*: *int* = 30, *quiet*: *bool* = False) → *int*

Visualize the flowsheet in a web application.

The web application is started in a separate thread and this function returns immediately.

Also open a browser window to display the visualization app. The URL is also printed (unless `quiet` is True).

Note: The visualization server runs in its own thread. If the program that it is running in stops, the visualization UI will not be able to save or refresh its view. This is not an issue in a [REPL](#) like the Python console, IPython, or Jupyter Notebook, since these all run until the user explicitly closes them. But if you are running from a script, you need to do something to avoid having the program exit after the `visualize()` method returns (which happens very quickly). For example, loop forever in a try/catch clause that will handle `KeyboardInterrupt` exceptions:

```
# Example code for a script, to keep program running after starting visualize()
↳thread
my_model.fs.visualize() # this returns immediately
try:
    print("Type ^C to stop the program")
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    print("Program stopped")
```

Parameters

- **flowsheet** – IDAES flowsheet to visualize
- **name** – Name of flowsheet to display as the title of the visualization
- **save_as** – If a string or path then save to a file.
- **browser** – If true, open a browser
- **port** – Start listening on this port. If not given, find an open port.
- **log_level** – An IDAES logging level, which is a superset of the built-in `logging` module levels. See `idaes.logger` for details
- **quiet** – If True, suppress printing any messages to standard output (console)

Returns Port number where server is listening

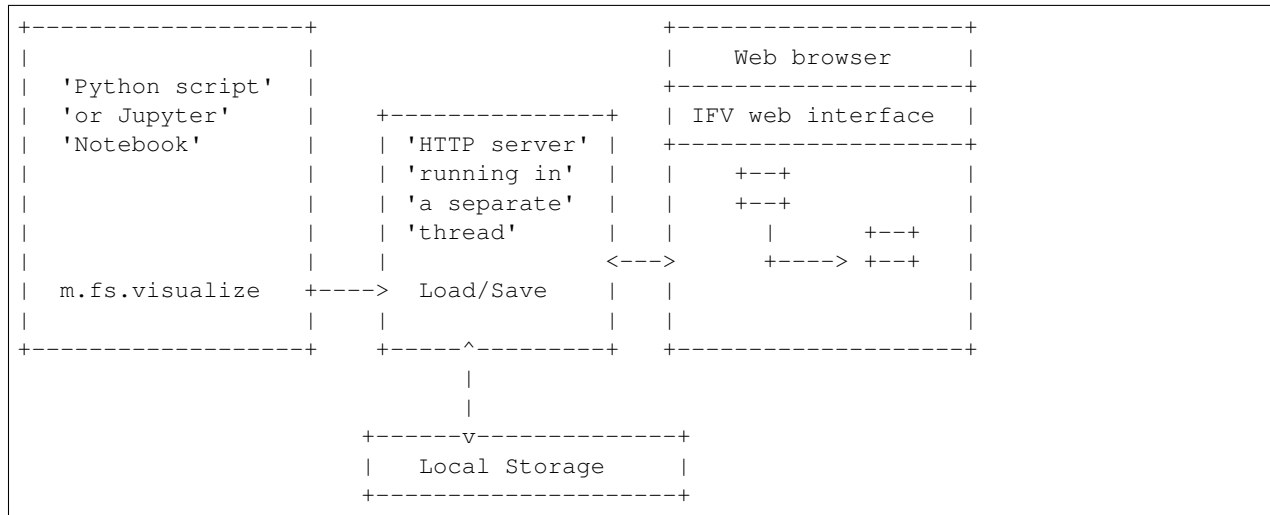
Raises `ValueError` – if the data storage at ‘save_as’ can’t be opened

Software notes

This section provides some additional details for developers or users more interested in the programming details.

Client/server architecture

The `visualize()` command works by starting an HTTP server in a separate thread, and serving requests from the UI (or any other requester). The server only responds to requests from your computer, not the internet. When you exit the script or Jupyter Notebook that called *visualize* then you will also stop the server – and the associated IFV page will no longer be able to save or refresh the flowsheet. The architecture diagram is shown below.



Persistence architecture

The saving of the model uses the module `idaes.ui.fsvis.persist`. This module implements the well-known “”, which makes it easy to extend by adding a new `DataStore` sub-class and updating the logic in the factory method, `create()`, to create and return instances of that class for a given input type. The input in this case comes from the `save_as` argument to the `visualize()` method.

4.2.8 IDAES Model Libraries

The documentation for the models are found in the technical specifications linked below.

- [Generic IDAES Model Library](#)
- [Power Generation Model Library](#)
- [Gas Solid Contactors Model Library](#)

4.2.9 Configuration

Some behavior of IDAES is configurable through the IDAES global ConfigBlock. IDAES's configuration is obtained by first setting everything to internal defaults; then loading a global config file, if it exists; then loading a config file from the current working directory, if it exists. After the `idaes` module is imported, the `idaes` ConfigBlock can be accessed at `idaes.cfg`. Some configuration options can be changed after importing `idaes` by calling `idaes.reconfig()`.

Configuration files are in **JSON format**. The default configuration is shown below and can be used as a template to create new configuration files. This is the configuration used by IDAES if nothing else is provided.

```
{
  "use_idaes_solvers":true,
  "logger_capture_solver":true,
  "logger_tags":[
    "framework",
    "model",
    "flowsheet",
    "unit",
    "control_volume",
    "properties",
    "reactions"
  ],
  "valid_logger_tags":[
    "framework",
    "model",
    "flowsheet",
    "unit",
    "control_volume",
    "properties",
    "reactions",
    "ui"
  ],
  "ipopt-idaes":{
    "options":{
      "nlp_scaling_method":"gradient-based"
    }
  },
  "logging":{
    "version":1,
    "disable_existing_loggers":false,
    "formatters":{
      "default_format":{
        "format": "%(asctime)s [%(levelname)s] %(name)s: %(message)s",
        "datefmt": "%Y-%m-%d %H:%M:%S"
      }
    },
    "handlers":{
      "console":{
        "class": "logging.StreamHandler",
        "formatter": "default_format",
        "stream": "ext://sys.stdout"
      }
    },
    "loggers":{
      "idaes":{
        "level": "INFO",
        "propagate": true,

```

(continues on next page)

(continued from previous page)

```
        "handlers": ["console"]
    },
    "idaes.solve":{
        "propagate": false,
        "level": "INFO",
        "handlers": ["console"]
    },
    "idaes.init":{
        "propagate": false,
        "level": "INFO",
        "handlers": ["console"]
    },
    "idaes.model":{
        "propagate":false,
        "level": "INFO",
        "handlers": ["console"]
    }
}
}
```

Global Configuration Files

IDAES configuration files are named `idaes.conf`. The easiest way to find where the global configuration file should be placed is to run the command `idaes data-directory`. A global configuration file won't exist unless a user creates one. The default configuration above can be used as a start.

Windows

On Windows the global configuration file is located at `%LOCALAPPDATA%\idaes\idaes.conf`.

UNIX-Like

On Unix-like systems the global configuration files is located at `$HOME/.idaes/idaes.conf`.

Other

On systems that have neither an `%LOCALAPPDATA%` or `$HOME` environment variable, global config files are not currently supported.

Local Configuration Files

Local configuration files are also named `idaes.conf` and can be placed in the working directory, which is the directory you launch Python from. You can also use the Python command `chdir()` to change the working directory before importing `idaes`.

In addition to reading local configuration files when `idaes` is imported, you can read a configuration file anytime by calling `idaes.read_config(path)`. Reading a configuration file will automatically apply any resulting configuration changes.

Changing the Configuration in a Script or Interactive Session

The `idaes` configuration can be changed anytime after the `idaes` module is imported. The standard `ConfigBlock` options are described in detail below. For example to change whether you want to use the solvers provided by `idaes` or ones you have installed elsewhere, you would first use the command `idaes.cfg["use_idaes_solvers"] = False` then to make the change take effect use `idaes.reconfig()`. Not all option changes require `idaes.reconfig()`, so whether they do or don't is provided in the options descriptions below.

Important Configuration Entries

The `ConfigBlock` has several options, but they are not all important to end-users. This section lists the commonly used entries.

logging

This section of the file configures IDAES loggers. Once the configuration is read, Python's standard `logging.config.dictConfig()` is used to set the logger configuration. See Python's logging documentation for more information.

IDAES has four main loggers defined in the standard configuration, although additional loggers can be added if desired. The standard loggers are:

1. `idaes`, this is the root logger of most IDAES logging, unless otherwise noted.
2. `idaes.init`, this is the root of IDAES initialization loggers.
3. `idaes.solve`, this is the root of IDAES solver loggers and solver information.
4. `idaes.model`, this is the root of model loggers. Model loggers are usually used by models written using the IDAES framework, but not part of the `idaes` package.

If changes to the logger configuration dictionary are made after importing `idaes` a call to `idaes.reconfig()` is required for it to take effect.

use_idaes_solvers

This option can be set to `False` (`false` in JSON) to direct the IDAES framework not to use solvers obtained with the `idaes get-extensions` command before using the solvers that may have been otherwise installed by the user. This can be used if a user would prefer to use solver versions they have installed apart from IDAES.

Changes require `idaes.reconfig()`. The default setting is `True`.

logger_capture_solver

If a solver call is done from inside a solver logging context, this setting will send the solver output to the logger if `True`, and not capture the solver output for the logger if `False`. If solver output is not captured it will be sent to the screen, and not be logged.

Changes do not require `idaes.reconfig()`. The default setting is `True`.

logger_tags

Loggers created with the `idaes.logging` module can be assigned tags. Output from these loggers is recorded if the loggers tag is in the `logger_tags` set. The default behavior can be configured in a configuration file. The tag set can also be modified at any time via functions in the `idaes.logging` module. This is a subset of `valid_log_tags`.

Changes do not require `idaes.reconfig()`. The default setting is: `["framework", "model", "flowsheet", "unit", "control_volume", "properties", "reactions"]`.

valid_log_tags

When setting logger tags for `idaes.logging` loggers they are compared against a list of valid tags. This is done to guard against spelling errors. If the default set of defined tags is not sufficient tags can be added.

Changes do not require `idaes.reconfig()`. The default setting is: `["framework", "model", "flowsheet", "unit", "control_volume", "properties", "reactions", "ui"]`.

ipopt-idaes

This is a config block that provides default configuration for the `ipopt-idaes`. `ipopt-idaes` is a wrapper for the Pyomo `ipopt` solver class that allows the default solver options to be configured in the general `idaes` `ConfigBlock`. Currently only solver options can be configured in the `options` sub-`ConfigBlock`.

For example to set the default NLP scaling method for `ipopt` to use `idaes`-provided scaling factors, use the command `idaes.cfg["ipopt-idaes"]["options"]["nlp_scaling_method"] = "user-scaling"`

Any `ipopt` solver options that can be passed via command line argument to the `ipopt` AMPL executable solver can be set under `idaes.cfg["ipopt-idaes"]["options"]` or equivalently in a configuration file.

Changes do not require `idaes.reconfig()`. The default options are: `{"nlp_scaling_method": "gradient-based"}`.

4.2.10 Logging

IDAES provides some logging extensions to provide finer control over information logging and to allow solver output to be logged. Logging can be a useful tool for debugging.

Getting Loggers

There are four main roots of IDAES loggers (`idaes`, `idaes.model`, `idaes.init`, `idaes.solve`). All of these loggers are standard Python loggers, and can be used as such. The main differences between using the IDAES logging functions to get the loggers and plain Python methods are that the IDAES functions make it a little easier to get loggers that fit into IDAES's standard logging hierarchy, and the IDAES loggers have a few additional named logging levels, which allow for finer control over the information displayed. Logging levels are described in detail later.

A tag can also be specified and used to filter logging records. By default the tag is `None` and log records won't be filtered. Valid tags are in the set `{None, "framework", "model", "flowsheet", "unit", "control_volume", "properties", "reactions"}`. Users may add to the set of valid names. To see how to control which logging tags are logged, see section “Tags” below. To avoid filtering out import warning and error messages, records logged at the `WARNING` level and above are not filtered out regardless of tag.

idaes Logger

Loggers descending from `idaes` (other than `idaes.init`, `idaes.model`, or `idaes.solve`) are used for general IDAES framework logging. Typically the module name `__name__` is used for the logger name. Modules in the `idaes` package already start with `idaes`, but if an IDAES logger is requested for a module outside of the `idaes` package `idaes.` is prepended to the name.

`idaes.logger.getLogger(name, level=None, tag=None)`

Return an `idaes` logger.

Parameters

- **name** – usually `__name__`
- **level** – standard IDAES logging level (default use IDAES config)
- **tag** – logger tag for filtering, see `valid_log_tags()`

Returns logger

Example

```
import idaes.logger as idaeslog

_log = idaeslog.getLogger(__name__, tag="framework")
```

idaes.init Loggers

The `init` logger will always descend from “`idaes.init`”. This logger is used in IDAES model initialization methods, and can be used in user models as well. Initialization methods are usually attached to a Pyomo Block. Blocks have a `name` attribute. So the logger name is usually given as the block name, and the `getInitLogger()` function prepends `idaes.init..` The advantage of using the block name over the module name is that users can see exactly which model instance the initialization log messages are coming from.

`idaes.logger.getInitLogger(name, level=None, tag=None)`

Get a model initialization logger

Parameters

- **name** – Object name (usually Pyomo Component name)
- **level** – Log level
- **tag** – logger tag for filtering, see `valid_log_tags()`

Returns logger

Example

```
import idaes.logger as idaeslog

class DummyBlock(object):
    """A dummy block for demonstration purposes"""
    def __init__(name):
        self.name = name

    def initialize(outlvl=idaeslog.INFO):
        init_log = idaeslog.getInitLogger(self.name, level=outlvl, tag="unit")
```

idaes.model Loggers

The model logger is used to provide a standard way to produce log messages from user models that are not part of the `idaes` package. The logger name has `idaes.model` prepended to the name provided by the user. This is convenient because it provides a way to use a standard configuration system for user model loggers. The user can choose any name they like for these loggers.

`idaes.logger.getLogger(name, level=None, tag=None)`

Get a logger for an IDAES model. This function helps users keep their loggers in a standard location and use the IDAES logging config.

Parameters

- **name** – Name (usually `__name__`). Any starting ‘`idaes.`’ is stripped off, so if a model is part of the `idaes` package, ‘`idaes`’ won’t be repeated.
- **level** – Standard Python logging level (default use IDAES config)
- **tag** – logger tag for filtering, see `valid_log_tags()`

Returns logger

Example

```
import idaes.logger as idaeslog

_log = idaeslog.getLogger("my_model", level=idaeslog.DEBUG, tag="model")
```

idaes.solve Loggers

The solve logger will always descend from “`idaes.solve`”. This logger is used to log solver output. Since solvers may produce a lot of output, it can be useful to specify different handlers for the solve logger to direct it to a separate file.

`idaes.logger.getSolveLogger(name, level=None, tag=None)`

Get a solver logger

Parameters

- **name** – logger name is “`idaes.solve.`” + name (if name starts with “`idaes.`” it is removed before creating the logger name)
- **level** – Log level
- **tag** – logger tag for filtering, see `valid_log_tags()`

Returns logger

Tags

Logger tags are provided to allow control over what types of log records to display. The logger tag is just a string that gets attached to a logger, which specifies that a logger generates records of a certain type. You can then specify what tags you want to see information from. A filter removes any tags that are not in the list of tags to display at levels below WARNING.

The set of tags to display information from is a global setting in the `idaes.logger` module. When getting a logger, you can set its tag by providing the `tag` argument, see “Getting Loggers” above.

The following functions can be used to specify which logging tags to display:

`idaes.logger.log_tags()`

Returns a set of logging tags to be logged.

Returns (set) tags to be logged

`idaes.logger.set_log_tags(tags)`

Specify a set of tags to be logged

Parameters `tags` (*iterable of str*) – Tags to log

Returns None

`idaes.logger.add_log_tag(tag)`

Add a tag to the list of tags to log.

Parameters `tag` (*str*) – Tag to log

Returns None

`idaes.logger.remove_log_tag(tag)`

Remove a tag from the list of tags to log.

Parameters `tag` (*str*) – Tag to no longer log

Returns None

The tags are validated against a list of valid tags to provide error checking for typos and to enforce some standard tag names. To provide more flexibility, users can add to the list of valid tag names, but cannot remove names.

`idaes.logger.valid_log_tags()`

Returns a set of valid logging tag names.

Returns (set) valid tag names

`idaes.logger.add_valid_log_tag(tag)`

Add a tag name to the list of valid names.

Parameters `tag` (*str*) – A tag name

Returns None

Levels

Several logging level constants are defined in the `idaes.logger` module. These include the standard Python Levels. The following levels are provided for IDAES loggers. The additional levels of info provide finer control over the amount of logging information produced by IDAES loggers.

Constant Name	Value	Name	Log Method
CRITICAL	50	CRITICAL	<code>critical()</code>
ERROR	40	ERROR	<code>error(), exception()</code>
WARNING	30	WARNING	<code>warning()</code>
INFO_LOW	21	INFO	<code>info_low()</code>
INFO	20	INFO	<code>info()</code>
INFO_HIGH	19	INFO	<code>info_high()</code>
DEBUG	10	DEBUG	<code>debug()</code>
NOTSET	0	NOTSET	–

Utility Functions

There are some additional utility functions to perform logging tasks that are common in the IDAES framework.

`idaes.logger.condition(res)`

Get the solver termination condition to log. This isn't a specific value that you can really depend on, just a message to pass on from the solver for the user's benefit. Sometimes the solve is in a try-except, so we'll handle None and str for those cases, where you don't have a real result.

Logging Solver Output

The solver output can be captured and directed to a logger using the `idaes.logger.solver_log(logger, level)` context manager, which uses `pyutilib.misc.capture_output()` to temporarily redirect `sys.stdout` and `sys.stderr` to a string buffer. The `logger` argument is the logger to log to, and the `level` argument is the level at which records are sent to the logger. The output is logged by a separate logging thread, so output can be logged as it is produced instead of after the solve completes. If the `solver_log()` context manager is used, it can be turned on and off by using the `idaes.logger.solver_capture_on()` and `idaes.logger.solver_capture_off()` functions. If the capture is off solver output won't be logged and it will go to standard output as usual.

The `solver_log` context yields an object with `tee` and `thread` attributes. `thread` is the logging thread, which is not needed for most uses. The `tee` attribute should be passed to the `tee` argument of the `solve` method. Tee tells the Pyomo solver to display solver output. The solver log context can provide this argument by determining if the solver output would be logged at the given level.

Example

```
import idaes.logger as idaeslog
import pyomo.environ as pyo

solver = pyo.SolverFactory("ipopt")

model = pyo.ConcreteModel()
model.x = pyo.Var()
model.y = pyo.Var()
model.x.fix(3)
model.c = pyo.Constraint(expr=model.y==model.x**2)
```

(continues on next page)

(continued from previous page)

```
log = idaeslog.getSolveLogger("solver.demo")
log.setLevel(idaeslog.DEBUG)

with idaeslog.solver_log(log, idaeslog.DEBUG) as slc:
    res = solver.solve(model, tee=slc.tee)
```

4.2.11 Modeling Extensions

The IDAES platform includes several modeling extensions that provide additional capabilities including surrogate modeling, material design, and control. A brief description of each is provided below.

Surrogate modeling

ALAMOPY: ALAMO Python

ALAMOPY.ALAMO Options

This page lists in more detail the ALAMOPY options and the relation of ALAMO and ALAMOPY.

Contents

- *ALAMOPY.ALAMO Options*
 - *Basic ALAMOPY.ALAMO options*
 - * *Data Arguments*
 - * *Available Basis Functions*
 - * *ALAMO Regression Options*
 - * *Validation Capabilities*
 - * *File Options*
 - *ALAMOPY results dictionary*
 - * *Output models*
 - * *Fitness metrics*
 - * *Regression description*
 - * *Performance specs*
 - *Advanced user options in depth*
 - * *Custom Basis Functions*
 - * *Custom Constraints*
 - * *Basis Function Groups and Constraints*

Basic ALAMOPY.ALAMO options

Data Arguments

- **xmin, xmax**: minimum/maximum values of inputs, if not given they are calculated
- **zmin, zmax**: minimum/maximum values of outputs, if not given they are calculated
- **xlabels**: user-specified labels given to the inputs
- **zlabels**: user-specified labels given to the outputs

```
alamo(x_inputs, z_outputs, xlabels=['x1','x2'], zlabels=['z1','z2'])
alamo(x_inputs, z_outputs, xmin=(-5,0), xmax=(10,15))
```

Available Basis Functions

- **linfcns, expfcns, logfcns, sinfcns, cosfcns**: 0-1 option to include linear, exponential, logarithmic, sine, and cosine transformations. For example

```
linfcns = 1, expfcns = 1, logfcns = 1, sinfcns = 1, cosfcns = 1
```

This results in basis functions = x_1 , $\exp(x_1)$, $\log(x_1)$, $\sin(x_1)$, $\cos(x_1)$ * **monomialpower, multi2power, multi3power**: list of monomial, binomial, and trinomial powers. For example

```
monomialpower = (2,3,4), multi2power = (1,2,3), multi3power = (1,2,3)
```

This results in the following basis functions:

- Monomial functions = x^2 , x^3 , x^4
- Binomial functions = x_1*x_2 , $(x_1*x_2)^2$, $(x_1*x_2)^3$
- Trinomial functions = $(x_1*x_2*x_3)$, $(x_1*x_2*x_3)^2$, $(x_1*x_2*x_3)^3$
- **ratio**power: list of ratio powers. For example

```
ratio
```

This results in basis functions = (x_1/x_2) , $(x_1/x_2)^2$, $(x_1/x_2)^3$

```
alamo(x_inputs, z_outputs, linfcns=1, logfcns=1, expfcns=1)
alamo(x_inputs, z_outputs, linfcns=1, multi2power=(2,3))
```

Note: Custom basis functions are discussed in the Advanced User Section.

ALAMO Regression Options

- **showalm**: print ALAMO output to the screen
- **expandoutput**: add a key to the output dictionary for multiple outputs
- **solvemip, builder, linearerror**: A 01 indicator to solve with an optimizer (GAMSSOLVER), use a greedy heuristic, or use a linear objective instead of squared error.
- **modeler**: Fitness metric to be used for model building (1-8)
 - 1. **BIC**: Bayesian information criterion

- 2. **Cp**: Mallow’s Cp
 - 3. **AICc**: the corrected Akaike’s information criterion
 - 4. **HQC**: the Hannan-Quinn information criterion
 - 5. **MSE**: mean square error
 - 6. **SSEp**: sum of square error plus a penalty proportional to the model size (Note: convpen is the weight of the penalty)
 - 7. **RIC**: the risk information criterion
 - 8. **MADp**: the maximum absolute deviation plus a penalty proportional to model size (Note: convpen is the weight of the penalty)
- **regularizer**: Regularization method used to reduce the number of potential basis functions before optimization of the selected fitness metric. Possible values are 0 and 1, corresponding to no regularization and regularization with the lasso, respectively.
 - **maxterms**: Maximum number of terms to be fit in the model
 - **convpen**: When MODELER is set to 6 or 8 the size of the model is weighted by CONVPEN.
 - **almopt**: name of the alamo option file
 - **simulator**: a python function to be used as a simulator for ALAMO, a variable that is a python function (not a string)
 - **maxiter**: max iteration of runs

Validation Capabilities

- **xval, zval**: validation input/output variables
- **loo**: leave-one-out evaluation
- **lmo**: leave-many-out evaluation
- **cvfun**: cross-validation function (True/False)

File Options

- **almname**: specify a name for the .alm file
- **savescratch**: saves .alm and .lst
- **savetrace**: saves tracefile
- **saveopt**: save .opt options file
- **savegams**: save the .gms gams file

ALAMOPY results dictionary

The results from `alamopy.alamo` are returned as a python dictionary. The data can be accessed by using the dictionary keys listed below. For example

```
regression_results = doalamo(x_input, z_output, **kwargs)
model = regression_results['model']
```

Output models

- **f(model)**: A callable function
- **pymodel**: name of the python model written
- **model**: string of the regressed model

Note: A python script named after the output variables is written to the current directory. The model can be imported and used for further evaluation, for example to evaluate residuals:

```
import z1
residuals = [y-z1.f(inputs[0],inputs[1]) for y,inputs in zip(z,x)]
```

Fitness metrics

- **size**: number of terms chosen in the regression
- **R2**: R2 value of the regression
- **Objective value metrics**: ssr, rmse, madp

Regression description

- **version**: Version of ALAMO
- **xlabels, zlabels**: The labels used for the inputs/outputs
- **xdata, zdata**: array of xdata/zdata
- **ninputs, nbas**: number of inputs/basis functions

Performance specs

There are three types of regression problems that are used: ordinary linear regression (olr), classic linear regression (clr), and a mixed integer program (mip). Performance metrics include the number of each problems and the time spent on each type of problem. Additionally, the time spent on other operations and the total time are included.

- **numolr, olrtime, numclr, clrtime, nummip, miptime**: number of type of regression problems solved and time
- **othertime**: Time spent on other operations
- **totaltime**: Total time spent on the regression

Advanced user options in depth

Similar to ALAMO, there are advanced capabilities for customization and constrained regression facilitated by methods in ALAMOPY including custom basis functions, custom constraints on the response surface, and basis function groups. These methods interact with the regression using the alamo option file.

Custom Basis Functions

Custom basis functions can be added to the built-in functions to expand the functional forms available. In ALAMO, this can be done with the following syntax

```
NCUSTOMBAS #
BEGIN_CUSTOMBAS
x1^2 * x2^2
END_CUSTOMBAS
```

To use this advanced capability in ALAMOPY, the following function is called. Note it is necessary to use the xlabel assigned to the input parameters.

```
addCustomFunctions(fcn_list)
addCustomFunctions(["x1^2 * x2^2", "...", "..."])
```

Custom Constraints

Custom constraints can be placed on response surface or regressed function of the output variable. In ALAMO, this is controlled using custom constraints, CUSTOMCON. The constraints, a function **g(x_inputs, z_outputs)** are applied to a specific output variable, which is the index of the output variable, and are less than or equal to 0 ($g \leq 0$).

```
CRNCUSTOM #
BEGIN_CUSTOMCON
1 z1 - x1 + x2 + 1
END_CUSTOMCON
```

To use this advanced capability in ALAMOPY, the following function is called. Note it is necessary to use the xlabel assigned to the input parameters.

```
addCustomConstraints(custom_constraint_list, **kwargs)
addCustomConstraints(["1 z1 - x1 + x2 +1", "...", "..."])
```

Basis Function Groups and Constraints

In addition to imposing constraints on the response surface it produces, ALAMO has the ability to enforce constraints on groups of selected basis functions. This can be accomplished using NGROUPS and identifying groups of basis functions. For ALAMO, this is achieved by first defining the groups with

```
NGROUPS 3
BEGIN_GROUPS
# Group-id Member-type Member-indices <Powers>
1 LIN 1 2
2 MONO 1 2
3 GRP 1 2
END_GROUPS
```

To add groups to ALAMOPY, you can use the following methods. Each Basis group has an index number that will be used as reference in the group constraints. The groups are defined by three or four parameters. Options for Member-type are LIN, LOG, EXP, SIN, COS, MONO, MULTI2, MULTI3, RATIO, GRP, RBF, and CUST.

```
addBasisGroup(type_of_function, input_indices, powers)
addBasisGroups(groups)

addBasisGroup("MONO", "1", "2")
addBasisGroups([[ "LIN", "1 2"], [ "MONO", "1", "2"], [ "GRP", "1 2"]])
```

With the groups defined, constraints can be placed on the groups using the constraint-types NMT (no-more-than), ATL (at-least), REQ (requires), and XCL (exclude). For NMT and ATL the integer-parameter is the number of members in the group that should be selected based on the constraint. For REQ and XCL the integer-parameter is the group-id number of excluded or required basis functions.

```
BEGIN_GROUPCON
# Group-id Output-id Constraint-type Integer-parameter
3 1 NMT 1
END_GROUPCON
```

To add the basis constraints to alamopy, you can use the following methods.

```
addBasisConstraint(group_id, output_id, constraint_type, intParam)
addBasisConstraints(groups_constraint_list)

addBasisConstraint(3,1,"NMT",1)
addBasisConstraints([[3,1,"NMT",1]])
```

The purpose of ALAMOPY (Automatic Learning of Algebraic MOdels PYthon wrapper) is to provide a wrapper for the software ALAMO which generates algebraic surrogate models of black-box systems for which a simulator or experimental setup is available. Consider a system for which the outputs \mathbf{z} are an unknown function \mathbf{f} of the system inputs \mathbf{x} . The software identifies a function \mathbf{f} , i.e., a relationship between the inputs and outputs of the system, that best matches data (pairs of \mathbf{x} and corresponding \mathbf{z} values) that are collected via simulation or experimentation.

Basic Usage

ALAMOPY's main function is **alamopy.alamo**. Data can be read in or simulated using available python packages. The main arguments of the `alamopy.alamo` python function are inputs and outputs, which are 2D arrays of data. For example

```
regression_results =alamopy.alamo(x_inputs, z_outputs, **kwargs)
```

where ****kwargs** is a set of named keyword arguments than can be passed to the `alamo` python function to customize the basis function set, names of output files, and other options available in ALAMO.

Warning: The `alamopy.doalamo` function is deprecated. It is being replaced with `alamopy.alamo`

Options for *alamopy.alamo*

Possible arguments to be passed to ALAMO through `do_alamo` and additional arguments that govern the behavior of `doalamo`.

- `xlabels` - list of strings to label the input variables
- `zlabels` - list of strings to label the output variables
- `functions` - `logfcns`, `expfcns`, `cosfcns`, `sinfncns`, `linfcns`, `intercept`. These are '0-1' options to activate these functions
- `monomialpower`, `multi2power`, `multi3power`, `ratio`power. List of terms to be used in the respective basis functions
- `modeler` - integer 1-7 determines the choice of fitness metric
- `solvemip` - '0-1' option that will force the solving of the `.gms` file

These options are specific to *alamopy* and will not change the behavior of the underlying `.alm` file.

- `expandoutput` - '0-1' option that can be used to collect more information from the ALAMO `.lst` and `.trc` file
- `showalm` - '0-1' option that control if the ALAMO output is printed to screen
- `almname` - A string that will assign the name of the `.alm` file
- `outkeys` - '0-1' option for dictionary indexing according to the output labels
- `outkeys` - '0-1' option for dictionary indexing according to the output labels
- `outkeys` - '0-1' option for dictionary indexing according to the output labels
- `savetrace` - '0-1' option that controls the status of the trace file
- `savescratch` - '0-1' option to save the `.alm` and `.lst` files
- `almopt` - A string option that will append a text file of the same name to the end of each `.alm` file to facilitate advanced user access in an automated fashion

ALAMOPY Output

There are multiple outputs from the running *alamopy.alamo*. Outputs include:

- `f(model)`: A callable function
- `pymodel`: name of the python model written
- `model`: string of the regressed model

Note: A python script named after the output variables is written to the current directory. The model can be imported and used for further evaluation, for example to evaluate residuals:

```
import z1
residuals = [y-z1.f(inputs[0],inputs[1]) for y,inputs in zip(z,x)]
```


Additional Results

After the regression of a model, ALAMOPY provides confidence interval analysis and plotting capabilities using the results output.

Plotting

The plotting capabilities of ALAMOPY are available in the **almplot** function. Almplot will plot the function based on one of the inputs.

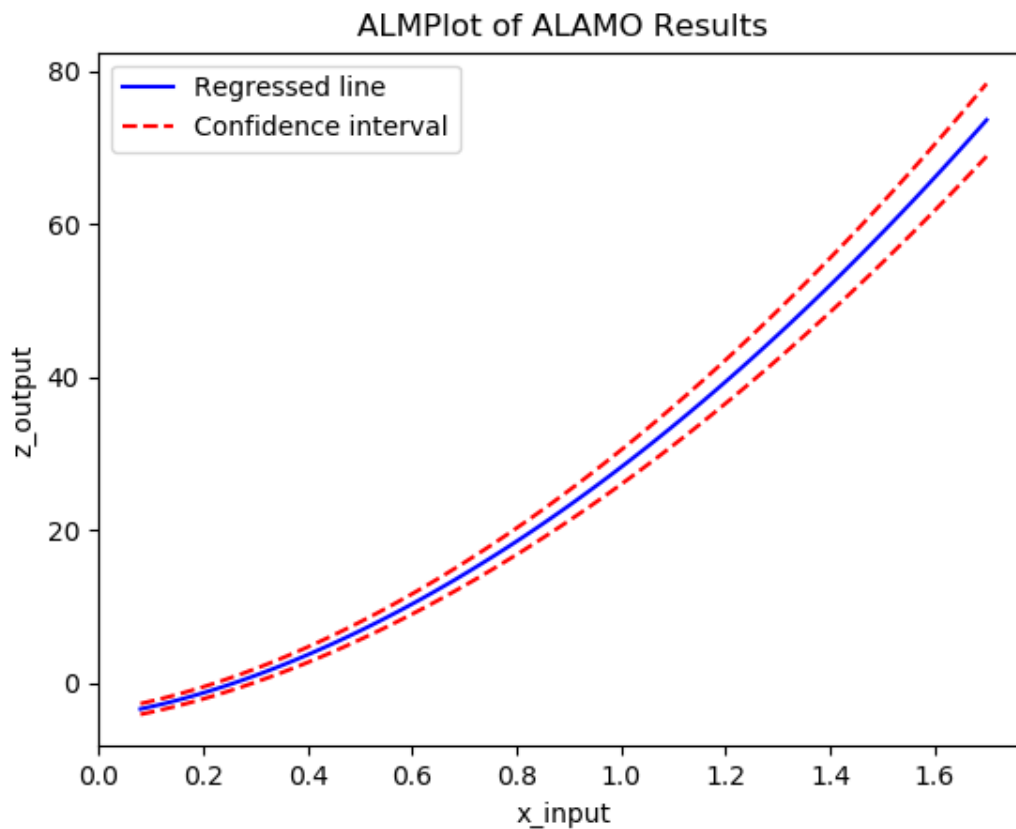
```
result = alamopy.alamo(x_in, z_out, kargs)
alamopy.almplot(result)
```

Confidence intervals

Confidence intervals can similarly be calculated for the weighting of selected basis functions using the **almconfidence** function.

This adds **conf_inv** (confidence intervals) and **covariance** (covariance matrix) to the results dictionary. This also gets incorporated into the plotting function if it is available.

```
result = alamopy.alamo(x_in, z_out, kargs)
result = alamopy.almconfidence(result)
alamopy.almplot(result)
```



Advanced Regression Capabilities

Similar to ALAMO, there are advanced capabilities for customization and constrained regression facilitated by methods in ALAMOPY including custom basis functions, custom constraints on the response surface, and basis function groups. These methods interact with the regression using the alamo option file.

Custom Basis Functions

Custom basis functions can be added to the built-in functions to expand the functional forms available. To use this advanced capability in ALAMOPY, the following function is called. Note it is necessary to use the xlabel assigned to the input parameters.

```
addCustomFunctions(fcn_list)
addCustomFunctions(["x1^2 * x2^2", "...", "..."])
```

Custom Constraints

Custom constraints can be placed on response surface or regressed function of the output variable. In ALAMO, this is controlled using custom constraints, CUSTOMCON. The constraints, a function **g(x_inputs, z_outputs)** are applied to a specific output variable, which is the index of the output variable, and are less than or equal to 0 (**g <= 0**).

To use this advanced capability in ALAMOPY, the following function is called. Note it is necessary to use the xlabel assigned to the input parameters.

```
addCustomConstraints(custom_constraint_list, **kwargs)
addCustomConstraints(["1 z1 - x1 + x2 +1", "...", "..."])
```

Basis Function Groups and Constraints

In addition to imposing constraints on the response surface it produces, ALAMO has the ability to enforce constraints on groups of selected basis functions. To define groups in ALAMOPY, you can use the following methods. Each Basis group has an index number that will be used as reference in the group constraints. The groups are defined by three or four parameters. Options for Member-type are LIN, LOG, EXP, SIN, COS, MONO, MULTI2, MULTI3, RATIO, GRP, RBF, and CUST.

```
addBasisGroup(type_of_function, input_indices, powers)
addBasisGroups(groups)

addBasisGroup("MONO", "1", "2")
addBasisGroups([["LIN", "1 2"], ["MONO", "1", "2"], ["GRP", "1 2"]])
```

With the groups defined, constraints can be placed on the groups using the constraint-types NMT (no-more-than), ATL (at-least), REQ (requires), and XCL (exclude). For NMT and ATL the integer-parameter is the number of members in the group that should be selected based on the constraint. For REQ and XCL the integer-parameter is the group-id number of excluded or required basis functions.

To add the basis constraints to alamopy, you can use the following methods.

```
addBasisConstraint(group_id, output_id, constraint_type, intParam)
addBasisConstraints(groups_constraint_list)

addBasisConstraint(3, 1, "NMT", 1)
addBasisConstraints([["3, 1, "NMT", 1]])
```

ALAMOPY Examples

Three examples are included with ALMAOPY. These examples demonstrate different use cases, and provide a template for utilizing user-defined mechanisms.

- `ackley.py`
- `branin.py`
- `camel6.py` with a Jupyter notebook

RIPE: Reaction Identification and Parameter Estimation

The RIPE module provides tools for reaction network identification. RIPE uses reactor data consisting of concentration, or conversion, values for multiple species that are obtained dynamically, or at multiple process conditions (temperatures, flow rates, working volumes) to identify probable reaction kinetics. The RIPE module also contains tools to facilitate adaptive experimental design. The experimental design tools in RIPE require the use of the python package RBFopt. More information for RBFopt is available at www.github.com/coin-or/rbfopt

Basic Usage

RIPE can be used to build models for static datasets through the function `ripe.ripemodel`

```
ripe_results = ripe.ripemodel(data, kwargs)
```

- `data` is provided to RIPE as one, two, or three dimensional python data structures, where the first axis corresponds to observations at different process conditions, the second axis corresponds to observations of different chemical species, and the third axis corresponds to dynamic observation of a chemical species at a specified process condition.

RIPE adaptive experimental design can be accessed using `ripe.ems`

```
[proposed_x, errors] = ripe.ems(ripe_results, simulator, l_bounds, u_bounds, n_
↪species, kwargs)
```

- `ripe_results` - The results from `ripe.ripemodel`, additional information provided in the results section
- `simulator` - a black-box simulator for the unknown process.
- `l_bounds/u_bounds` - lower and upper bounds for the input variables in the adaptive design
- `nspecies` - the number of chemical species present in the black-box system

Reaction stoichiometries and mechanisms are provided explicitly to `ripemodel` through the keyword arguments `mechanisms` and `stoichiometry`. Detailed explanations of the forms of these arguments are provided in the stoichiometry and mechanism specification section. Additional keyword arguments can be found in the additional options section.

RIPE Output

By default, one file will be generated

- `riperesults.txt` - a file containing the selected reactions and parameter estimates

Reaction Stiochiometry and Mechanism Specification

Considered reaction stiochiometries are provided through keyword arguments.

Stoichiometry

Considered reaction stoichiometries are defined as a list of lists, where reactants and products are defined as negative and positive integers, respectively, according to their stoichiometric coefficients. A set of considered reaction stoichiometries must be provided. If process data consists of species conversion, a positive coefficient should be specified.

Mechanisms

Considered reaction mechanisms are provided explicitly to RIPE through `q` keyword argument. If no kinetic mechanisms are specified, mass action kinetics are ascribed to every considered stoichiometry. RIPE contains kinetic mechanisms defined internally, and called through `ripe.mechs.<mechanism>`. The available mechanisms include:

- `massact` - mass action kinetics, order informed by reaction stoichiometry

19 empirical rate forms included relate specifically to catalyst conversion in chemical looping combustion reactors include:

- Random nucleation
- Power law models
- Avrami-Erofeev models

These internal kinetics can be specified by calling `ripe.mechs.massact` or `ripe.mechs.clcforms` respectively. User-defined kinetic mechanisms can also be supplied to RIPE as python functions. An example is provided in the file `crac.py`.

Additional Results and Options

In addition to the arguments stoichiometry and mechanism, a number of other optional arguments are available, including:

Arguments relating to process conditions

- `x0` - initial concentration at each process condition for every species
- `time` - time associated with dynamic samples for every process condition
- `temp` - temperature associated with every process condition
- `flow` - flow rate at every process condition for every species
- `vol` - reactor volume at every process condition

Arguments related to RIPE algorithmic function

- `tref` - reference temperature for reformulated Arrhenius models
- `ccon` - specified cardinality constraint instead of BIC objective
- `sigma` - expected variance of noise, estimated if not provided
- `onemechper` - one mechanism per stoichiometry in selected model, true by default

Additional arguments

- `minlp_path` - path to baron or other minlp solver, can also be set in `shared.py`
- `alamo_path` - path to alamo, can also be set in `shared.py`
- `expand_output` - provide estimates for noise variance in model results
- `zscale` - linear scaling of observed responses between -1 and 1
- `ascale` - linear scaling of activities between -1 and 1
- `hide_output` - suppress output to terminal
- `keepfiles` - keep scratch files for debugging
- `showpyomo` - show pyomo output to terminal, false by default

RIPE Examples

Three examples are included with RIPE. These examples demonstrate different use cases, and provide a template for utilizing user-defined mechanisms.

- `clc.py` - a chemical looping combustion example in which catalyst conversion is observed over time
- `isoT.py` - an example that utilizes both `ripe.ripemodel` and `ripe.ems`
- `crac.py` - an example that utilizes user-defined reaction mechanisms

All of these examples are built for Linux machines. They can be called from the command line by calling python directly, or can be called from inside a python environment using `execfile()`.

HELMET: HELMholtz Energy Thermodynamics

The purpose of HELMET (HELMholtz Energy Thermodynamics) is to provide a framework for regressing multi-parameter equations of state that identify an equation for Helmholtz energy and multiple thermodynamic properties simultaneously. HELMET uses best subset selection to simultaneously model various thermodynamic properties based on the properties thermodynamic relation to Helmholtz energy. The generated model is a function of reduced density and inverse reduced temperature and uses partial derivatives to calculate the different properties. Constraints are placed on the regression to maintain thermodynamically feasible values and improve extrapolation and behavior of the model based on physical restrictions.

Warning: This is the first public release of HELMET. Future work will include mixtures, regression using Pyomo models, and increased plotting and preprocessing capabilities.

Basic Usage

Warning: To use this software, ALAMOPY and the solver BARON are required.

For the basic use of HELMET, the main regression steps can be imported from `helmet.HELMET`. These functions provide general capabilities of HELMET for new users.

```
import helmet.Helmet as Helmet
```

The methods available in `helmet.Helmet` perform the necessary steps of the regression properties.

1. **`initialize(**kwargs)`**

Initializes key thermodynamic constants, the location of data and sampling, properties to be fit, and optimization settings

- **`molecule`** - name of the chemical of interest, directs naming of files and where the data should exist
- **`fluid_data`** - a tuple containing key thermodynamic constants (critical temperature, critical pressure, critical density, molecular weight, triple point, accentric factor)
- **`filename`** - used for location of data
- **`gamsname`** - used for naming of files
- **`max_time`** - max time used for the solver
- **`props`** - list of thermodynamic properties to be fit

Supported thermodynamic properties are

- Pressure: 'PVT'
- Isochoric heat capacity: 'CV'
- Isobaric heat capacity: 'CP'
- Speed of Sound: 'SND'

- **`sample`** - sample ratio, ex. `sample = 3` then a third of datapoints will be used

2. **`prepareAncillaryEquations(plot=True)`**

Fits equations to saturated vapor and liquid density and vapor pressure. The keyword argument `plot` defaults to `False`

3. **`viewPropertyData()`**

Plots the different thermodynamic properties available and a way to check that the importing of data is successful

4. **`setupRegression(numTerms = 12, gams=True)`**

Writes the optimization program for modelling the thermodynamic properties. Currently this is through GAMS but in the future it can also be solved using Pyomo.

5. **`runRegression()`**

Begins the modelling of the multiparameter equation

6. **`viewResults(filename)`**

Based on the optimization settings, the solution of the regression is parsed and fitness metrics are calculated. The results can be visualized with different plots.

HELMET Output

The output for HELMET is a single equation representing Helmholtz energy. Partial derivatives of this equation will give you the fit thermodynamic properties as well as other properties related to Helmholtz energy.

HELMET Examples

The provided HELMET example uses data modified for this application and made available by the IAPWS organization at <http://www.iapws.org/95data.html> for IAPWS Formulation 1995 for Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use.

PySMO: Python-based Surrogate Modelling Objects

The PySMO toolbox provides tools for generating different types of reduced order models. It provides IDAES users with a set of surrogate modeling tools which supports flowsheeting and direct integration into an equation-oriented modeling framework. It allows users to directly integrate reduced order models with algebraic high-fidelity process models within an single IDAES flowsheet.

PySMO provides two sets of tools necessary for sampling and surrogate model generation.

Surrogate Generation

PySMO offers tools for generating three types of surrogates:

Generating Polynomial Models with PySMO

The *pysmo.polynomial_regression* method learns polynomial models from data. Presented with a small number of samples generated from experiments or computer simulations, the approach determines the most accurate polynomial approximation by comparing the accuracy and performance of polynomials of different orders and basis function forms.

pysmo.polynomial_regression considers three types of basis functions

- univariate polynomials,
- second-degree bivariate polynomials, and
- user-specified basis functions.

Thus, for a problem with m sample points and n input variables, the resulting polynomial is of the form

$$y_k = \sum_{i=1}^n \beta_i x_{ik}^\alpha + \sum_{i,j>i}^n \beta_{ij} x_{ik} x_{jk} + \beta_\Phi \Phi(x_{ik}) \quad i, j = 1, \dots, n; i \neq j; k = 1, \dots, m; \alpha \leq 10 \quad (4.1)$$

Basic Usage

To generate a polynomial model with PySMO, the *pysmo.polynomial_regression* class is first initialized, and then the method `training` is called on the initialized object:

```
# Required imports
>>> from idaes.surrogates.pysmo import polynomial_regression
>>> import pandas as pd

# Load dataset from a csv file
>>> xy_data = pd.read_csv('data.csv', header=None, index_col=0)

# Initialize the PolynomialRegression class, extract the list of features and train_
↳ the model
>>> pr_init = polynomial_regression.PolynomialRegression(xy_data, xy_data, maximum_
↳ polynomial_order=3, *kwargs)
>>> features = pr_init.get_feature_vector()
>>> pr_init.training()
```

- **xy_data** is a two-dimensional python data structure containing the input and output training data. The output values **MUST** be in the last column.
- **maximum_polynomial_order** refers to the maximum polynomial order to be considered when training the surrogate.

Optional Arguments

- **multinomials** - boolean option which determines whether bivariate terms are considered in polynomial generation.
- **training_split** - option which determines fraction of training data to be used for training (the rest will be for testing). Default is 0.8.
- **number_of_crossvalidations** - Number of cross-validations during training. Default number is 3.

pysmo.polynomial_regression Output

The result of the *pysmo.polynomial_regression* method is a python object containing information about the problem set-up, the final optimal polynomial order, the polynomial coefficients and different error and quality-of-fit metrics such as the mean-squared-error (MSE) and the R^2 coefficient-of-fit. A Pyomo expression can be generated from the object simply passing a list of variables into the function *generate_expression*:

```
# Create a python list from the headers of the dataset supplied for training
>>> list_vars = []
>>> for i in features.keys():
>>>     list_vars.append(features[i])
# Pass list to generate_expression function to obtain a Pyomo expression as output
>>> print(pr_init.generate_expression(list_vars))
```


Prediction with *pysmo.polynomial_regression* models

Once a polynomial model has been trained, predictions for values at previously unsampled points :math:x_{unsampled} can be evaluated by calling the `predict_output()` method on the unsampled points:

```
# Create a python list from the headers of the dataset supplied for training
>>> y_unsampled = pr_init.predict_output(x_unsampled)
```

The confidence intervals for the regression paramaters may be viewed using the method `confint_regression`.

Flowsheet Integration

The result of the polynomial training process can be passed directly into a process flowsheet as an objective or a constraint. The following code snippet demonstrates how an output polynomial model may be integrated directly into a Pyomo flowsheet as an objective:

```
# Required imports
>>> import pyomo.environ as pyo
>>> from idaes.surrogates.pysmo import polynomial_regression
>>> import pandas as pd

# Create a Pyomo model
>>> m = pyo.ConcreteModel()
>>> i = pyo.Set(initialize=[1, 2])

# Create a Pyomo variable with indexed by the 2D-set i with initial values {0, 0}
>>> init_x = {1: 0, 2: 0}
>>> def x_init(m, i):
>>>     return (init_x[i])
>>> m.x = pyo.Var(i, initialize=x_init)

# Train a simple polynomial model on data available in csv format, resulting in the_
↪ Python object polyfit
>>> xy_data = pd.read_csv('data.csv', header=None, index_col=0)
>>> pr_init = polynomial_regression.PolynomialRegression(xy_data, xy_data, maximum_
↪ polynomial_order=3)
>>> features = pr_init.get_feature_vector()
>>> polyfit = pr_init.training()

# Use the resulting polynomial as an objective, passing in the Pyomo variable x
>>> m.obj = pyo.Objective(expr=polyfit.generate_expression([m.x[1], m.x[2]]))

# Solve the model
>>> instance = m
>>> opt = pyo.SolverFactory("ipopt")
>>> result = opt.solve(instance, tee=True)
```

Further details about *pysmo.polynomial_regression* may be found by consulting the examples or reading the paper [...]

Available Methods

class `idaes.surrogate.pysmo.polynomial_regression.FeatureScaling`

A class for scaling and unscaling input and output data. The class contains two main methods: `data_scaling` and `data_unscaling`

static `data_scaling` (*data*)

`data_scaling` performs column-wise minimax scaling on the input dataset.

Parameters *data* – The input data set to be scaled. Must be a numpy array or dataframe.

Returns

tuple containing:

- **scaled_data** : A 2-D Numpy Array containing the scaled data. All array values will be between [0, 1].
- **data_minimum** : A 2-D row vector containing the column-wise minimums of the input data.
- **data_maximum** : A 2-D row vector containing the column-wise maximums of the input data.

Return type (tuple)

Raises `TypeError` – Raised when the input data is not a numpy array or dataframe

static `data_unscaling` (*x_scaled*, *x_min*, *x_max*)

`data_unscaling` performs column-wise un-scaling on the a minmax-scaled input dataset.

Parameters

- **x_scaled** (*NumPy Array*) – Data to be un-scaled. Data values should be between 0 and 1.
- **x_min** (*NumPy vector*) – $n \times 1$ vector containing the actual minimum value for each column. Must contain same number of elements as the number of columns in *x_scaled*.
- **x_max** (*NumPy vector*) – $n \times 1$ vector containing the actual minimum value for each column. Must contain same number of elements as the number of columns in *x_scaled*.

Returns A 2-D numpy array containing the scaled data, $x_{min} + x_{scaled} * (x_{max} - x_{min})$

Return type NumPy Array

Raises `IndexError` – Raised when the dimensions of the arrays are inconsistent.

```

class idaes.surrogate.pyomo.polynomial_regression.PolynomialRegression(original_data_input,
                                                                    re-
                                                                    gres-
                                                                    sion_data_input,
                                                                    max-
                                                                    i-
                                                                    mum_polynomial_order,
                                                                    num-
                                                                    ber_of_crossvalidations=None,
                                                                    no_adaptive_samples=None,
                                                                    train-
                                                                    ing_split=None,
                                                                    max_fraction_training_samp
                                                                    max_iter=None,
                                                                    so-
                                                                    lu-
                                                                    sion_method=None,
                                                                    multi-
                                                                    no-
                                                                    mi-
                                                                    als=None,
                                                                    fname=None,
                                                                    over-
                                                                    write=False)

```

The PolynomialRegression class performs polynomial regression on a training data set.

The class must first be initialized by calling PolynomialRegression. Regression is then carried out by calling training.

For a given dataset with n features x_1, x_2, \dots, x_n , Polyregression is able to consider three types of basis functions:

- (a) Mononomial terms ($x_i^p, p \leq 10$) for all individual features. The maximum degree to be considered can be set by the user (**maximum_polynomial_order**)
- (b) All first order interaction terms x_1x_2, x_1x_3 etc. This can be turned on or off by the user (set **multinomials**)
- (c) User defined input features, e.g. $\sin(x_1)$. These must be Pyomo functions and should be provided as a list by the user calling `set_additional_terms` method before the polynomial training is done.

Example:

```

# Initialize the class and set additional terms
>>> d = PolynomialRegression(full_data, training_data, maximum_polynomial_order=2,
    ↪ max_iter=20, multinomials=1, solution_method='pyomo')
>>> p = d.get_feature_vector()
>>> d.set_additional_terms([...extra terms...])

# Train polynomial model and predict output for an test data x_test
>>> d.training()
>>> predictions = d.predict_output(x_test)

```

Parameters

- **regression_data_input** (NumPy Array of Pandas Dataframe) – The dataset for regression training. It is expected to contain the features (X) and output (Y) data, with the output values (Y) in the last column.

- **original_data_input** (*NumPy Array of Pandas Dataframe*) – If **regression_data_input** was drawn from a larger dataset by some sampling approach, the larger dataset may be provided here. When additional data is not available, the same data supplied for training_data can be supplied - this tells the algorithm not to carry out adaptive sampling.
- **maximum_polynomial_order** (*int*) – The maximum polynomial order to be considered.

Further details about the optional inputs may be found under the `__init__` method.

```
__init__(original_data_input, regression_data_input, maximum_polynomial_order, number_of_crossvalidations=None, no_adaptive_samples=None, training_split=None, max_fraction_training_samples=None, max_iter=None, solution_method=None, multinomials=None, fname=None, overwrite=False)
```

Initialization of PolynomialRegression class.

Parameters

- **regression_data_input** (*NumPy Array of Pandas Dataframe*) – The dataset for regression training. It is expected to contain features and output data, with the output values (Y) in the last column.
- **original_data_input** (*NumPy Array of Pandas Dataframe*) – If **regression_data_input** was drawn from a larger dataset by some sampling approach, the larger dataset may be provided here.
- **maximum_polynomial_order** (*int*) – The maximum polynomial order to be considered.

Keyword Arguments

- **number_of_crossvalidations** (*int*) – The number of polynomial fittings and cross-validations to be carried out for each polynomial function/expression. Must be a positive, non-zero integer. Default=3.
- **training_split** (*float*) – The training/test split to be used for regression_data_input. Must be between 0 and 1. Default = 0.75
- **solution_method** (*str*) – The method to be used for solving the least squares optimization problem for polynomial regression. Three options are available:
 - (a) "MLE" : The mle (maximum likelihood estimate) method solves the least squares problem using linear algebra. Details of the method may be found in Forrester et al.
 - (b) "BFGS" : This approach solves the least squares problem using scipy's BFGS algorithm.
 - (c) "pyomo": This option solves the optimization problem in pyomo with IPOPT as solver. This is the default option.
- **multinomials** (*bool*) – This option determines whether or not multinomial terms are considered during polynomial fitting. Takes 0 for No and 1 for Yes. Default = 1.

Returns `self` object containing all the input information.

Raises

- **ValueError** –
 - The input datasets (**original_data_input** or **regression_data_input**) are of the wrong type (not Numpy arrays or Pandas Dataframes)
- **Exception** –

- **maximum_polynomial_order** is not a positive, non-zero integer or **maximum_polynomial_order** is higher than the number of training samples available
- **Exception** –
 - **solution_method** is not ‘mle’, ‘pyomo’ or ‘bfgs’
- **Exception** –
 - **multinomials** is not binary (0 or 1)
- **Exception** –
 - **training_split** is not between 0 and 1
- **Exception** –
 - **number_of_crossvalidations** is not a positive, non-zero integer
- **Exception** –
 - **max_fraction_training_samples** is not between 0 and 1
- **Exception** –
 - **no_adaptive_samples** is not a positive, non-zero integer
- **Exception** –
 - **max_iter** is not a positive, non-zero integer
- **warnings.warn** –
 - When the number of cross-validations is too high, i.e. **number_of_crossvalidations** > 10

confint_regression (*confidence=0.95*)

The **confint_regression** method prints the confidence intervals for the regression parameters.

Parameters confidence – Required confidence interval level, default = 0.95 (95%)

generate_expression (*variable_list*)

The **generate_expression** method returns the Pyomo expression for the polynomial model trained.

The expression is constructed based on a supplied list of variables **variable_list** and the output of training.

Parameters variable_list (*list*) – List of input variables to be used in generating expression. This can be the a list generated from the results of **get_feature_vector**. The user can also choose to supply a new list of the appropriate length.

Returns Pyomo expression of the polynomial model based on the variables provided in **variable_list**.

Return type Pyomo Expression

get_feature_vector ()

The **get_feature_vector** method generates the list of regression features from the column headers of the input dataset.

Returns An indexed parameter list of the variables supplied in the original data

Return type Pyomo IndexedParam

Example:

```
# Create a small dataframe with three columns ('one', 'two', 'three') and two
↳rows (A, B)
>>> xy_data = pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
↳orient='index', columns=['one', 'two', 'three'])

# Initialize the PolynomialRegression class and print the column headers
↳for the variables
>>> f = PolynomialRegression(xy_data, xy_data, maximum_polynomial_order=1,
↳multinomials=True, training_split=0.8)
>>> p = f.get_feature_vector()
>>> for i in p.keys():
>>>     print(i)
one
two
```

predict_output (*x_data*)

The `predict_output` method generates output predictions for input data `x_data` based a previously generated polynomial fitting.

Parameters `x_data` – Numpy array of designs for which the output is to be evaluated/predicted.

Returns Output variable predictions based on the polynomial fit.

Return type Numpy Array

set_additional_terms (*term_list*)

`set_additional_terms` accepts additional user-defined features for consideration during regression.

Parameters `term_list` (*list*) – List of additional terms to be considered as regression features. Each term in the list must be a Pyomo-supported intrinsic function.

Example:

```
# To add the sine and cosine of a variable with header 'X1' in the dataset as
↳additional regression features:
>>> xy_data = pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
↳orient='index', columns=['X1', 'X2', 'Y'])
>>> A = PolynomialRegression(xy_data, xy_data, maximum_polynomial_order=5)
>>> p = A.get_feature_vector()
>>> A.set_additional_terms([ pyo.sin(p['X1']) , pyo.cos(p['X1']) ])
```

training ()

The `training` method trains a polynomial model to an input dataset. It calls the core method which is called in the `PolynomialRegression` class (`polynomial_regression_fitting`). It accepts no user input, inheriting the information passed in class initialization.

Returns

Python Object (results) containing the results of the polynomial regression process including:

- the polynomial order (**self.final_polynomial_order**)
- polynomial coefficients (**self.optimal_weights_array**), and
- MAE and MSE errors as well as the R^2 (**results.errors**).

Return type `tuple`

References:

[1] Forrester et al.'s book "Engineering Design via Surrogate Modelling: A Practical Guide", <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470770801>

Generating Radial Basis Function (RBF) models with PySMO

The `pysmo.radial_basis_function` package has the capability to generate different types of RBF surrogates from data based on the basis function selected. RBFs models are usually of the form where

$$y_k = \sum_{j=1}^{\Omega} w_j \psi(\|x_k - z_j\|) \quad k = 1, \dots, m \quad (4.2)$$

where z_j are basis function centers (in this case, the training data points), w_j are the radial weights associated with each center z_j , and ψ is a basis function transformation of the Euclidean distances.

PySMO offers a range of basis function transformations ψ , as shown in the table below.

Table 3: List of available RBF basis transformations, $d = \|x_k - z_j\|$

Transformation type	PySMO option name	$\psi(d)$
Linear	'linear'	d
Cubic	'cubic'	d^3
Thin-plate spline	'spline'	$d^2 \ln(d)$
Gaussian	'gaussian'	$e^{(-d^2 \sigma^2)}$
Multiquadric	'mq'	$\sqrt{1 + (\sigma d)^2}$
Inverse mMultiquadric	'imq'	$1/\sqrt{1 + (\sigma d)^2}$

Selection of parametric basis functions increase the flexibility of the radial basis function but adds an extra parameter (σ) to be estimated.

Basic Usage

To generate an RBF model with PySMO, the `pysmo.radial_basis_function` class is first initialized, and then the function `training` is called on the initialized object:

```
# Required imports
>>> from idaes.surrogates.pysmo import radial_basis_function
>>> import pandas as pd

# Load dataset from a csv file
>>> xy_data = pd.read_csv('data.csv', header=None, index_col=0)

# Initialize the RadialBasisFunctions class, extract the list of features and train
↳ the model
>>> rbf_init = radial_basis_function.RadialBasisFunctions(xy_data, *kwargs)
>>> features = rbf_init.get_feature_vector()
>>> rbf_fit = rbf_init.training()
```

- *xy_data* is a two-dimensional python data structure containing the input and output training data. The output values **MUST** be in the last column.

Optional Arguments

- *basis_function* - option to specify the type of basis function to be used in the RBF model. Default is 'gaussian'.
- *regularization* - boolean which determines whether regularization of the RBF model is considered. Default is True.
 - When regularization is turned on, the resulting model is a regressing RBF model.
 - When regularization is turned off, the resulting model is an interpolating RBF model.

pysmo.radial_basis_function Output

The result of *pysmo.radial_basis_function* (*rbf_fit* in above example) is a python object containing information about the problem set-up, the optimal radial basis function weights w_j and different error and quality-of-fit metrics such as the mean-squared-error (MSE) and the R^2 coefficient-of-fit. A Pyomo expression can be generated from the object simply passing a list of variables into the function *generate_expression*:

```
# Create a python list from the headers of the dataset supplied for training
>>> list_vars = []
>>> for i in features.keys():
>>>     list_vars.append(features[i])

# Pass list to generate_expression function to obtain a Pyomo expression as output
>>> print(rbf_init.generate_expression(list_vars))
```

Similar to the *pysmo.polynomial_regression* module, the output of the *generate_expression* function can be passed into an IDAES or Pyomo module as a constraint, objective or expression.

Prediction with *pysmo.radial_basis_function* models

Once an RBF model has been trained, predictions for values at previously unsampled points *x_unsampled* can be evaluated by calling the *predict_output()* function on the unsampled points:

```
# Create a python list from the headers of the dataset supplied for training
>>> y_unsampled = rbf_init.predict_output(x_unsampled)
```

Further details about *pysmo.radial_basis_function* module may be found by consulting the examples or reading the paper [...]

Available Methods

class `idaes.surrogate.pysmo.radial_basis_function.FeatureScaling`

A class for scaling and unscaling input and output data. The class contains two main methods: *data_scaling_minmax* and *data_unscaling_minmax*

static `data_scaling_minmax(data)`

data_scaling_minmax performs column-wise min-max scaling on the input dataset.

Parameters *data* – The input data set to be scaled. Must be a numpy array or dataframe.

Returns

tuple containing:

- **scaled_data** : A 2-D Numpy Array containing the scaled data. All array values will be between [0, 1].
- **data_minimum** : A 2-D row vector containing the column-wise minimums of the input data.
- **data_maximum** : A 2-D row vector containing the column-wise maximums of the input data.

Return type (tuple)

Raises **TypeError** – Raised when the input data is not a numpy array or dataframe

static data_unscaling_minmax (*x_scaled, x_min, x_max*)

`data_unscaling_minmax` performs column-wise un-scaling on the a minmax-scaled input dataset.

Parameters

- **x_scaled** (*NumPy Array*) – Data to be un-scaled. Data values should be between 0 and 1.
- **x_min** (*NumPy vector*) – $n \times 1$ vector containing the actual minimum value for each column. Must contain same number of elements as the number of columns in `x_scaled`.
- **x_max** (*NumPy vector*) – $n \times 1$ vector vector containing the actual minimum value for each column. Must contain same number of elements as the number of columns in `x_scaled`.

Returns A 2-D numpy array containing the scaled data, $x_{min} + x_{scaled} * (x_{max} - x_{min})$

Return type NumPy Array

Raises **IndexError** – Raised when the dimensions of the arrays are inconsistent.

```
class idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunctions (XY_data,
                                                                    ba-
                                                                    sis_function=None,
                                                                    so-
                                                                    lu-
                                                                    tion_method=None,
                                                                    reg-
                                                                    u-
                                                                    lar-
                                                                    iza-
                                                                    tion=None,
                                                                    fname=None,
                                                                    over-
                                                                    write=False)
```

The `RadialBasisFunctions` class generates a radial basis function fitting for a training data set.

The class must first be initialized by calling **RadialBasisFunctions**. Regression is then carried out by calling the method `training`.

For a given dataset with n features x_1, \dots, x_n , `RadialBasisFunctions` is able to consider six types of basis transformations:

- Linear ('linear')
- Cubic ('cubic')
- Gaussian ('gaussian')
- Multiquadric ('mq')

- Inverse multiquadric ('imq')
- Thin-plate spline ('spline')

training selects the best hyperparameters (regularization parameter λ and shape parameter σ , where necessary) by evaluating the leave-one-out cross-validation error for each (λ, σ) pair.

It should be noted that the all the training points are treated as centres for the RBF, resulting in a square system.

Example:

```
# Initialize the class
>>> d = RadialBasisFunctions(training_data, basis_function='gaussian', solution_
↪method='pyomo', regularization=True))
>>> p = d.get_feature_vector()

# Train RBF model and predict output for an test data x_test
>>> d.training()
>>> predictions = d.predict_output(x_test)
```

Parameters `XY_data` (*Numpy Array or Pandas Dataframe*) – The dataset for RBF training. **`XY_data`** is expected to contain the features (X) and output (Y) data, with the output values (Y) in the last column.

Further details about the optional inputs may be found under the `__init__` method.

`__init__` (*XY_data*, *basis_function=None*, *solution_method=None*, *regularization=None*, *fname=None*, *overwrite=False*)
Initialization of **RadialBasisFunctions** class.

Parameters `XY_data` (*Numpy Array or Pandas Dataframe*) – The dataset for RBF training. **`XY_data`** is expected to contain feature and output information, with the output values (y) in the last column.

Keyword Arguments

- **`basis_function`** (*str*) – The basis function transformation to be applied to the training data. Two classes of basis transformations are available for selection:
 - Fixed basis transformations, which require no shape parameter σ :
 - (a) 'cubic' : Cubic basis transformation
 - (b) 'linear' : Linear basis transformation
 - (c) 'spline' : Thin-plate spline basis transformation
 - Parametric basis transformations which require a shape parameter:
 - (a) 'gaussian' : Gaussian basis transformation (Default)
 - (b) 'mq' : Multiquadric basis transformation
 - (c) 'imq' : Inverse multiquadric basis transformation
- **`solution_method`** (*str*) – The method to be used for solving the RBF least squares optimization problem. Three options are available:
 - (a) 'algebraic' : The explicit algebraic method solves the least squares problem using linear algebra.
 - (b) 'BFGS' : This approach solves the least squares problem using SciPy's BFGS algorithm.

(c) 'pyomo' : This option solves the optimization problem in Pyomo with IPOPT as solver. This is the default.

- **regularization** (*bool*) – This option determines whether or not the regularization parameter λ is considered during RBF fitting. Default setting is True.

Returns self object with the input information

Raises

- **ValueError** – The input dataset is of the wrong type (not a NumPy array or Pandas Dataframe)
- **Exception** –
 - **basis_function** entry is not valid.
- **Exception** –
 - **solution_method** is not 'algebraic', 'pyomo' or 'bfgs'.
- **Exception** –
 - λ is not boolean.

Example:

```
# Specify the gaussian basis transformation
>>> d = RadialBasisFunctions(XY_data, basis_function='gaussian')
```

generate_expression (*variable_list*)

The generate_expression method returns the Pyomo expression for the RBF model trained.

The expression is constructed based on the supplied list of variables **variable_list** and the results of the previous RBF training process.

Parameters **variable_list** (*list*) – List of input variables to be used in generating expression. This can be the a list generated from the output of get_feature_vector. The user can also choose to supply a new list of the appropriate length.

Returns Pyomo expression of the RBF model based on the variables provided in **variable_list**

Return type Pyomo Expression

get_feature_vector ()

The get_feature_vector method generates the list of regression features from the column headers of the input dataset.

Returns An indexed parameter list of the variables supplied in the original data

Return type Pyomo IndexedParam

Example:

```
# Create a small dataframe with three columns ('one', 'two', 'three') and two
↳ rows (A, B)
>>> xy_data = pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
↳ orient='index', columns=['one', 'two', 'three'])

# Initialize the **RadialBasisFunctions** class with a linear kernel and
↳ print the column headers for the variables
>>> f = RadialBasisFunctions(xy_data, basis_function='linear')
>>> p = f.get_feature_vector()
>>> for i in p.keys():
```

(continues on next page)

(continued from previous page)

```
>>> print(i)
one
two
```

predict_output (*x_data*)

The `predict_output` method generates output predictions for input data `x_data` based a previously generated RBF fitting.

Parameters `x_data` (*NumPy Array*) – Designs for which the output is to be evaluated/predicted.

Returns Output variable predictions based on the rbf fit.

Return type Numpy Array

static r2_calculation (*y_true, y_predicted*)

`r2_calculation` returns the R^2 as a measure-of-fit between the true and predicted values of the output variable.

Parameters

- **y_true** (*NumPy Array*) – Vector of actual values of the output variable
- **y_predicted** (*NumPy Array*) – Vector of predictions for the output variable based on the surrogate

Returns R^2 measure-of-fit between actual and predicted data

Return type float

training ()

Main function for RBF training.

To train the RBF:

- (1) The best values of the hyperparameters (σ , λ) are selected via LOOCV.
- (2) The necessary basis transformation at the optimal hyperparameters is generated.
- (3) The condition number for the transformed matrix is calculated.
- (4) The optimal radial weights are evaluated using the selected optimization method.
- (5) The training predictions, prediction errors and r-square coefficient of fit are evaluated by calling the methods `error_calculation` and `r2_calculation`
- (6) A results object is generated by calling the `ResultsReport` class.

The LOOCV error for each (σ , λ) pair is evaluated by calling the function `loo_error_estimation_with_rippa_method`.

The pre-defined shape parameter set considers 24 irregularly spaced values ranging between 0.001 - 1000, while the regularization parameter set considers 21 values ranging between 0.00001 - 1.

Returns

self object (results) containing the all information about the best RBF fitting obtained, including:

- the optimal radial weights (**results.radial_weights**),
- when relevant, the optimal shape parameter found σ (**results.sigma**),
- when relevant, the optimal regularization parameter found λ (**results.regularization**),

- the RBF predictions for the training data (**results.output_predictions**), and
- the R^2 value on the training data (**results.R2**)

Return type `tuple`

References:

- [1] Forrester et al.'s book "Engineering Design via Surrogate Modelling: A Practical Guide", <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470770801>
- [2] Hongbing Fang & Mark F. Horstemeyer (2006): Global response approximation with radial basis functions, <https://www.tandfonline.com/doi/full/10.1080/03052150500422294>
- [3] Rippa, S. (1999): An algorithm for selecting a good value for the parameter c in radial basis function interpolation, <https://doi.org/10.1023/A:1018975909870>
- [4] Mongillo M.A. (2011) Choosing Basis Functions and Shape Parameters for Radial Basis Function Methods, <https://doi.org/10.1137/11S010840>

Generating Kriging Models with PySMO

The *pysmo.kriging* trains Ordinary Kriging models. Interpolating kriging models assume that the outputs $\hat{y} \in \mathbb{R}^{m \times 1}$ are correlated and may be treated as a normally distributed stochastic process. For a set of input measurements $X = \{x_1, x_2, \dots, x_m\}; x_i \in \mathbb{R}^n$, the output \hat{y} is modeled as the sum of a mean (μ) and a Gaussian process error,

$$\hat{y}_k = \mu + \epsilon(x_k) \quad k = 1, \dots, m \quad (4.3)$$

Kriging models assume that the errors in the outputs ϵ are correlated proportionally to the distance between corresponding points,

$$\text{cor}[\epsilon(x_j), \epsilon(x_k)] = \exp\left(-\sum_{i=1}^n \theta_i |x_{ij} - x_{ik}|^{\tau_i}\right) \quad j, k = 1, \dots, m; \tau_i \in [1, 2]; \theta_i \geq 0 \quad (4.4)$$

The hyperparameters of the Kriging model $(\mu, \sigma^2, \theta_1, \dots, \theta_n, \tau_1, \dots, \tau_n)$ are selected such that the concentrated log likelihood function is maximized.

Basic Usage

To generate a Kriging model with PySMO, the *pysmo.kriging* class is first initialized, and then the function *training* is called on the initialized object:

```
# Required imports
>>> from idaes.surrogates.pysmo import kriging
>>> import pandas as pd

# Load dataset from a csv file
>>> xy_data = pd.read_csv('data.csv', header=None, index_col=0)

# Initialize the KrigingModel class, extract the list of features and train the model
>>> krg_init = kriging.KrigingModel(xy_data, *kwargs)
>>> features = krg_init.get_feature_vector()
>>> krg_init.training()
```

- *xy_data* is a two-dimensional python data structure containing the input and output training data. The output values **MUST** be in the last column.

Optional Arguments

- *numerical_gradients*: Whether or not numerical gradients should be used in training. This choice determines the algorithm used to solve the problem.
 - True: The problem is solved with BFGS using central differencing with $\Delta = 10^{-6}$ to evaluate numerical gradients.
 - False: The problem is solved with Basinhopping, a stochastic optimization algorithm.
- *regularization* - Boolean option which determines whether or not regularization is considered during Kriging training. Default is True.
 - When regularization is turned on, the resulting model is a regressing kriging model.
 - When regularization is turned off, the resulting model is an interpolating kriging model.

pysmo.kriging Output

The result of *pysmo.kriging* is a python object containing information about the optimal Kriging hyperparameters $(\mu, \sigma^2, \theta_1, \dots, \theta_n)$ and different error and quality-of-fit metrics such as the mean-squared-error (MSE) and the R^2 coefficient-of-fit. A Pyomo expression can be generated from the object simply passing a list of variables into the function *generate_expression*:

```
# Create a python list from the headers of the dataset supplied for training
>>> list_vars = []
>>> for i in features.keys():
>>>     list_vars.append(features[i])

# Pass list to generate_expression function to obtain a Pyomo expression as output
>>> print(krg_init.generate_expression(list_vars))
```

Similar to the *pysmo.polynomial_regression* module, the output of the *generate_expression* function can be passed into an IDAES or Pyomo module as a constraint, objective or expression.

Prediction with *pysmo.kriging* models

Once a Kriging model has been trained, predictions for values at previously unsampled points *x_unsampled* can be evaluated by calling the *predict_output()* function on the unsampled points:

```
# Create a python list from the headers of the dataset supplied for training
>>> y_unsampled = kriging_init.predict_output(x_unsampled)
```

Further details about *pysmo.kriging* module may be found by consulting the examples or reading the paper [...]

Available Methods

```
class idaes.surrogate.pysmo.kriging.KrigingModel (XY_data, numerical_gradients=True,
                                                regularization=True, fname=None,
                                                overwrite=False)
```

The KrigingModel class trains a Kriging model for a training data set.

The class must first be initialized by calling **KrigingModel**. Model training is then carried out by calling the training method.

KrigingModel is able to generate either an interpolating or a regressing Kriging model depending on the settings used during initialization..

Example:

```
# Initialize the class
>>> d = KrigingModel(training_data, numerical_gradients=True,
    ↪ regularization=True)
>>> p = d.get_feature_vector()

# Train Kriging model and predict output for an test data x_test
>>> d.training()
>>> predictions = d.predict_output(x_test)
```

Parameters XY_data (*NumPy Array or Pandas Dataframe*) – The dataset for Kriging training. **XY_data** is expected to contain both the features (X) and output (Y) information, with the output values (Y) in the last column.

Further details about the optional inputs may be found under the `__init__` method.

`__init__` (*XY_data, numerical_gradients=True, regularization=True, fname=None, overwrite=False*)
Initialization of **KrigingModel** class.

Parameters XY_data (*NumPy Array or Pandas Dataframe*) – The dataset for Kriging training. **XY_data** is expected to contain feature and output data, with the output values (y) in the last column.

Keyword Arguments

- **numerical_gradients** (*bool*) – Whether or not numerical gradients should be used in training. This choice determines the algorithm used to solve the problem.
 - `numerical_gradients = True`: The problem is solved with BFGS using central differencing with a step size of 10^{-6} to evaluate numerical gradients.
 - `numerical_gradients = False`: The problem is solved with Basinhopping, a stochastic optimization algorithm.

- **regularization** (*bool*) – This option determines whether or not regularization is considered during Kriging training. Default is True.
 - When regularization is turned off, the model generates an interpolating kriging model.

Returns self object with the input information and settings.

Raises

- **ValueError** –
 - The input dataset is of the wrong type (not a NumPy array or Pandas Dataframe)
- **Exception** –
 - numerical_gradients is not boolean
- **Exception** –
 - regularization is not boolean

Example:

```
# Initialize Kriging class with no numerical gradients - solution algorithm_  
↪ will be Basinhopping  
>>> d = KrigingModel(XY_data, numerical_gradients=False)
```

generate_expression (*variable_list*)

The generate_expression method returns the Pyomo expression for the Kriging model trained.

The expression is constructed based on the supplied list of variables **variable_list** and the results of the previous Kriging training process.

Parameters **variable_list** (*list*) – List of input variables to be used in generating expression. This can be the a list generated from the output of `get_feature_vector`. The user can also choose to supply a new list of the appropriate length.

Returns Pyomo expression of the Kriging model based on the variables provided in **variable_list**

Return type Pyomo Expression

get_feature_vector ()

The get_feature_vector method generates the list of regression features from the column headers of the input dataset.

Returns An indexed parameter list of the variables supplied in the original data

Return type Pyomo IndexedParam

predict_output (*x_pred*)

The predict_output method generates output predictions for input data `x_pred` based a previously trained Kriging model.

Parameters **x_pred** (*NumPy Array*) – Array of designs for which the output is to be evaluated/predicted.

Returns Output variable predictions based on the Kriging model.

Return type NumPy Array

static r2_calculation (*y_true, y_predicted*)

`r2_calculation` returns the R^2 as a measure-of-fit between the true and predicted values of the output variable.

Parameters

- **y_true** (*NumPy Array*) – Vector of actual values of the output variable
- **y_predicted** (*NumPy Array*) – Vector of predictions for the output variable based on the surrogate

Returns R^2 measure-of-fit between actual and predicted data

Return type `float`

training()

Main function for Kriging training.

To train the Kriging model:

- (1) The Kriging exponent τ_i is fixed at 2.
- (2) The optimal Kriging hyperparameters $(\mu, \sigma^2, \theta_1, \dots, \theta_n)$ are evaluated by calling the `optimal_parameter_evaluation` method using either BFGS or Basinhopping.
- (3) The training predictions, prediction errors and r-square coefficient of fit are evaluated by calling the functions `error_calculation` and `self.r2_calculation`
- (4) A results object is generated by calling the `ResultsReport` class.

Returns

self object (results) containing the all information about the best Kriging model obtained, including:

- the Kriging model hyperparameters (**results.optimal_weights**),
- when relevant, the optimal regularization parameter found λ (**results.regularization_parameter**),
- the Kriging mean (**results.optimal_mean**),
- the Kriging variance (**results.optimal_variance**),
- the Kriging model regularized co-variance matrix (**results.optimal_covariance_matrix**),
- the inverse of the co-variance matrix (**results.covariance_matrix_inverse**),
- the RBF predictions for the training data (**results.output_predictions**),
- the RMSE of the training output predictions (**results.training_rmse**), and
- the R^2 value on the training data (**results.R2**)

Return type `tuple`

References:

- [1] Forrester et al.'s book "Engineering Design via Surrogate Modelling: A Practical Guide", <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470770801>
- [2] D. R. Jones, A taxonomy of global optimization methods based on response surfaces, Journal of Global Optimization, <https://link.springer.com/article/10.1023%2FA%3A1012771025575>

Sampling

The PySMO package offers five common sampling methods for one-shot design:

Latin Hypercube Sampling (LHS)

LHS is a stratified random sampling method originally developed for efficient uncertainty assessment. LHS partitions the parameter space into bins of equal probability with the goal of attaining a more even distribution of sample points in the parameter space that would be possible with pure random sampling.

The `pysmo.sampling.LatinHypercubeSampling` method carries out Latin Hypercube sampling. This can be done in two modes:

- The samples can be selected from a user-provided dataset, or
- The samples can be generated from a set of provided bounds.

Available Methods

```
class idaes.surrogate.pysmo.sampling.LatinHypercubeSampling(data_input,    num-  
                                                             ber_of_samples=None,  
                                                             sam-  
                                                             pling_type=None)
```

A class that performs Latin Hypercube Sampling. The function returns LHS samples which have been selected randomly after sample space stratification.

It should be noted that no minimax criterion has been used in this implementation, so the LHS samples selected will not have space-filling properties.

To use: call class with inputs, and then run `sample_points` method.

Example:

```
# To select 10 LHS samples from "data"  
>>> b = rbf.LatinHypercubeSampling(data, 10, sampling_type="selection")  
>>> samples = b.sample_points()
```

```
__init__(data_input, number_of_samples=None, sampling_type=None)
```

Initialization of **LatinHypercubeSampling** class. Two inputs are required.

Parameters

- **data_input** (*NumPy Array, Pandas Dataframe or list*) – The input data set or range to be sampled.
 - When the aim is to select a set of samples from an existing dataset, the dataset must be a NumPy Array or a Pandas Dataframe and **sampling_type** option must be set to “selection”. The output variable (y) is assumed to be supplied in the last column.
 - When the aim is to generate a set of samples from a data range, the dataset must be a list containing two lists of equal lengths which contain the variable bounds and **sampling_type** option must be set to “creation”. It is assumed that no range contains no output variable information in this case.
- **number_of_samples** (*int*) – The number of samples to be generated. Should be a positive integer less than or equal to the number of entries (rows) in **data_input**.

- **sampling_type** (*str*) – Option which determines whether the algorithm selects samples from an existing dataset (“selection”) or attempts to generate sample from a supplied range (“creation”). Default is “creation”.

Returns **self** function containing the input information

Raises

- **ValueError** – The input data (**data_input**) is the wrong type.
- **Exception** – When **number_of_samples** is invalid (not an integer, too large, zero, or negative)

sample_points()

sample_points generates or selects Latin Hypercube samples from an input dataset or data range. When called, it:

1. generates samples points from stratified regions by calling the `lhs_points_generation`,
2. generates potential sample points by random shuffling, and
3. when a dataset is provided, selects the closest available samples to the theoretical sample points from within the input data.

Returns A numpy array or Pandas dataframe containing **number_of_samples** points selected or generated by LHS.

Return type NumPy Array or Pandas Dataframe

References

- [1] Loeven et al paper titled “A Probabilistic Radial Basis Function Approach for Uncertainty Quantification” <https://pdfs.semanticscholar.org/48a0/d3797e482e37f73e077893594e01e1c667a2.pdf>
- [2] Webpage on low discrepancy sampling methods: <http://planning.cs.uiuc.edu/node210.html>
- [3] Swiler, Laura and Slepoy, Raisa and Giunta, Anthony: “Evaluation of sampling methods in constructing response surface approximations” <https://arc.aiaa.org/doi/abs/10.2514/6.2006-1827>

Full-Factorial Sampling

The `pysmo.sampling.UniformSampling` method carries out Uniform (full-factorial) sampling. This can be done in two modes:

- The samples can be selected from a user-provided dataset, or
- The samples can be generated from a set of provided bounds.

Available Methods

```
class idaes.surrogate.pysmo.sampling.UniformSampling(data_input,  
                                                    list_of_samples_per_variable,  
                                                    sampling_type=None,  
                                                    edges=None)
```

A class that performs Uniform Sampling. Depending on the settings, the algorithm either returns samples from an input dataset which have been selected using Euclidean distance minimization after the uniform samples have been generated, or returns samples from a supplied data range.

Full-factorial samples are based on dividing the space of each variable randomly and then generating all possible variable combinations.

- The number of points to be sampled per variable needs to be specified in a list.

To use: call class with inputs, and then `sample_points` function

Example:

```
# To select 50 samples on a (10 x 5) grid in a 2D space:  
>>> b = rbf.UniformSampling(data, [10, 5], sampling_type="selection")  
>>> samples = b.sample_points()
```

`__init__` (*data_input*, *list_of_samples_per_variable*, *sampling_type=None*, *edges=None*)
Initialization of UniformSampling class. Three inputs are required.

Parameters

- **data_input** (*NumPy Array*, *Pandas Dataframe* or *list*) – The input data set or range to be sampled.
 - When the aim is to select a set of samples from an existing dataset, the dataset must be a NumPy Array or a Pandas Dataframe and **sampling_type** option must be set to “selection”. The output variable (Y) is assumed to be supplied in the last column.
 - When the aim is to generate a set of samples from a data range, the dataset must be a list containing two lists of equal lengths which contain the variable bounds and **sampling_type** option must be set to “creation”. It is assumed that no range contains no output variable information in this case.
- **list_of_samples_per_variable** (*list*) – The list containing the number of subdivisions for each variable. Each dimension (variable) must be represented by a positive integer variable greater than 1.
- **sampling_type** (*str*) – Option which determines whether the algorithm selects samples from an existing dataset (“selection”) or attempts to generate sample from a supplied range (“creation”). Default is “creation”.

Keyword Arguments **edges** (*bool*) – Boolean variable representing how the points should be selected. A value of True (default) indicates the points should be equally spaced edge to edge, otherwise they will be in the centres of the bins filling the unit cube

Returns `self` function containing the input information

Raises

- **ValueError** – The **data_input** is the wrong type
- **ValueError** – When **list_of_samples_per_variable** is of the wrong length, is not a list or contains elements other than integers
- **Exception** – When **edges** entry is not Boolean

sample_points()

`sample_points` generates or selects full-factorial designs from an input dataset or data range.

Returns A numpy array or Pandas dataframe containing the sample points generated or selected by full-factorial sampling.

Return type NumPy Array or Pandas Dataframe

References

[1] Loeven et al paper titled “A Probabilistic Radial Basis Function Approach for Uncertainty Quantification” <https://pdfs.semanticscholar.org/48a0/d3797e482e37f73e077893594e01e1c667a2.pdf>

Halton Sampling

Halton sampling is a low-discrepancy sampling method. It is a deterministic sampling method based on the Halton sequence, a sequence constructed by a set of co-prime bases. The Halton sequence is an n -dimensional extension of the Van der Corput sequence; each individual Halton sequence is based on a radix inverse function defined on a prime number.

The `pysmo.sampling.HaltonSampling` method carries out Halton sampling. This can be done in two modes:

- The samples can be selected from a user-provided dataset, or
- The samples can be generated from a set of provided bounds.

The Halton sampling method is only available for low-dimensional problems $n \leq 10$. At higher dimensions, the performance of the sampling method has been shown to degrade.

Available Methods

```
class idaes.surrogate.pysmo.sampling.HaltonSampling(data_input, num-  
                                                    ber_of_samples=None, sam-  
                                                    pling_type=None)
```

A class that performs Halton Sampling.

Halton samples are based on the reversing/flipping the base conversion of numbers using primes.

To generate n samples in a p -dimensional space, the first p prime numbers are used to generate the samples.

Note: Use of this method is limited to use in low-dimensionality problems (less than 10 variables). At higher dimensions, the performance of the sampling method has been shown to degrade.

To use: call class with inputs, and then `sample_points` function.

Example:

```
# For the first 10 Halton samples in a 2-D space:
>>> b = rbf.HaltonSampling(data, 10, sampling_type="selection")
>>> samples = b.sample_points()
```

```
__init__(data_input, number_of_samples=None, sampling_type=None)
```

Initialization of **HaltonSampling** class. Two inputs are required.

Parameters

- **data_input** (*NumPy Array, Pandas Dataframe or list*) – The input data set or range to be sampled.
 - When the aim is to select a set of samples from an existing dataset, the dataset must be a NumPy Array or a Pandas Dataframe and **sampling_type** option must be set to “selection”. The output variable (Y) is assumed to be supplied in the last column.
 - When the aim is to generate a set of samples from a data range, the dataset must be a list containing two lists of equal lengths which contain the variable bounds and **sampling_type** option must be set to “creation”. It is assumed that no range contains no output variable information in this case.
- **number_of_samples** (*int*) – The number of samples to be generated. Should be a positive integer less than or equal to the number of entries (rows) in **data_input**.
- **sampling_type** (*str*) – Option which determines whether the algorithm selects samples from an existing dataset (“selection”) or attempts to generate sample from a supplied range (“creation”). Default is “creation”.

Returns **self** function containing the input information.

Raises

- **ValueError** – The **data_input** is the wrong type.
- **Exception** – When the **number_of_samples** is invalid (not an integer, too large, zero or negative.)

sample_points()

The `sample_points` method generates the Halton samples. The steps followed here are:

1. Determine the number of features in the input data.
2. Generate the list of primes to be considered by calling `prime_number_generator` from the sampling superclass.
3. Create the first **number_of_samples** elements of the Halton sequence for each prime.
4. Create the Halton samples by combining the corresponding elements of the Halton sequences for each prime.
5. When in “selection” mode, determine the closest corresponding point in the input dataset using Euclidean distance minimization. This is done by calling the `nearest_neighbours` method in the sampling superclass.

Returns A numpy array or Pandas dataframe containing **number_of_samples** Halton sample points.

Return type NumPy Array or Pandas Dataframe

References

- [1] Loeven et al paper titled “A Probabilistic Radial Basis Function Approach for Uncertainty Quantification” <https://pdfs.semanticscholar.org/48a0/d3797e482e37f73e077893594e01e1c667a2.pdf>
- [2] Webpage on low discrepancy sampling methods: <http://planning.cs.uiuc.edu/node210.html>

Hammersley Sampling

Hammersley sampling is a low-discrepancy sampling method based on the Hammersley sequence. The Hammersley sequence is the same as the Halton sequence except in the first dimension where points are located equidistant from each other.

The `pysmo.sampling.HammersleySampling` method carries out Hammersley sampling. This can be done in two modes:

- The samples can be selected from a user-provided dataset, or
- The samples can be generated from a set of provided bounds.

The Hammersley sampling method is only available for low-dimensional problems $n \leq 10$. At higher dimensions, the performance of the sampling method has been shown to degrade.

Available Methods

class `idaes.surrogate.pysmo.sampling.HammersleySampling` (*data_input*, *number_of_samples=None*, *sampling_type=None*)

A class that performs Hammersley Sampling.

Hammersley samples are generated in a similar way to Halton samples - based on the reversing/flipping the base conversion of numbers using primes.

To generate n samples in a p -dimensional space, the first $(p - 1)$ prime numbers are used to generate the samples. The first dimension is obtained by uniformly dividing the region into **no_samples points**.

Note: Use of this method is limited to use in low-dimensionality problems (less than 10 variables). At higher dimensionalities, the performance of the sampling method has been shown to degrade.

To use: call class with inputs, and then `sample_points` function.

Example:

```
# For the first 10 Hammersley samples in a 2-D space:
>>> b = rbf.HammersleySampling(data, 10, sampling_type="selection")
>>> samples = b.sample_points()
```

__init__ (*data_input*, *number_of_samples=None*, *sampling_type=None*)

Initialization of **HammersleySampling** class. Two inputs are required.

Parameters

- **data_input** (*NumPy Array, Pandas Dataframe or list*) – The input data set or range to be sampled.
 - When the aim is to select a set of samples from an existing dataset, the dataset must be a NumPy Array or a Pandas Dataframe and **sampling_type** option must be set to “selection”. The output variable (Y) is assumed to be supplied in the last column.
 - When the aim is to generate a set of samples from a data range, the dataset must be a list containing two lists of equal lengths which contain the variable bounds and **sampling_type** option must be set to “creation”. It is assumed that no range contains no output variable information in this case.

- **number_of_samples** (*int*) – The number of samples to be generated. Should be a positive integer less than or equal to the number of entries (rows) in **data_input**.
- **sampling_type** (*str*) – Option which determines whether the algorithm selects samples from an existing dataset (“selection”) or attempts to generate sample from a supplied range (“creation”). Default is “creation”.
- **Returns** – **self** function containing the input information.
- **Raises** – **ValueError**: When **data_input** is the wrong type.

Exception: When the **number_of_samples** is invalid (not an integer, too large, zero, negative)

sample_points()

The **sampling_type** method generates the Hammersley sample points. The steps followed here are:

1. Determine the number of features n_f in the input data.
2. Generate the list of $(n_f - 1)$ primes to be considered by calling `prime_number_generator`.
3. Divide the space $[0, \text{number_of_samples} - 1]$ into **number_of_samples** places to obtain the first dimension for the Hammersley sequence.
4. For the other $(n_f - 1)$ dimensions, create first **number_of_samples** elements of the Hammersley sequence for each of the $(n_f - 1)$ primes.
5. Create the Hammersley samples by combining the corresponding elements of the Hammersley sequences created in steps 3 and 4
6. When in “selection” mode, determine the closest corresponding point in the input dataset using Euclidean distance minimization. This is done by calling the `nearest_neighbours` method in the `sampling` superclass.

Returns A numpy array or Pandas dataframe containing **number_of_samples** Hammersley sample points.

Return type NumPy Array or Pandas Dataframe

References

[1] Loeven et al paper titled “A Probabilistic Radial Basis Function Approach for Uncertainty Quantification” <https://pdfs.semanticscholar.org/48a0/d3797e482e37f73e077893594e01e1c667a2.pdf>

[2] Webpage on low discrepancy sampling methods: <http://planning.cs.uiuc.edu/node210.html>

[3] Holger Dammertz’s webpage titled “Hammersley Points on the Hemisphere” which discusses Hammersley point set generation in two dimensional spaces, http://holger.dammertz.org/stuff/notes_HammersleyOnHemisphere.html

Centroidal voronoi tessellation (CVT) sampling

In CVT, the generating point of each Voronoi cell coincides with its center of mass; CVT sampling locates the design samples at the centroids of each Voronoi cell in the input space. CVT sampling is a geometric, space-filling sampling method which is similar to k-means clustering in its simplest form.

The `pysmo.sampling.CVTSampling` method carries out CVT sampling. This can be done in two modes:

- The samples can be selected from a user-provided dataset, or
- The samples can be generated from a set of provided bounds.

The CVT sampling algorithm implemented here is based on McQueen’s method which involves a series of random sampling and averaging steps, see <http://kmh-lanl.hansonhub.com/uncertainty/meetings/gunz03vgr.pdf>.

Available Methods

```
class idaes.surrogate.pysmo.sampling.CVTSampling(data_input, num-  
                                              ber_of_samples=None, toler-  
                                              ance=None, sampling_type=None)
```

A class that constructs Centroidal Voronoi Tessellation (CVT) samples.

CVT sampling is based on the generation of samples in which the generators of the Voronoi tessellations and the mass centroids coincide.

To use: call class with inputs, and then `sample_points` function.

Example:

```
# For the first 10 CVT samples in a 2-D space:  
>>> b = rbf.CVTSampling(data_bounds, 10, tolerance = 1e-5, sampling_type="creation"  
→ )  
>>> samples = b.sample_points()
```

```
__init__(data_input, number_of_samples=None, tolerance=None, sampling_type=None)
```

Initialization of CVTSampling class. Two inputs are required, while an optional option to control the solution accuracy may be specified.

Parameters

- **data_input** (*NumPy Array, Pandas Dataframe or list*) – The input data set or range to be sampled.
 - When the aim is to select a set of samples from an existing dataset, the dataset must be a NumPy Array or a Pandas Dataframe and **sampling_type** option must be set to “selection”. The output variable (Y) is assumed to be supplied in the last column.
 - When the aim is to generate a set of samples from a data range, the dataset must be a list containing two lists of equal lengths which contain the variable bounds and **sampling_type** option must be set to “creation”. It is assumed that no range contains no output variable information in this case.
- **number_of_samples** (*int*) – The number of samples to be generated. Should be a positive integer less than or equal to the number of entries (rows) in **data_input**.
- **sampling_type** (*str*) – Option which determines whether the algorithm selects samples from an existing dataset (“selection”) or attempts to generate sample from a supplied range (“creation”). Default is “creation”.

Keyword Arguments tolerance (*float*) – Maximum allowable Euclidean distance between centres from consecutive iterations of the algorithm. Termination condition for algorithm.

- The smaller the value of tolerance, the better the solution but the longer the algorithm requires to converge. Default value is 10^{-7} .

Returns `self` function containing the input information.

Raises

- **ValueError** – When **data_input** is the wrong type.

- **Exception** – When the **number_of_samples** is invalid (not an integer, too large, zero, negative)
- **Exception** – When the tolerance specified is too loose (tolerance > 0.1) or invalid
- **warnings.warn** – when the tolerance specified by the user is too tight (tolerance < 10^{-9})

sample_points()

The `sample_points` method determines the best/optimal centre points (centroids) for a data set based on the minimization of the total distance between points and centres.

Procedure based on McQueen’s algorithm: iteratively minimize distance, and re-position centroids. Centre re-calculation done as the mean of each data cluster around each centre.

Returns A numpy array or Pandas dataframe containing the final **number_of_samples** centroids obtained by the CVT algorithm.

Return type NumPy Array or Pandas Dataframe

References

[1] Loeven et al paper titled “A Probabilistic Radial Basis Function Approach for Uncertainty Quantification” <https://pdfs.semanticscholar.org/48a0/d3797e482e37f73e077893594e01e1c667a2.pdf>

[2] Centroidal Voronoi Tessellations: Applications and Algorithms by Qiang Du, Vance Faber, and Max Gunzburger <https://doi.org/10.1137/S0036144599352836>

[3] D. G. Loyola, M. Pedernana, S. G. García, “Smart sampling and incremental function learning for very large high dimensional data” <https://www.sciencedirect.com/science/article/pii/S0893608015001768?via%3Dihub>

More Information about PySMO’s Sampling Methods

The sampling methods are able to generate samples based from variable bounds or select samples from a user-provided dataset. To use any of the method, the class is first initialized with the required parameters, and then the `sample_points` method is called.

Examples

The following code snippet shows basic usage of the package for generating samples from a set of bounds:

```
# Required imports
>>> from idaes.surrogates.pysmo import sampling as sp

# Declaration of lower and upper bounds of 3D space to be sampled
>>> bounds = [[0, 0, 0], [1.2, 0.1, 1]]

# Initialize the Halton sampling method and generate 10 samples
>>> space_init = sp.HaltonSampling(bounds_list, sampling_type='creation', number_of_
↳ samples=10)
>>> samples = space_init.sample_points()
```

The following code snippet shows basic usage of the package for selecting sample points from an existing dataset:

```
# Required imports
>>> from idaes.surrogates.pysmo import sampling as sp
>>> import pandas as pd

# Load dataset from a csv file
>>> xy_data = pd.read_csv('data.csv', header=None, index_col=0)

# Initialize the CVT sampling method and generate 25 samples
>>> space_init = sp.CVTSampling(xy_data, sampling_type='selection', number_of_
↳ samples=25)
>>> samples = space_init.sample_points()
```

Note: The results of the sampling process will be a Numpy array or Pandas dataframe, depending on the format of the input data.

Characteristics of sampling methods available in PySMO

Table 4: Characteristics of the different sampling methods

	Determinis- tic	Stochastic	Low-discrepancy	Space-filling	Geometric
LHS		✓			✓
Full-factorial	✓				✓
Halton	✓		✓		
Hammersley	✓		✓		
CVT	✓			✓	✓

Further information about the sampling tools and their input options may be found by accessing the individual sampling methods. Examples and details of the characteristics of the sampling methods may be found at [More Information about PySMO's Sampling Methods](#).



ALAMOp, RIPE, and HELMET are data driven machine learning (ddm-learning) tools. They are regression tools for the development of property models for kinetics and thermodynamics of a system. The provided tools include both ALAMOp and RIPE that can access ALAMO and other solvers through the Python API.



Python-based Surrogate Modeling Objects (PySMO) is a framework for general-purpose surrogate modeling techniques, integrated with the Pyomo mathematical optimization framework (on which IDAES is also based).

MatOpt: Nanomaterials Optimization

The MatOpt module provides tools for nanomaterials design using Mathematical Optimization. MatOpt can be used to design crystalline nanostructured materials, including but not limited to particles, wires, surfaces, and periodic bulk structures.

The main goals of this package are as follows:

- To automate many of the steps that are necessary for utilizing mathematical optimization to design materials, speeding up the development of new mathematical models and accelerating new materials discovery.
- To simplify the representation of nanostructured materials and their structure-function relationships as Pyomo objects, streamlining the creation of materials optimization problems in the Pyomo modeling language.
- To provide a simple interface so that users need not handle the details of casting efficient mathematical optimization models, invoking mathematical optimization solvers, or utilizing specialized Pyomo syntax to do this.

Thank you for your interest in MatOpt. We would love to hear your feedback! Please report any thoughts, questions or bugs to: gounaris@cmu.edu

If you are using MatOpt, please consider citing:

- Hanselman, C.L., Yin, X., Miller, D.C. and Gounaris, C.E., 2021. [MatOpt: A Python package for nanomaterials discrete optimization.](#)

Basic Usage

There are two main sub-modules contained in the package serving two distinct purposes:

- The `matopt.materials` module contains objects and methods for efficiently representing and manipulating a nanomaterial and its design space.
- The `matopt.opt` module contains objects and methods for speeding up the casting of a Mixed-integer Linear Programming (MILP) model with simplified modeling syntax and automatic model formulation.

Dependencies

User access to the MILP solver CPLEX through Pyomo is assumed. For users who do not have access to CPLEX, the use of [NEOS-CPLEX](#) is suggested as an alternative.

Define design canvas

Several pieces of information about the material and design space need to be specified in order to formulate a materials optimization problem. To fulfill this need, the `matopt.materials` module defines generic and simple objects for describing the type of material to be designed and its design space, also referred to as a “canvas”.

Some key objects are listed as follows:

class `idaes.apps.matopt.materials.lattices.lattice.Lattice`

A class used to represent crystal lattice locations.

The class encodes methods for determining which Cartesian coordinates to consider as sites on an infinite crystal lattice. A `Lattice` can be constructed from a point on the lattice (i.e., a shift from the origin), an alignment (i.e., rotation from a nominal orientation), and appropriate scaling factors. With these attributes, we generally support the translation, rotation, and rescaling of lattices. Additionally, `Lattice` objects include a method for determining which sites should be considered neighbors.

class `idaes.apps.matopt.materials.canvas.Canvas` (*Points=None, NeighborhoodIndexes=None, DefaultNN=0*)

A class for combining geometric points and neighbors.

This class contains a list of Cartesian points coupled with a graph of nodes for sites and arcs for bonds. A `Canvas` object establishes a mapping from the abstract, mathematical modeling of materials as graphs to the geometry of the material lattice. The list of points and neighbor connections necessary to create a `Canvas` object can be obtained from the combination of `Lattice`, `Shape`, and `Tiling` objects.

class `idaes.apps.matopt.materials.design.Design` (*Canvas_=None, Contents=None*)

A class used to represent material designs.

This class combines a `Canvas` objects and a list of contents. It assigns an element (possibly `None`) to each point in the `Canvas`. This generally works for any type of content, but it is intended to work with `Atom` objects and can be used to generate CFG, PDB, POSCAR, and XYZ files.

Build model via descriptors

The material type and design space specified provide indices, sets, and parameters for the optimization model. Using simple syntax, inspired by materials-related terminology, `MatOpt` users define a `MatOptModel` object, which will be translated into a Pyomo `ConcreteModel` object automatically.

`MatOpt` uses `MaterialDescriptor` objects to represent variables, constraints, and objectives. A `MatOptModel` object holds lists of `MaterialDescriptor` objects. By default, several universal site descriptors are pre-defined in the model.

Descriptor	Explanation
Y_{ik}	Presence of a building block of type k at site i
Y_i	Presence of any type of building block at site i
X_{ijkl}	Presence of a building block of type k at site i and a building block of type l at site j
X_{ij}	Presence of any building block at site i and any building block at site j
C_{ikl}	Count of neighbors of type l next to a building block of type k at site i
C_i	Count of any type of neighbors next to a building block at site i

User-specified descriptors are defined by `DescriptorRule` objects in conjunction with `Expr` expression objects. Available expressions include:

Expression	Explanation
LinearExpr	Multiplication and addition of coefficients to distinct descriptors
SiteCombination	Summation of site contributions from two sites
SumNeighborSites	Summation of site contributions from all neighboring sites
SumNeighborBonds	Summation of bond contributions to all neighboring sites
SumSites	Summation across sites
SumBonds	Summation across bonds
SumSiteTypes	Summation across site types
SumBondTypes	Summation across bond types
SumSitesAndTypes	Summation across sites and site types
SumBondsAndTypes	Summation across bonds and bond types
SumConfs	Summation across conformation types
SumSitesAndConfs	Summation across sites and conformation types

Several types of `DescriptorRules` are available.

Rule	Explanation
LessThan	Descriptor less than or equal to an expression
EqualTo	Descriptor equal to an expression
GreaterThan	Descriptor greater than or equal to an expression
FixedTo	Descriptor fixed to a scalar value
PiecewiseLinear	Descriptor equal to the evaluation of a piecewise linear function
Implies	Indicator descriptor that imposes other constraints if equal to 1
NegImplies	Indicator descriptor that imposes other constraints if equal to 0
ImpliesSiteCombination	Indicator bond-indexed descriptor that imposes constraints on the two sites
ImpliesNeighbors	Indicator site-indexed descriptor that imposes constraints on neighboring sites

From the combination of the above pre-defined descriptors, expressions, and rules, a user can specify a wide variety of other descriptors, as necessary.

```
class idaes.apps.matopt.opt.mat_modeling.MaterialDescriptor(name, canv=None,
                                                            atoms=None,
                                                            confDs=None,
                                                            bounds=(None,
                                                            None), integer=False,
                                                            binary=False,
                                                            rules=[], **kwargs)
```

A class to represent material geometric and energetic descriptors.

This class holds the information to define mathematical optimization variables for the properties of materials. Additionally, each descriptor has a ‘rules’ list to which the user can append rules defining the descriptor and constraining the design space.

name

A unique (otherwise Pyomo will complain) name

Type string

canv

The canvas that the descriptor will be indexed over

Type Canvas

atoms

The building blocks to index the descriptor over.

Type list<BBlock>

confDs

The designs for conformations to index over.

Type list<Design>

integer

Flag to indicate if the descriptor takes integer values.

Type bool

binary

Flag to indicate if the descriptor takes boolean values.

Type bool

rules

List of rules to define and constrain the material descriptor design space.

Type list<DescriptorRules>

bounds

If tuple, the lower and upper bounds on the descriptor values across all indices. If dict, the bounds can be individually set for each index.

Type tuple/dict/func

See `IndexedElem` for more information on indexing. See `DescriptorRule` for information on defining descriptors.

Solve optimization model

Once the model is fully specified, the user can optimize it in light of a chosen descriptor to serve as the objective to be maximized or minimized, as appropriate. Several functions are provided for users to choose from.

class `idaes.apps.matopt.opt.mat_modeling.MatOptModel` (*canv*, *atoms=None*,
confDs=None)

A class for the specification of a materials optimization problem.

Once all the material information is specified, we use this class to specify the material design problem of interest. This class is intended to be interpretable without mathematical optimization background while the conversion to Pyomo optimization models happens automatically.

canv

The canvas of the material design space

Type Canvas

atoms

The list of building blocks to consider. Note: This list does not need to include a void-atom type. We use 'None' to represent the absence of any building block at a given site.

Type list<BBlock>

confDs

The list of conformations to consider.

Type list<Design>

maximize (*func*, ***kwargs*)

Method to maximize a target functionality of the material model.

Parameters

- **func** (`MaterialDescriptor/Expr`) – Material functionality to optimize.

- ****kwargs** – Arguments to `MatOptModel.optimize`

Returns (`Design`/`list<Design>`) Optimal designs.

Raises `pyutilib.ApplicationError` if `MatOpt` can not find usable solver (CPLEX or NEOS-CPLEX –

See `MatOptModel.optimize` method for details.

minimize (*func*, ****kwargs**)

Method to minimize a target functionality of the material model.

Parameters

- **func** (`MaterialDescriptor/Expr`) – Material functionality to optimize.
- ****kwargs** – Arguments to `MatOptModel.optimize`

Returns (`Design`/`list<Design>`) Optimal designs.

Raises `pyutilib.ApplicationError` if `MatOpt` can not find usable solver (CPLEX or NEOS-CPLEX –

See `MatOptModel.optimize` method for details.

optimize (*func*, *sense*, *nSolns=1*, *tee=True*, *disp=1*, *keepfiles=False*, *tilim=3600*, *trelim=None*, *solver='cplex'*)

Method to create and optimize the materials design problem.

This method automatically creates a new optimization model every time it is called. Then, it solves the model via Pyomo with the CPLEX solver.

If multiple solutions (called a ‘solution pool’) are desired, then the `nSolns` argument can be provided and the `populate` method will be called instead.

Parameters

- **func** (`MaterialDescriptor/Expr`) – Material functionality to optimize.
- **sense** (*int*) – flag to indicate the choice to minimize or maximize the functionality of interest. Choices: minimize/maximize (Pyomo constants 1,-1 respectively)
- **nSolns** (*int*) – Optional, number of `Design` objects to return. Default: 1 (See `MatOptModel.populate` for more information)
- **tee** (*bool*) – Optional, flag to turn on solver output. Default: True
- **disp** (*int*) – Optional, flag to control level of `MatOpt` output. Choices: 0: No `MatOpt` output (other than solver tee) 1: `MatOpt` output for outer level method 2: `MatOpt` output for solution pool & individual solns. Default: 1
- **keepfiles** (*bool*) – Optional, flag to save temporary pyomo files. Default: True
- **tilim** (*float*) – Optional, solver time limit (in seconds). Default: 3600
- **trelim** (*float*) – Optional, solver tree memory limit (in MB). Default: None (i.e., Pyomo/CPLEX default)
- **solver** (*str*) – Solver choice. Currently only `cplex` or `neos-cplex` are supported Default: `cplex`

Returns (`Design`/`list<Design>`) Optimal design or designs, depending on the number of solutions requested by argument `nSolns`.

Raises `pyutilib.ApplicationError` if `MatOpt` can not find usable solver (CPLEX or NEOS-CPLEX –

populate (*func, sense, nSolns, tee=True, disp=1, keepfiles=False, tilim=3600, trelim=None, solver='cplex'*)

Method to a pool of solutions that optimize the material model.

This method automatically creates a new optimization model every time it is called. Then, it solves the model via Pyomo with the CPLEX solver.

The populate method iteratively solves the model, interprets the solution as a Design object, creates a constraint to disallow that design, and resolves to find the next best design. We build a pool of Designs that are guaranteed to be the nSolns-best solutions in the material design space.

Parameters

- **func** (*MaterialDescriptor/Expr*) – Material functionality to optimize.
- **sense** (*int*) – flag to indicate the choice to minimize or maximize the functionality of interest. Choices: minimize/maximize (Pyomo constants 1,-1 respectively)
- **nSolns** (*int*) – Optional, number of Design objects to return. Default: 1 (See `MatOptModel.populate` for more information)
- **tee** (*bool*) – Optional, flag to turn on solver output. Default: True
- **disp** (*int*) – Optional, flag to control level of MatOpt output. Choices: 0: No MatOpt output (other than solver tee) 1: MatOpt output for outer level method 2: MatOpt output for solution pool & individual solns. Default: 1
- **keepfiles** (*bool*) – Optional, flag to save temporary pyomo files. Default: True
- **tilim** (*float*) – Optional, solver time limit (in seconds). Default: 3600
- **trelim** (*float*) – Optional, solver tree memory limit (in MB). Default: None (i.e., Pyomo/CPLEX default)
- **solver** (*str*) – Solver choice. Currently only cplex or neos-cplex are supported Default: cplex

Returns (*list<Design>*) A list of optimal Designs in order of decreasing optimality.

Raises `pyutilib.ApplicationError` if MatOpt can not find usable solver (CPLEX or NEOS-CPLEX –

MatOpt Output

The results of the optimization process will be loaded into Design objects automatically. Users can then save material design(s) into files for further analysis and visualization using suitable functions provided. MatOpt provides interfaces to several standard crystal structure file formats, including CFG, PDB, POSCAR, and XYZ.

MatOpt Examples

Several [case studies](#) are provided to illustrate the detailed usage of MatOpt. In each case, a Jupyter notebook with explanations as well as an equivalent Python script is provided.

References

- Hanselman, C.L. and Gounaris, C.E., 2016. A mathematical optimization framework for the design of nanopatterned surfaces. *AIChE Journal*, 62(9), pp.3250-3263.
- Hanselman, C.L., Alfonso, D.R., Lekse, J.W., Matranga, C., Miller, D.C. and Gounaris, C.E., 2019. A framework for optimizing oxygen vacancy formation in doped perovskites. *Computers & Chemical Engineering*, 126, pp.168-177.
- Hanselman, C.L., Zhong, W., Tran, K., Ulissi, Z.W. and Gounaris, C.E., 2019. Optimization-based design of active and stable nanostructured surfaces. *The Journal of Physical Chemistry C*, 123(48), pp.29209-29218.
- Isenberg, N.M., Taylor, M.G., Yan, Z., Hanselman, C.L., Mpourmpakis, G. and Gounaris, C.E., 2020. Identification of optimally stable nanocluster geometries via mathematical optimization and density-functional theory. *Molecular Systems Design & Engineering*.
- Yin, X., Isenberg, N.M., Hanselman, C.L., Mpourmpakis, G. and Gounaris, C.E., 2021. A mathematical optimization-based design framework for identifying stable bimetallic nanoclusters. *In preparation*.
- Hanselman, C.L., Yin, X., Miller, D.C. and Gounaris, C.E., 2021. MatOpt: A Python package for nanomaterials discrete optimization.

Caprese

Nonlinear Model Predictive Control

Nonlinear Model Predictive Control (NMPC) is control strategy in which control inputs are determined by the solution of an optimization problem every time the plant is sampled.

Optimization Problem

An explanation of the optimization problem solved in this implementation of NMPC is forthcoming.

Available Methods

Class for performing NMPC simulations of IDAES flowsheets

```
class idaes.apps.caprese.nmpc.NMPCSim(plant_model=None, plant_time_set=None, controller_model=None, controller_time_set=None, inputs_at_t0=None, measurements=None, sample_time=None, **kwargs)
```

This is a user-facing class to perform NMPC simulations with Pyomo models for both plant and controller. The user must provide the models to use for each, along with sets to treat as “time,” inputs in the plant model, and measurements in the controller model. Its functionality is primarily to ensure that these components (as defined by the names relative to the corresponding provided models) exist on both models.

Moving Horizon Estimation

Caprese is actively under development. A module for MHE is forthcoming.



Caprese is a module for simulation of IDAES flowsheets with nonlinear program (NLP)-based control and estimation strategies, namely Nonlinear Model Predictive Control (NMPC) and Moving Horizon Estimation (MHE).

Degeneracy Hunter

Degeneracy Hunter is a collection of tools for diagnostics of mathematical programs. The core ideas behind Degeneracy Hunter are explained here: <https://www.sciencedirect.com/science/article/pii/B9780444635785501304>

Degeneracy Hunter is currently included in IDAES for beta testing purposes. This page will be updated as part of the official release. If you would like to help test Degeneracy Hunter, please look at the example notebook available here: <https://github.com/IDAES/examples-pse/pull/21> Please report any thoughts, questions or bugs to: adowling@nd.edu

ALAMOPY: ALAMO Python

ALAMOPY provides a wrapper for the software ALAMO which generates algebraic surrogate models of black-box systems for which a simulator or experimental setup is available.

RIPE: Reaction Identification and Parameter Estimation

RIPE provides tools for reaction network identification. RIPE uses reactor data consisting of concentration, or conversion, values for multiple species that are obtained dynamically, or at multiple process conditions (temperatures, flow rates, working volumes) to identify probable reaction kinetics. The RIPE module also contains tools to facilitate adaptive experimental design.

HELMET: HELMholtz Energy Thermodynamics

HELMET provides a framework for regressing multiparameter equations of state that identify an equation for Helmholtz energy and multiple thermodynamic properties simultaneously.

PySMO: Python-based Surrogate Modelling Objects

PySMO provides tools for generating different types of reduced order models. It provides IDAES users with a set of surrogate modeling tools which supports flowsheeting and direct integration into an equation-oriented modeling framework. It allows users to directly integrate reduced order models with algebraic high-fidelity process models within an single IDAES flowsheet.



MatOpt: Nanomaterials Optimization

MatOpt provides tools for nanomaterials design using Mathematical Optimization. MatOpt can be used to design crystalline nanostructured materials, including but not limited to particles, wires, surfaces, and periodic bulk structures.



Caprese

Caprese is a module for the simulation of IDAES flowsheets with nonlinear program (NLP)-based control and estimation strategies, namely Nonlinear Model Predictive Control (NMPC) and Moving Horizon Estimation (MHE).



Degeneracy Hunter

Degeneracy Hunter is coming soon!

4.3 Advanced User Guide

4.3.1 Advanced User Installation

Advanced users who plan to develop their own models or tools are encouraged to install IDAES using Git and GitHub as described in this section, rather than using the instructions in the [getting started](#) section. These advanced users will greatly benefit from improved version control and code integration capabilities.

- *Git and GitHub Basics*
- *Installation with GitHub*
 - *Github Setup*
 - *Fork the Repository*
 - *Clone Your Fork*
 - *Add Upstream Remote*
 - *Create the Python Environment*
 - *Finish the Installation*
 - *Update IDAES*

Git and GitHub Basics

Git is a distributed version control system that keeps track of changes in a set of files, while GitHub is a hosting service for Git repositories that adds many other features that are useful for collaborative software development.

Both Git and GitHub are widely used and there are excellent tutorials and resources for each. See [Atlassian Github tutorials](#) , [GitHub help](#), and [Git documentation](#).

A limited reference for Git and GitHub terminology and commands is provided [here](#), users that are new to Git and GitHub are strongly encouraged to use the more detailed resources above.

Terminology and Commands

This section gives a high-level introduction to Git and GitHub terminology and commands.

More details resources include [Atlassian Github tutorials](#) , [GitHub help](#), and [Git documentation](#).

- *Terminology*
 - *Summary*
 - *Branches*
 - *Forks*
 - *Pull Requests*
- *Git Commands*

Terminology

Summary

branch A name for a series of commits. See [Branches](#).

fork Copy of a repository in GitHub. See [Forks](#).

pull request (PR) A request to compare and merge code in a GitHub repository. See [Pull Requests](#).

Branches

A *branch* is a series of commits that allows you to separate the code development from the main code. There is a good description of what Git branches are and [how they work here](#). Understanding this takes a little study, but this pays off by making Git’s behavior much less mysterious. The short, practical version is that a branch is a name for a series of commits that you want to group together, and keep separable from other series of commits. From Git’s perspective, the branch is just a name for the first commit in that series.

It is recommended that you create new branches on which to develop your work, and reserve the “main” branch for merging in work that has been completed and approved on GitHub. One way to do this is to create branches that correspond directly to issues on GitHub, and include the issue number in the branch name.

Forks

A *fork* is a copy of a repository, in the GitHub shared space (a copy of a repository from GitHub down to your local disk is called a “clone”). In this context, that means a copy of the “idaes-dev” repository from the IDAES organization (<https://github.com/IDAES/idaes-dev>) to your own user space, e.g., <https://github.com/myname/idaes-dev>). The mechanics of creating and using forks on GitHub are given [here](#).

Pull Requests

A fundamental procedure in the development lifecycle is what is called a “pull request”. Understanding what these are, and do, is important for participating fully in the software development process. First, understand that pull requests are for collaborative development (GitHub) and not part of the core revision control functionality that is offered by Git. The official GitHub description of pull requests is [here](#). However, it gets technical rather quickly, so a higher-level explanation may be helpful:

Pull requests are a mechanism that GitHub provides to look at what the code on some branch from your fork of the repository would be like if it were merged with the main branch in the main (e.g., `idaes-pse/idaes-dev`) repository. You can think of it as a staging area where the code is merged and all the tests are run, without changing the target repository. Everyone on the team can see a pull request, comment on it, and review it.

Git Commands

The Git tool has many different commands, but there are several really important ones that tend to get used as verbs in software development conversations, and therefore are good to know:

add Put a file onto the list of “things I want to commit” (see “commit”), called “staging” the file.

commit Save the changes in “staged” files into Git (since the last time you did this), along with a user-provided description of what the changes mean (called the “commit message”).

push Move local committed changes to the GitHub-hosted “remote” repository by “pushing” them across the network.

pull Update your local files with changes from the GitHub-hosted “remote” repository by “pulling” them across the network.

Note that the *push* and *pull* commands require GitHub (or some other service that can host a remote copy of the repository).

Installation with GitHub

The main IDAES GitHub repository is `idaes-pse`. This repository includes the core framework, model libraries, and integrated tools. It contains all of the release versions of IDAES and is frequently updated with new features.

The following instructions describe how to install and update the `idaes-pse` repository.

Github Setup

In order to use GitHub, you need to create a login on [GitHub](#).

Fork the Repository

You use a “fork” of a repository (or “repo” for short) to create a space where you have complete control and can make changes without directly affecting the main repository.

You should first visit the `idaes-pse` repo on Github at <https://github.com/IDAES/idaes-pse/>.

Then you should click on the fork icon in the top right and click on your username. These steps will have created your own fork of the repo with the same name under your username.

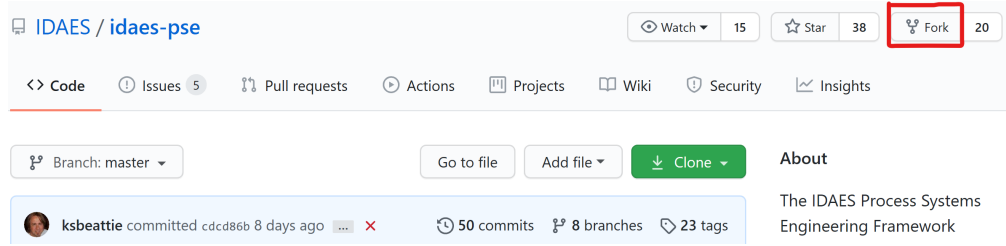


Fig. 6: Figure 1. Screenshot showing where to click to fork the Github repo

Clone Your Fork

A “clone” is a copy of a Github repository on your local machine. This is what you need to do in order to actually edit and change the files. To make a clone of the fork you created in the previous step, change to a directory where you want to put the source code and run the command:

```
git clone https://github.com/MYNAME/idaes-pse.git
cd idaes-pse
```

Of course, replace MYNAME with your username. This will download all the files in the latest version of the repository onto your local disk.

Note: After the `git clone`, subsequent Git commands should be performed from the “idaes-pse” directory.

Add Upstream Remote

In order to guarantee that your fork can be synchronized with the “main” `idaes-pse` repo in the GitHub IDAES organization, you need to add a pointer to that repository as a *remote*. This repository will be called *upstream* and linked with the following command:

```
git remote add upstream https://github.com/IDAES/idaes-pse.git
```

To check to see if you added the remote correctly use the following command:

```
git remote -v
```


You should see that there are two remotes, origin and upstream. Both have two lines showing the remote name, the url, and the access (fetch or push). Origin is the pointer to your fork and was automatically added with the clone command, while upstream is the pointer to the main idaes-pse repo that you just added.

Create the Python Environment

Once you have the repo cloned, you can change into that directory (by default, it will be called “idaes-pse” like the repo) and install the Python packages.

But before you do that, you need to get the Python package manager fully up and running. We use a Python packaging system called [Conda](#) and we specifically use its minimal version [Miniconda](#). If you do not already have Conda, please follow the installation instructions for your operating system in [getting started](#).

After Miniconda is installed, we recommend creating a separate conda environment for IDAES. If you are unfamiliar with environments, a good starting guide is [here](#). Create and activate a conda environment for the new IDAES installation with the following commands (we officially support python 3.7, but you may choose a version you prefer):

```
conda create -n idaes python=3.7
conda activate idaes
```

Note: When setting up a conda environment like this, you **must** `conda activate idaes` whenever you open a fresh terminal window and wish to use IDAES.

Finish the Installation

Now that conda and pip are installed, and you are in the “idaes” conda environment, you can run the following commands to install the requirements for IDAES and get the extensions (e.g. binaries for the solver IPOPT and other external function calls):

```
pip install .
idaes get-extensions
```

Warning: The IDAES binary extensions are not yet supported on Mac/OSX

Note: There are a few options for installing the requirements.

- 1) `pip install .` - basic install using what is in `setup.py`
- 2) `pip install .[dev]` - basic install as above and also installs the Sphinx doc tools for building the documentation locally
- 3) `pip install -r requirements.txt` - same as # 1 but installs our “development” Pyomo and PyUtilib
- 4) `pip install -r requirements-dev.txt` - same as # 3 with the Sphinx doc tools

Also note that these pip installs would override any package within the conda environment, so if you would like a specific package (e.g. git clone Pyomo), you should look at the requirements files and only install the packages you need.

You can test that everything is installed properly by running the tests with [Pytest](#):

```
pytest -m "not integration"
```

The not integration tag skips some tests that are slow. If you like, you can run all of the tests with just `pytest`.

Update IDAES

The main branch of `idaes-pse` is frequently updated and a new IDAES release occurs quarterly. It is recommended that you update your fork and local repositories and conda environment periodically.

```
pip install -U idaes-pse
pip install -U .
```

4.3.2 Developer Documentation

This section of the documentation is intended for developers, and **much of it is targeted at the IDAES internal team**. Hopefully many of the principles and ideas are also applicable to external contributors.

Developer Contents

IDAES Contributor Guide

About

This page tries to give all the essential information needed to contribute software to the IDAES project. It is designed to be useful to both internal and external collaborators.

Code and other file locations

Source code The main Python package is under the *idaes/* directory. Sub-directories, aka subpackages, should be documented elsewhere. If you add a new directory in this tree, be sure to add a *__init__.py* in that directory so Python knows it is a subpackage with Python modules. Code that is not part of the core package is under *apps/*. This code can have any layout that the creator wants.

Documentation The documentation for the core package is under *docs*.

Examples Examples are under the *examples/* directory. Tutorials from workshops are under the *examples/workshops/* subdirectory.

Developer environment

Development of IDAES will require an extra set of required package not needed by regular users. To install those extra developer tools use the command `pip install -r requirements-dev.txt` rather than `pip install -r requirements.txt`

Code style

The code style is not entirely consistent. But some general guidelines are:

- follow the [PEP8](#) style (or variants such as [Black](#))
- use [Google-style](#) docstrings on classes, methods, and functions
- format your docstrings as [reStructuredText](#) so they can be nicely rendered as HTML by Sphinx
- add logging to your code by creating and using a global log object named for the module, which can be created like: `_log = logging.getLogger(__name__)`
- take credit by adding a global author variable: `__author__ = 'yourname'`

Tests

For general information about writing tests in Python, see [Testing](#).

There are three types of tests:

Python source code The Python tests are integrated into the Python source code directories. Every package (directory with `.py` modules and an `__init__.py` file) should also have a `tests/` sub-package, in which are test files. These, by convention are named `test_<something>.py`.

Doctests With some special `reStructuredText` “directives” (see “Writing tests”), the documentation can contain tests. This is particularly useful for making sure examples in the documentation still run without errors.

Jupyter notebook tests (coming soon)

Writing tests

We use [pytest](#) to run our tests. The main advantage of this framework over the built-in `unittest` that comes with Python is that almost no boilerplate code is required. You write a function named `test_<something>()` and, inside it, use the (pytest-modified) `assert` keyword to check that things are correct.

Writing the Python unit tests in the `tests/` directory is, hopefully, quite straightforward. Here is an example (out of context) that tests a couple of things related to configuration in the core unit model library:

```
def test_config_block():
    m = ConcreteModel()

    m.u = Unit()

    assert len(m.u.config) == 2
    assert m.u.config.dynamic == useDefault
```

See the existing tests for many more examples.

For tests in the documentation, you need to wrap the test itself in a directive called `testcode`. Here is an example:

```
.. testcode::

    from pyomo.environ import *
    from pyomo.common.config import ConfigValue
    from idaes.core import ProcessBlockData, declare_process_block_class

    @declare_process_block_class("MyBlock")
```

(continues on next page)

(continued from previous page)

```
class MyBlockData(ProcessBlockData):
    CONFIG = ProcessBlockData.CONFIG()
    CONFIG.declare("xinit", ConfigValue(default=1001, domain=float))
    CONFIG.declare("yinit", ConfigValue(default=1002, domain=float))
    def build(self):
        super(MyBlockData, self).build()
        self.x = Var(initialize=self.config.xinit)
        self.y = Var(initialize=self.config.yinit)
```

First, note that `reStructuredText` directive and indented Python code. The indentation of the Python code is important. You have to write an entire program here, so all the imports are necessary (unless you use the `testsetup` and `testcleanup` directives, but honestly this isn't worth it unless you are doing a lot of tests in one file). Then you write your Python code as usual.

Running tests

Running all tests is done by, at the top directory, running the command: `pytest`.

The documentation test code will actually be run by a special hook in the `pytest` configuration that treats the `Makefile` like a special kind of test. As a result, *when you run `pytest` in any way that includes the “docs/” directory (including the all tests mode), then all the documentation tests will run, and errors/etc. will be reported through `pytest`*. A useful corollary is that, to run documentation tests, do: `pytest docs/Makefile`

You can run specific tests using the `pytest` syntax, see its documentation or `pytest -h` for details.

Documentation

The documentation is built from its sources with a tool called `Sphinx`. The sources for the documentation are:

- hand-written text files, under `docs/`, with the extension “.rst” for `reStructuredText`.
- the Python source code
- selected Jupyter Notebooks

Building documentation

Note: To build the documentation locally, you will need to have the `Sphinx` tools installed. This will be done for you by running `pip install requirements-dev.txt` (“developer” setup) as opposed to the regular `pip install requirements.txt` (“user” setup).

To build the documentation locally, use our custom `build.py` script.

```
cd docs python build.py
```

The above commands will do a completely clean build to create HTML output.

If the command succeeds, the final line will look like:

```
=== SUCCESS
```

If it fails, it will instead print something like:

```
*** ERROR in 'html'
***
*** message about the command that failed
*** and any additional info
***
```

If you want to see the commands actually being run, add `-v` to the command line.

By default the build command removes all existing built files before running the Sphinx commands. To turn this off, and rebuild only “new” things, add `-dirty` to the command line.

Previewing documentation

The generated documentation can be previewed locally by opening the generated HTML files in a web browser. The files are under the `docs/build/` directory, so you can open the file `docs/build/index.html` to get started.

Github Repository Overview

This section describes the layout of the [Github repositories](#). Later sections will give guidelines for contributing code to these repositories.

Repositories

Repository name	Public?	Description
idaes-pse	Yes	Main public repository, including core framework and integrated tools
idaes-dev	No	Main private repository, where code is contributed before being “mirrored” to the public <i>ideas-pse</i> repository
workspace	No	Repository for code that does not belong to any particular CRADA or NDA, but also is never intended to be released open-source

The URL for an IDAES repository, e.g. “some-repo”, will be `https://github.com/IDAES/some-repo`.

Public vs. Private

All these repositories except for “idaes-pse” will only be visible on Github, on the web, for people who have been added to the IDAES developer team in the IDAES “organization” (See [About Github organizations](#)). If you are a member of the IDAES team and not in the IDAES Github organization, please contact one of the core developers. The *idaes-pse* repository will be visible to anyone, even people without a Github account.

Collaborative Software Development

This page gives guidance for all developers on the project.

Note: Many details here are targeted at members of the IDAES project team. However, we strongly believe in the importance of transparency in the project's software practices and approaches. Also, understanding how we develop the software internally should be generally useful to understand the review process to expect for external contributors.

Although the main focus of this project is developing open source software (OSS), it is also true that some of the software may be developed internally or in coordination with industry under a [CRADA](#) or [NDA](#).

It is the developer's responsibility, for a given development effort, to keep in mind what role you must assume and thus which set of procedures must be followed.

CRADA/NDA If you are developing software covered by a CRADA, NDA, or other legal agreement that does not explicitly allow the data and/or code to be released as open-source under the IDAES license, then you must follow procedures under [Developing Software with Proprietary Content](#).

Internal If you are developing non-CRADA/NDA software, which is not intended to be part of the core framework or (ever) released as open-source then follow procedures under [Developing Software for Internal Use](#).

Core/open-source If you are developing software with no proprietary data or code, which is intended to be released as open-source with the core framework, then follow procedures under [Developing software for Open-source Release](#).

Developing Software with Proprietary Content

Proprietary content is not currently being kept on Github, or any other collaborative version control platform. When this changes, this section will be updated.

Developing Software for Internal Use

Software for internal use should be developed in the `workspace` repository of the IDAES github organization. The requirements for reviews and testing of this code are not as strict as for the `idaes-dev` repository, but otherwise the procedures are the same as outlined for [open-source development](#).

Developing software for Open-source Release

We can break the software development process into five distinct phases, illustrated in Figure 1 and summarized below:

1. <i>Setup</i> : Prepare your local system for collaborative development
2. <i>Initiate</i> : Notify collaborators of intent to make some changes
3. <i>Develop</i> : Make local changes
4. <i>Collaborate</i> : Push the changes to Github, get feedback and merge

The rest of this page describes the what and how of each of these phases.

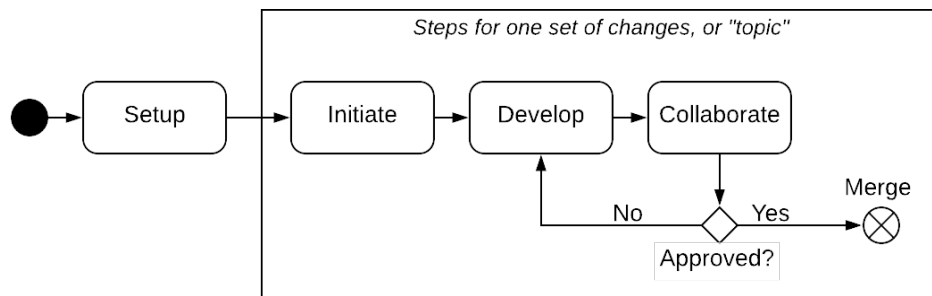


Fig. 7: Figure 1. Overview of software development workflow

1. Setup

Before you can start developing software collaboratively, you need to make sure you are set up in Github and set up your local development environment.

Github setup

To work within the project, you need to create a login on [Github](#). You also need to make sure that this login has been added to the IDAES organization by contacting one of the core developers.

If these steps are successful, you should be able to login to Github, visit the [IDAES Github organization](#), and see “Private” repositories such as *idaes-dev* and *workspace*.

Fork the repo

You use a “fork” of a repository (or “repo” for short) to create a space where you can save changes without directly affecting the main repository. Then, as we will see, you *request* that these changes be incorporated (after review).

This section assumes that the repository in question is *idaes-dev*, but the idea is the same for any other repo.

You should first visit the repo on Github by pointing your browser to <https://github.com/IDAES/idaes-dev/>. Then you should fork the repo into a repo of the same name under your name.

Clone your fork

A “clone” is a copy of a Github repository on your local machine. This is what you need to do in order to actually edit and change the files. To make a clone of the fork you created in the previous step, change to a directory where

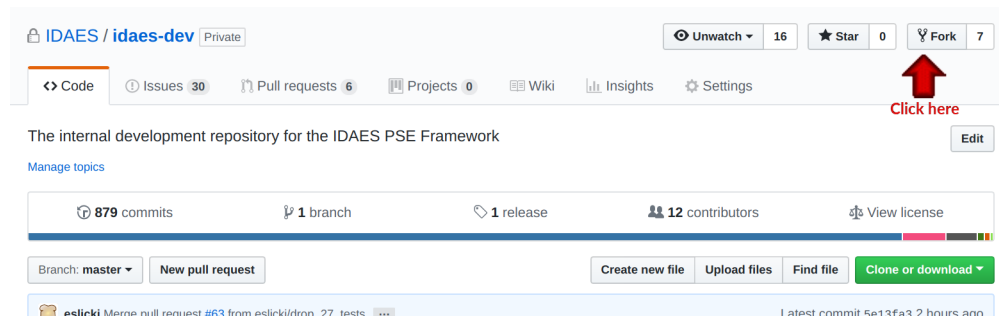


Fig. 8: Figure 2. Screenshot showing where to click to fork the Github repo

you want to put the source code and run the command:

```
git clone   
→git@github.  
→com:MYNAME/  
→idaes-dev.git  
cd idaes-dev
```

Of course, replace MYNAME with your login name. This will download all the files in the latest version of the repository onto your local disk.

Note: After the `git clone`, subsequent git commands should be performed from the “idaes-dev” directory.

Add upstream remote

In order to guarantee that your fork can be synchronized with the “main” idaes-dev repo in the Github IDAES organization, you need to add a pointer to that repository as a *remote*. This repository is called *upstream* (changes made there by the whole team flow down to your fork), so we will use that name for it in our command:

```
git remote   
→add upstream   
→git@github.  
→com:IDAES/  
→idaes-dev.git
```


Create the Python environment

Once you have the repo cloned, you can change into that directory (by default, it will be called “idaes-dev” like the repo) and install the Python packages.

But before you do that, you need to get the Python package manager fully up and running. We use a Python packaging system called [Conda](#). Below are instructions for installing a minimal version of Conda, called [Miniconda](#). The full version installs a large number of scientific analysis and visualization libraries that are not required by the IDAES framework.

```
wget_
↳https://repo.
↳anaconda.com/
↳miniconda/
↳Miniconda3-
↳latest-Linux-
↳x86_64.sh
bash_
↳Miniconda3-
↳latest-Linux-
↳x86_64.sh
```

Create and activate a conda environment (along with its own copy of pip) for the new IDAES installation (**you will need to** conda activate idaes **when you open a fresh terminal window and wish to use IDAES**):

```
conda create_  
↳ -n idaes pip  
conda activate_  
↳ idaes
```

Now that conda and pip are installed, and you are in the “idaes” conda environment, you can run the standard steps for installing a Python package in development mode:

```
pip install -r_  
↳ requirements.  
↳ txt  
python setup.  
↳ py develop
```

You can test that everything is installed properly by running the tests with [Pytest](#):

```
pytest
```

2. Initiate

We will call a set of changes that belong together, e.g. because they depend on each other to work, a “topic”. This section describes how to start work on a new topic. The workflow for initiating a topic is shown in Figure 3 below.

Create an issue on Github

To create an issue on Github, simply navigate to the repository page and click on the “Issues” tab. Then click on the “Issues” button and fill in a title and brief description of the issue. You do not need to list details about sub-steps required for the issue, as this sort of information is better put in the (related) pull request that you will create later. Assign the issue to the appropriate people, which is often yourself.

There is one more important step to take, that will allow the rest of the project to easily notice your issue: add the issue to the “Priorities” project. The screenshot below shows where you need to click to do this.

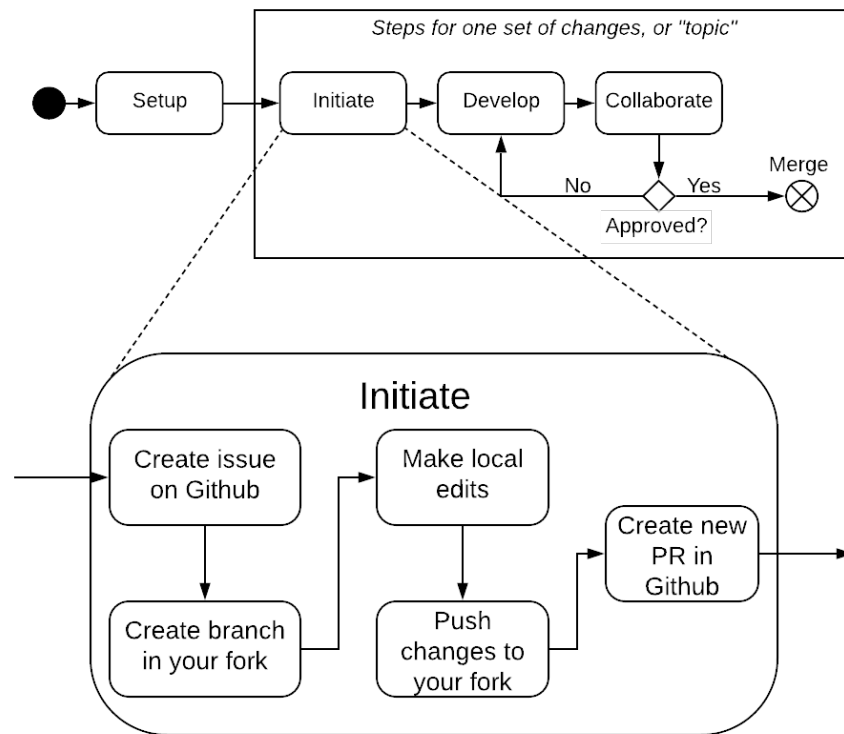


Fig. 9: Figure 3. Initiate topic workflow

Create a branch on your fork

It is certainly possible to do your work on your fork in the “main” branch. The problem that can arise here is if you need to do two unrelated things at the same time, for example working on a new feature and fixing a bug in the current code. This can be quite tricky to manage as a single set of changes, but very easy to handle by putting each new set of changes in its own branch, which we call a *topic* branch. When all the changes in the branch are done and merged, you can delete it both locally and in your fork so you don’t end up with a bunch of

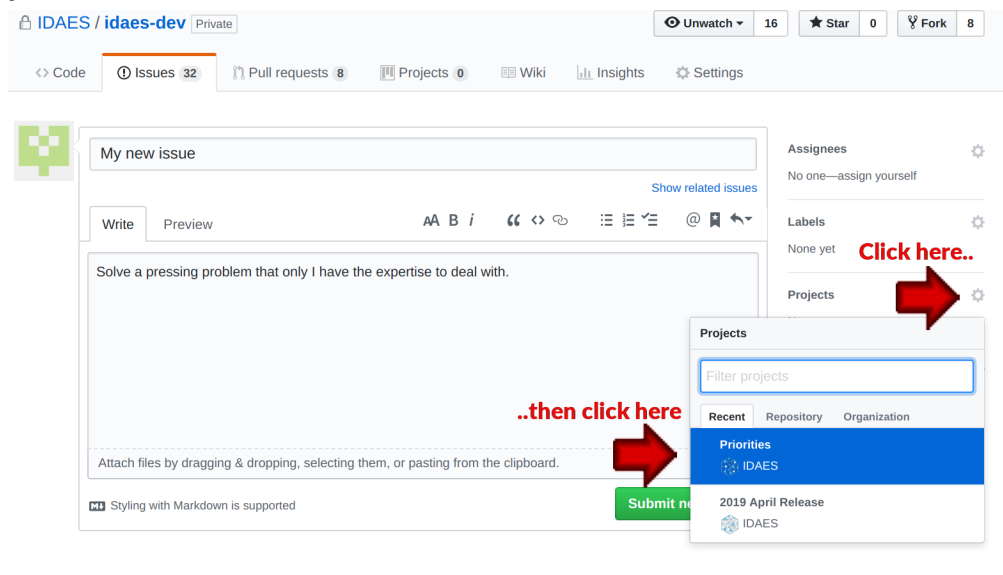


Fig. 10: Figure 4. Screenshot for creating an issue on Github

old branches cluttering up your git history.

The command for doing this is simple:

```
git_
↳ checkout -b_
↳ <BRANCH-NAME>
```

The branch name should be one word, with dashes or underscores as needed. One convention for the name that can be helpful is to include the issue number at the end, e.g. git co -b

mytopic-issue42. This is especially useful later when you are cleaning up old branches, and you can quickly see which branches are related to issues that are completed.

Make local edits and push changes

A new branch, while it feels like a change, is not really a change in the eyes of Git or Github, and by itself will not allow you to start a new pull request (which is the goal of this whole phase). The easiest thing to do is a special “empty” commit:

```
git commit -
↳ -allow-empty_
↳ -m 'Empty_
↳ commit_
↳ so I can_
↳ open a PR'
```

(continues on next page)

(continued from previous page)

Since this is your first “push” to this branch, you are going to need to set an upstream branch on the remote that should receive the changes. If this sounds complicated, it’s OK because git actually gives you cut-and-paste instructions. Just run the `git push` command with no other arguments:

```
$ git push
fatal:
↳The current
↳branch
↳mybranch-
↳issue3000
↳has
↳no upstream
↳branch.
To push
↳the current
↳branch
↳and set the
↳remote as
↳upstream, use

    git push --
↳set-upstream
↳origin
↳mybranch-
↳issue3000
```

Cut and paste the suggested command, and you’re ready to go. Subsequent calls to “push” will not require any additional arguments to work.

Start a new Pull Request on Github

Finally, you are ready to initiate the pull request. Right after you perform the `push` command above, head to the repository URL in Github (<https://github.com/IDAES/idaes-dev>) and you should see a highlighted bar below the tabs, as in Figure 5 below, asking if you want to start a pull-request.

Click on this and fill in the requested information. Remember to link to the issue you created earlier.

Depending on the Github plan, there may be a pull-down menu for creating the pull request that lets you create a “draft” pull request. If that is not present, you can signal this the old-fashioned way by adding “[WIP]” (for Work-in-Progress) at the beginning of the pull request title.

Either way, create the pull request. Do *not* assign reviewers until you are done making your changes (which is probably not now). This way the assigning of reviewers becomes an unambiguous signal that the PR is actually ready for review.

Note: Avoid having pull requests that take months to complete. It is better to divide up the work, even artificially, into a piece that can be reviewed and merged into the main repository within a week or two.

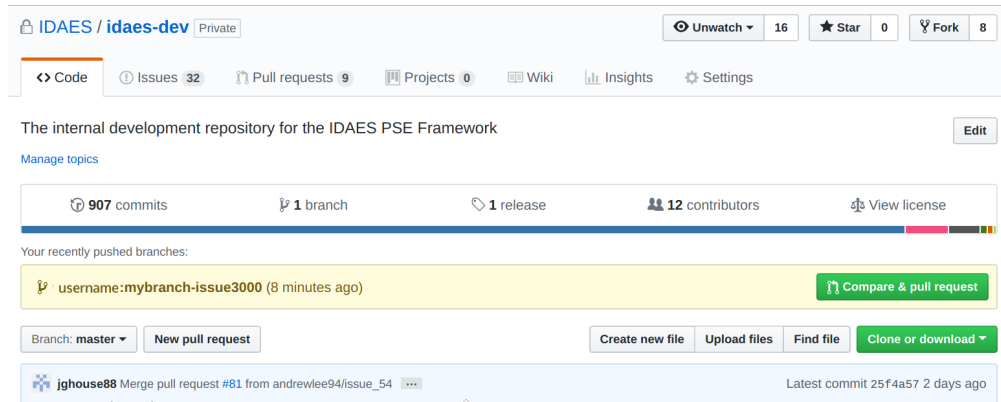


Fig. 11: Figure 5. Screenshot for starting a Pull Request on Github

3. Develop

The development process is a loop of adding code, testing and debugging, and committing and pushing to Github. You may go through many (many!) iterations of this loop before the code is ready for review. This workflow is illustrated in Figure 6.

Running tests

After significant edits, you should make sure you have tests for the new/changed functionality. This involves writing *Unit tests* as well as running the test suite and examining the results of the *Code coverage*.

This project uses *Pytest* to help with running the unit tests. From the top-level directory of the working tree, type:

```
pytest
```

Alternatively users of an IDE like PyCharm can run the tests from within the IDE.

Commit changes

The commands: `git add`, `git status`, and `git commit` are all used in combination to save a snapshot of a Git project's current state.¹

The `commit` command is the equivalent of “saving” your changes. But unlike editing a document, the set of changes may cover multiple files, including newly created files. To allow the user flexibility in specifying exactly which changes to save with each commit, the `add` command is used first to indicate files to “stage” for the next commit command. The `status` command is used to show the

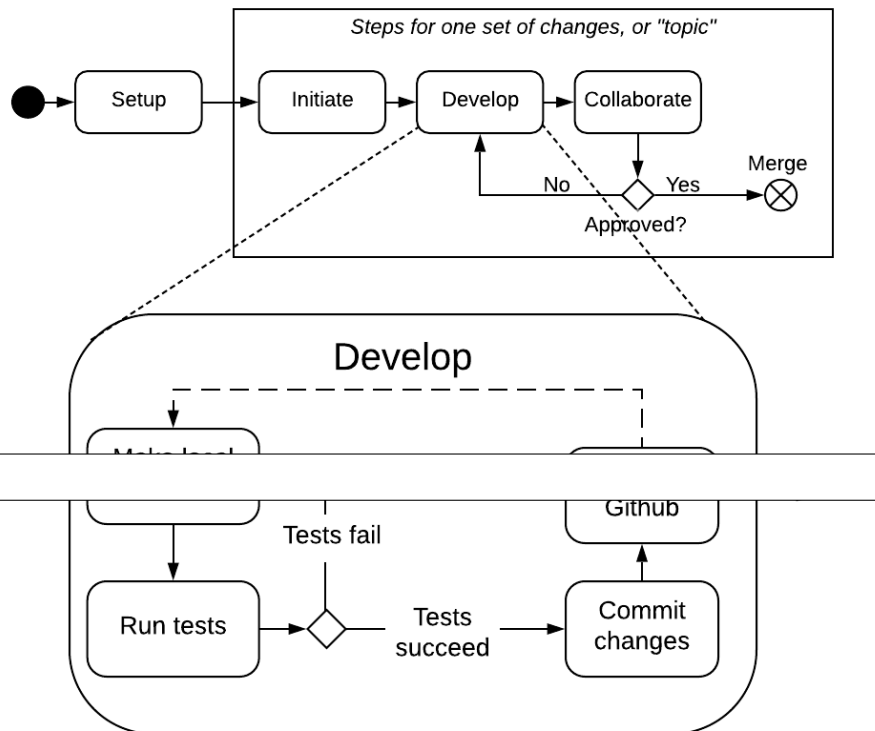


Fig. 12: Figure 6. Software development workflow

¹ Git has an additional saving mechanism called ‘the stash’. The stash is an ephemeral storage area for changes that are not ready to be committed. The stash operates on the working directory and has extensive usage options.* See the documentation for `git stash` for more information.

current status of the working tree.

A typical workflow goes like this:

```
$ ls
file1  file2
$ echo 'a' > file1 #
↪edit existing file
$ echo '1' > file3
↪# create new file
$ git status
↪--short # shows
↪changed/unstaged
↪and unknown file
M file1
?? file3
$ git
↪add file1 file3
↪# stage file1,
↪file3 for commit
$ git
↪status --short #
↪M=modified, A=added
M file1
A file3
$ git commit -m
↪"made some changes"
[main 067c16e]
↪made some changes
2 files changed,
↪2 insertions(+)
create
↪mode 100644 file3
```

Of course, in most IDEs you could use built-in commands for committing and adding files. The basic flow would be the same.

Synchronize with upstream changes

Hopefully you are not the only one on the team doing work, and therefore you should expect that the main repository may have new and changed content while you are in the process of working. To synchronize with the latest content from the “upstream” (IDAES organization) repository, you should periodically run one of the

two following commands:

```
git pull
# OR -- explicit
git fetch --all
git_
↳merge upstream/main
```

You'll notice that this merge command is using the name of the "upstream" remote that you *created earlier*.

Push changes to Github

Once changes are *tested* and committed, they need to be synchronized up to Github. This is done with the git push command, which typically takes no options (assuming you have set up your fork, etc., as described so far):

```
git push
```

The output of this command on the console should be an informative, if slightly cryptic, statement of how many changes were pushed and, at the bottom, the name of your remote fork and the local/remote branches (which should be the same). For example:

```
Counting_
↳objects: 5, done.
Delta_
↳compression using_
↳up to 8 threads.
Compressing objects:_
↳100% (5/5), done.
Writing_
↳objects: 100%_
↳(5/5), 528 bytes |_
↳528.00 KiB/s, done.
Total 5 (delta 4),
↳ reused 0 (delta 0)
remote: Resolving_
↳deltas: 100% (4/
↳4), completed with_
↳4 local objects.
```

(continues on next page)

(continued from previous page)

```

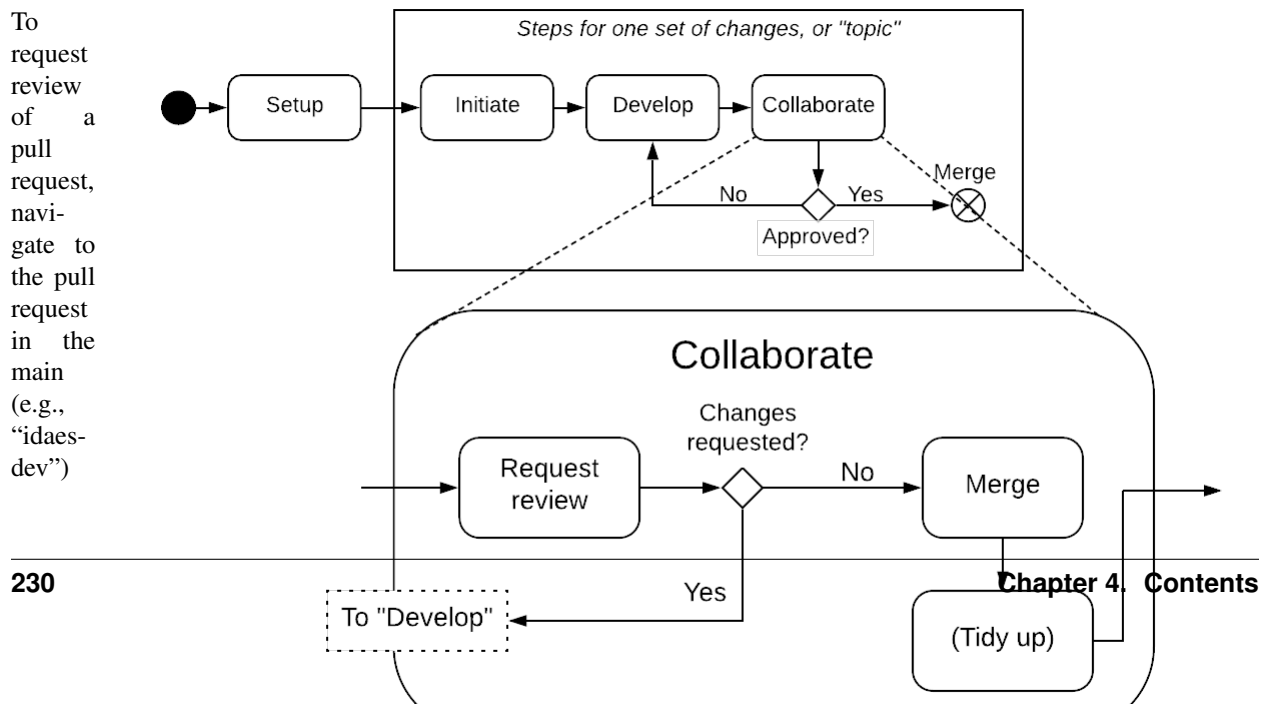
To github.
↪ com:dangunter/
↪ idaes-dev.git
   d535552..fe61fcc
↪ devdocs-issue65
↪ -> devdocs-issue65

```

4. Collaborate

The collaboration phase of our journey, shown in Figure 7, is mostly about communicating what you did to the other developers. Through the Github “review” mechanism, people will be able to suggest changes and improvements. You can make changes to the code (other people can also make changes, see [Shared forks](#)), and then push those changes up into the same Pull Request. When you get enough approving reviews, the code is merged into the main repository. At this point, you can delete the “topic branch” used for the pull request, and go back to [initiate](#) your next set of changes.

Request review



repos-
itory
and
select
some
names
in the
“Re-
view-
ers”
pull-
down
on the
right-
hand

side. You need to have two approving reviews. The reviewers should get an email, but you can also “@” people in a comment in the pull request to give them a little extra nudge.

See
the full
[code](#)
[review](#)
proce-
dure for
more
details.

Make changes

You
need
to
keep
track
of
the
com-
ments
and
re-
views,
and
make
changes
accord-
ingly.
Think
of a
pull
request
as a
discus-
sion.

Nor-
mally,
the
person
who
made
the pull request will make any requested edits. Occasionally, it may make sense for one or more other developers to jump in and make edits too, so how to do this is covered in the sub-section below.

Changes
made
while
the
code is
being
re-
viewed
use the
normal
De-
velop
work-
flow.

Shared forks

Other
de-
vel-
op-
ers
can
also
make
changes
in your
fork.
All
they
need
to do
is git
clone
your
fork
(not the
main
repos-
itory),
switch
to the
correct
topic

branch,
and
then

`git push` work directly to that branch. Note since this does not use the whole pull-request mechanism, all developers working on the same branch this way need to make sure the `git pull` to synchronize with updates from the other developers.

For
ex-
am-
ple,
if
Jack
wants
to
make
some
ed-
its
on
Rose's
fork,
on
a
topic
branch
called
"changes-
issue51"
he
could
do the
follow-
ing:

```
↪ $  
↪ git  
↪ clone  
↪ https://  
↪ /  
↪ github.  
↪ com/  
↪ rose/  
↪ idaes-  
↪ dev  
↪  
↪ #  
↪ clone  
↪ Rose  
↪ 's  
↪ fork  
  
↪ $  
↪ git  
↪ checkout  
↪ changes-  
↪ issue51
```

(continues on next page)

(continued from previous page)

```
→$  
→echo  
→  
→"Hello  
→"  
→>  
→>  
→  
→README.  
→txt  
→  
→  
→#  
→make  
→some  
→important  
→changes  
  
→$  
→pytest  
→  
→#  
→always  
→run  
→tests!  
→!  
  
→$  
→git  
→add  
→README.  
→txt  
→;  
→  
→git  
→commit  
→-  
→m  
→  
→"important  
→changes  
→"  
  
→$  
→git  
→push  
→  
→#  
→push  
→changes  
→to  
→the  
→fork
```

Hope-
fully

it also
is ob-
vious
that
devel-
opers
work-
ing
this
way
have
less
safe-
guards
for
over-
writing
each
other's
work,
and
thus
should
make
an
ef-
fort
to
communicate clearly and in a timely manner.

Merge

Once
all the
tests
pass
and
you
have
enough
ap-
proving
re-
views,
it's
time to
merge
the
code!
This
is the
easy
part: go

to the
bottom
of the
Pull
Re-
quest
and hit
the big
green
“merge” button.

Be-
fore
you
close
the
lap-
top
and
go
down
to
the
pub,
you
should
tidy
up.
First,
delete
your
lo-
cal
branch
(you
can
also
delete
that
branch
on
Github):

```
git_
↳ checkout_
↳ main_
↳
↳ #_
↳ switch_
↳ back_
↳ to_
↳ main_
↳ branch
git_
↳ branch_
```

(continues on next page)

```
↳ d_
↳ mychanges=
```


(continued from previous page)

Next,
you
should
make
sure
your
main
reflects
the
current
state of
the up-
stream
main
branch,
i.e. go
back
and
*syn-
chro-
nize
with
the up-
stream
remote,*
i.e. run
git
pull.

Now
you can
go and
enjoy
a tasty
bev-
erage.
Cheers!



Developer Standards

Contents
• <i>Developer Standards</i>
– <i>Model Formatting and General Standards</i>
* <i>Headers and Meta-data</i>
* <i>Coding Standard</i>
* <i>Model Organization</i>
* <i>Commenting</i>
– <i>Units of Measurement and Reference States</i>
– <i>Standard Variable Names</i>
– <i>Testing</i>

Model Formatting and General Standards

The section describes the recommended formatting used within the IDAES framework. Users are strongly encouraged to follow these standards in developing their models in order to improve readability of their code.

Headers and Meta-data

Model developers are encouraged to include some documentation in the header of their model files which

pro-
vides
a brief
de-
scrip-
tion
of the
purpose
of the
model
and
how
it was
developed. Some suggested information to include is:

- Model
name,
- Model
publi-
cation
date,
- Model
author
- Any
nec-
essary
licens-
ing and
dis-
claimer
infor-
mation
(see
below).
- Any
addi-
tional
infor-
mation
the
mod-
eler
feels
should
be in-
cluded.

Coding Standard

All
code
devel-
oped as
part of
IDAES
should
con-
form
to the
PEP-8
stan-
dard.

Model Organization

Whilst
the
overall
IDAES
mod-
eling
frame-
work
en-
forces
a hier-
archical
struc-
ture on
models,
model
devel-
opers
are still
encour-
aged to
arrange
their
models
in a
logical
fashion
to aid
other
users

in understanding the model. Model constraints should be grouped with similar constraints, and each grouping of constraints should be clearly commented.

For
prop-

erty
pack-
ages,
it
is
rec-
om-
mended
that
all
the
equa-
tions
nec-
es-
sary
for
cal-
cu-
lat-
ing
a
given
prop-
erty
be
grouped
to-
gether, clearly separated and identified by using comments.

Addi-
tion-
ally,
model
devel-
opers
are
encour-
aged
to con-
sider
break-
ing
their
model
up into
a num-
ber of
smaller
meth-
ods
where
this
makes

sense.

This

can fa-

cilitate

modifi-

cation

of the code by allowing future users to inherit from the base model and selectively overload sub-methods where desired.

Commenting

To

help

other

mod-

el-

ers

and

users

un-

der-

stand

the

how

a

model

works,

model

builders

are

strongly

encour-

aged to

com-

ment

their

code. It

is sug-

gested

that

every

constraint should be commented with a description of the purpose of the constraint, and if possible/necessary a reference to a source or more detailed explanation. Any deviations from standard units or formatting should be clearly identified here. Any initialization procedures, or other procedures required to get the model to converge should be clearly commented and explained where they appear in the code. Additionally, modelers are strongly encouraged to add additional comments explaining how their model works to aid others in understanding the model.

Units of Measurement and Reference States

Due to the flexibility provided by the IDAES modeling framework, there is no standard set of units of measurement or standard reference state that should be used in models.

This places the onus on the user to understand the units of measurement being used within their models and to ensure that they are consistent.

The standard units and reference states are described in the *user guide*.

Standard Variable Names

The
stan-
dard
vari-
able
names
are de-
scribed
in the
user
guide.

Testing

The
testing
stan-
dards
are in-
cluded
here.

Testing

Testing
is es-
sential
to the
process
of cre-
ating
soft-
ware.
“If it
isn’t
tested,
it
doesn’t
work”
is a
good
rule of
thumb.

*For
some
specific
advice
for
adding*

*new
tests
in the
IDAES
code,
see
IDAES
Con-
tributor
Guide.*

There
are
dif-
ferent
kinds
of
tests:
func-
tional,
accep-
tance,
perfor-
mance,
us-
ability.
We
will
pri-
marily
con-
cern
our-
selves
with
*func-
tional*
test-
ing
here,
i.e.

whether the thing being tested produces correct outputs for expected inputs, and gracefully handles everything else. Within functional testing, we can classify the testing according to the axes of *time*, i.e. how long the test takes to run, and *scope*, i.e. the amount of the total functionality being tested. Along these two axes we will pick out just two points, as depicted in Figure 1. The main tests you will write are “unit tests”, which run very quickly and test a focused amount of functionality. But sometimes you need something more involved (e.g. running solvers, using data on disk), and here we will label that kind of test “integration tests”.

Unit tests

Testing individual pieces of functionality, including the ability to report the correct kind of errors from bad inputs. Unit tests must always run quickly. If it takes more than 10 seconds, it is not a unit test, and it is expected that most unit tests take well under 1 second. The reason for this is that the entire unit test suite is run on every change in a Pull Request, and should also be run relatively frequently on local developer machines. If this suite of hundreds of tests takes more than a couple of minutes to run, it will introduce a significant bottleneck in the development workflow.

For Python code, we use the `pytest` testing framework. This is compatible with the built-in Python `unittest` framework, but has many nice features that make it easier and more powerful.

The best way to learn how to use `pytest` is to look at existing unit tests, e.g. the file `“idaes/core/tests/test_process_block.py”`. Test files are found in a directory named `“test/”` in every Python package (directory with an `“__init__.py”`). The tests are named `“test_{something}.py”`; this naming convention is important so `pytest` can automatically find all the tests.

When writing your own tests, make sure to remember to keep each test focused on a single piece of functionality. If a unit test fails, it should be obvious which code is causing the problem.

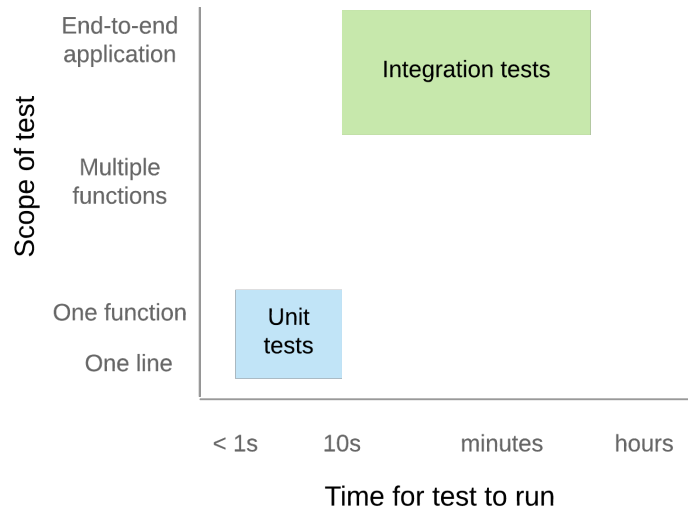


Fig. 14: Figure 1. Conceptual space of functional testing

Tagging tests

Since we use `pytest` for our testing, we have access to the very nice `pytest` “tag” feature, which uses `Python decorators` to add labels to tests.

An example of a test with a tag is (assume `import pytest` at the top of every test module):

```
@pytest.mark.unit
def test_something():
    assert 2. + 2. == 4.
```

Every test should be decorated with `@pytest.mark.<level>` where `<level>` has one of three values:

- **unit** Test runs quickly (under 2 seconds) and has no network/system dependencies. Uses only libraries installed by default with the software
- **component** Test may run more slowly (under 10 seconds, or so), e.g. it may run a solver or create a bunch of files. Like unit tests, it still shouldn’t depend on special libraries or dependencies.
- **integration** Test may take a long time to run, and may have complex dependencies.

The expectation is that unit tests should be run by developers rather frequently, component tests should be run by the continuous integration system before running code, and integration tests are run across the codebase regularly, but infrequently (e.g. daily).

Sometimes you may also want to run tests on only a particular platform. We currently support Windows, Linux, and (to a lesser extent) MacOS. To restrict a test to one or more of these platforms, typically Linux-only, use `@pytest.mark.<platform>`, or `@pytest.mark.no<platform>` where `<platform>` has one of three values:

- **linux / nolineux** Linux systems, regardless of distribution, e.g. CentOS, Ubuntu, Debian, *et al.*
- **win32 / nowin32** Windows 10
- **darwin / nodarwin** Mac OSX

As you may have guessed, the “no<platform>” version means that any operating system *except* “<platform>” will run the test. You can combine these tags as you wish, though until we have more than three options it is not necessary.

Here are a few examples:

```
@pytest.mark.unit
def test_something():
    print(
↪ "unit test, all platforms")

@pytest.mark.unit
@pytest.mark.nowin32
def test_something():
    print("unit test, all_
↪ platforms except Windows")

@pytest.mark.component
@pytest.mark.linux
def test_something():
    print("component_
↪ test, linux-only")

@pytest.mark.integration
@pytest.mark.nodarwin
def test_something():
    print(
↪ "integration test, all_
↪ platforms except MacOS")
```

Mocking

Mocking is a common, but important, technique for avoiding dependencies that make your tests slow, fragile, and harder to understand. The basic idea is to replace dependencies with fake, or “mock”, versions of them that will provide just enough realism for the test. Python provides a library, `unittest.mock`, to help with this process by providing objects that can report how they were used, and easily pretend to have certain functionality (returning, for example, fixed values). To make this all more concrete, consider a simple problem where you want to test a function that makes a system call (in this case, `os.remove`):

```
# file: mymodule.py
import os
def rm(filename):
    os.remove(filename)
```

Normally, to test this you would create a temporary file, and then see if it got removed. However, with mocking you can take a different approach entirely:

```
# file: test_mymodule.py
from mymodule import rm
from unittest import mock

@mock.patch('mymodule.os')
def test_rm(mock_os):
    rm("any path")
    # test_
    ↳ that rm called os.remove_
    ↳ with the right parameters
    mock_os.remove.assert_
    ↳ called_with("any path")
```

Here, we have “patched” the `os` module that got imported into “mymodule” (note: had to do `mymodule.os` instead of simply `os`, or the one mymodule uses would not get patched) so that when `rm` calls `os.remove`, it is really calling a fake method in `mock_os` that does nothing but record how it was called. The patched module is passed in to the test as an argument so you can examine it. So, now, you are not doing any OS operations at all! You can imagine how this is very useful with large files or external services.

Integration tests

Integration tests exercise an end-to-end slice of the overall functionality. At this time, the integration tests are all housed in Jupyter Notebooks, which serve double-duty as examples and tutorials for end users. We execute these notebooks and verify that they run correctly to completion at least once before each new release of the software.

Code coverage

The “coverage” of the code refers to what percentage of the code (“lines covered” divided by total lines) is executed by the automated tests. This is important because passing automated tests is only meaningful if the automated tests cover the majority of the code’s behavior. This is not a perfect measure, of course, since simply executing a line of code under one condition does not mean it would execute correctly under all conditions. The code coverage is evaluated locally and then integrated with Github through a tool called [Coveralls](#).

Code Review

“It’s a simple 3-step process. Step one: Fix! Step two: It! Step three: Fix it!” – Oscar Rogers (Kenan Thompson), Saturday Night Live, 2/2009

Code review is the last line of defense between a mistake that the IDAES team will see and a mistake the whole world will see. In the case of that mistake being a leak of proprietary information, the entire project is jeopardized, so we need to take this process seriously.

Summary

Warning: This section is an incomplete set of notes

Every piece of code must be reviewed by at least two people.

In every case, one of those people will be a designated “gatekeeper” and the one or more others will be “technical reviewers”.

The technical reviewers are expected to consider various aspects of the proposed changes (details below), and engage the author in a discussion on any aspects that are deemed lacking or missing.

The gatekeeper is expected to make sure all criteria have been met, and actually merge the PR.

Assigning Roles

The gatekeeper is a designated person, who will always be added to review a Pull Request (PR)

Gatekeeper is a role that will be one (?) person for some period like a week or two weeks

The role should rotate around the team, it's expected to be a fair amount of work and should be aligned with availability and paper deadlines, etc.

The originator of the PR will add as reviewers the gatekeeper and 1+ technical reviewers.

Originator responsibilities

The originator of the PR should include in the PR itself information about where to find:

Changes to code/data

Tests of the changes

Documentation of the changes

The originator should be responsive to the reviewers

Technical reviewer responsibilities

The technical reviewer(s) should look at the proposed changes for

Technical correctness (runs properly, good style, internal code documentation, etc.)

Tests

Documentation

No proprietary / sensitive information

Until they approve, the conversation in the PR is between the technical reviewers and the originator (the gatekeeper is not required to participate, assuming they have many PRs to worry about)

Gatekeeper responsibilities

The gatekeeper does not need to engage until there is at least one approving technical review.

Once there is, they should verify that:

- Changes do not contain proprietary data
- Tests are adequate and do not fail
- Documentation is adequate

Once everything is verified, the gatekeeper merges the PR

Automated Checks

The first level of code review is a set of automated checks that *must* pass before the code is ready for people to review it. These checks will run on the initiation of a *pull request* and on every new commit to that pull request that is pushed to Github (thus the name “continuous integration”).

The “continuous integration” of the code is hosted by an online service – we use [CircleCI](#) – that can automatically re-run the tests after every change (in this case, every new Pull Request or update to the code in an existing Pull Request) and report the results back to Github for display in the web pages. This status information can then be used as an automatic gatekeeper on whether the code can be merged into the main branch – if tests fail, then no merge is allowed. Following this procedure, it is not possible for the main branch to ever be failing its own tests.

Docker Container

This page documents information needed by developers for working with the IDAES docker container.

As is expected by Docker, the main file for creating the Docker image is the “Dockerfile” in the top-level directory.

docker-idaes script

You can build new Docker images using the `create` option to the *docker-idaes* script. For example:

```
./docker-idaes create
```

You need to have the IDAES installation activated. The script will automatically find the current version and attempt to build a Docker image with the same version. If it detects an existing image, it will skip the image build. Next, the script will try to use `docker save` to save the image as a compressed archive. This will also be skipped if an existing image file, with the same version as the “idaes” Python package, is detected.

Pushing an image to S3

The Docker images are stored on Amazon S3. Before you can upload a new image, you need to be part of the “IDAES-admin” group that is part of Amazon’s IAM (Identity Access Management) system. Please contact one of the core developers to learn how to join this IAM group.

Once you have the IAM keys, you need to create a file `~/.aws/credentials` that has the access key id and key from the IAM account. It will look like this:

```
[default]
aws_
↪access_key_id = IDGOESHERE
aws_secret_access_
↪key = accesskeygoeshere
```

The values for the ID and Access key are available from the AWS “IAM” service console.

Next you need to use the AWS command-line tools to copy the local image up to Amazon S3. For example, if the image was version “1.0.1”, you would use the following command:

```
aws s3 cp idaes-
↪pse-docker-1.0.1.tgz \
    s3://idaes/idaes-pse/
↪idaes-pse-docker-1.0.1.tgz
```

If the new image should be the latest, you also need to do an S3 -> S3 copy to create a new latest image:

```
aws s3 cp
↪s3://idaes/idaes-pse/idaes-
↪pse-docker-1.0.1.tgz \
    ↪
↪    s3://idaes/idaes-pse/
↪idaes-pse-docker-latest.tgz
```

Glossary

API Acronym for “Application Programming Interface”, this is the set of functions used by an external program to invoke the functionality of a library or application. For IDAES, it usually refers to Python functions and classes/methods in a Python module. By analogy, the APIs are to the IDAES library what a steering wheel, gearshift and pedals are to a car.

CRADA Cooperative Research and Development Agreement. A legal agreement between two or more parties that involves a statement of work and terms for sharing non-public data.

NDA Non-Disclosure Agreement. A legal agreement between two or more parties that involves terms for sharing non-public data.

4.3.3 Developing Custom Models

It is difficult to build a single model library that will suit all modeling needs, and thus users will inevitably encounter situations where they need to create new models to represent their processes. The IDAES Process Modeling Framework has been developed with this in mind, and all components of the framework have been designed to be fully accessible and modifiable. This section of the

documentation will explain how users can develop new models, or modify existing models, for use within the IDAES modeling environment.

- *Creating New Modeling Components*
- *Defining New Model Classes*
- *Inheriting from Existing Models*
- *Config Blocks*
- *The build Method*
- *Types of Models*

Creating New Modeling Components

All models within the IDAES Process Modeling Framework, be they models of unit operations or thermodynamic properties, are constructed in the same way and user defined models follow the same structure. Each model component is a set of instructions on how to assemble a Pyomo ‘Block’ containing the necessary variables, expressions and constraints to describe the desired process. These instructions are contained within Python *classes* which can be written and modified by the user.

Details on what is required when constructing custom models of different types will be provided in subsequent sections of this documentation, however there are some steps common to all types of models which will be discussed here.

Defining New Model Classes

There is a significant amount of work that has to be done behind the scenes in order to create a new model component and integrate it into the IDAES modeling framework. Rather than force the user to understand and implement this themselves, IDAES provides a number of standard tools which users can use to automate this when creating new models. These tools are used when declaring new components, and should be

used in most cases when a user is creating a new model (exceptions are mentioned in the detailed documentation for each type of model).

An example of declaring new model (named `NewModel`) is shown below:

```
@declare_process_
↪block_class("NewModel")
class_
↪NewModelData(BaseClass):
```

The first part of the declaration is the `declare_process_block_class` decorator, which automates all the code required to create a new type of Pyomo *Block*. This decorator needs to be provided with a name for the new model (`NewModel`). Understanding the details of class decoration within Python and the function of the `declare_process_block_class` decorator are not necessary for developing new models, however users who wish to read more can see the [technical specifications](#).

The second part of the declaration creates a `NewModelData` class which inherits from an existing `BaseClass`. The `NewModelData` class needs to contain the instructions necessary for building the desired model and must be populated by the user. In practice, each type of model (unit operation, thermophysical properties, etc.) has a set of common tasks which appear in most models of that type. To assist users with developing new model and reduce the need for them to rewrite these common tasks, IDAES provides a set of base classes which contain many of these common instructions. Model developers can use these base classes as a foundation for their new models using what is referred to as “inheritance”, as shown in the example above. In doing so, the new model class automatically gains access to all of the common instructions in the base class which can be used in constructing the new model.

Inheriting from Existing Models

In addition to inheriting from the IDAES base classes, it is also possible to inherit from existing models, which provides an easy way to build off an existing model instead of starting afresh. When inheriting from an existing model, you gain access to all the instructions written for constructing that model and can then add to or modify those instructions as needed. All of the models in the core IDAES model library were written with this in mind, and were designed to provide a core model representing the simplest representation of each piece of process equipment possible to allow users to easily build upon these as a foundation.

A useful concept when modifying existing models through inheritance is “overloading” of methods. Any method defined by the inherited class can be overloaded and replaced by a new method of the same name defined in the new class. Thus, it is possible to selectively modify and replace parts of the existing model if they were defined using methods. For example, suppose there is an existing model that meets most of a user’s needs, but the user would like to use a different equation for efficiency. If the existing model defined a method specifically for writing the efficiency constraint, then this can be replaced by inheriting the existing model and writing a new method for efficiency with the desired equation. This will overload the method in the original model, creating a new model which uses the desired equation. This requires little effort on the part of the user, but does require the original model to use modular methods for each performance equation however.

Config Blocks

Whilst the model class contains the instructions necessary to build a model object, it is often necessary to provide additional information when creating an instance of a model. One example of this is informing a unit model of which property package to use for a given instance. When creating a new model class, it is necessary to define the information that a user may pass to the class when creating an instance of the new model, which is done using configuration blocks (config blocks for short) – this is where the information in the “default” keyword is sent when an instance of a model is created.

Configuration blocks are defined by declaring a *CONFIG* object for each new model data class, as shown in the example below. The *CONFIG* object should be an instance of a Pyomo *ConfigBlock*.

```
@declare_process_
↳ block_class("NewModel")
class_
↳ NewModelData(BaseClass):

CONFIG = BaseClass.CONFIG()
```

Each type of model has a set of expected inputs (or arguments) which are determined by the type of model and can be inherited from the appropriate base class (as shown above). Users may also add custom configuration arguments to their models as needed by declaring new entries to the *CONFIG* block as shown below:

```
from pyomo.common.
↳ config import ConfigValue

@declare_process_
↳ block_class("NewModel")
class_
↳ NewModelData(BaseClass):
    _
↳ CONFIG = BaseClass.CONFIG()
    CONFIG.declare("new_
↳ argument", ConfigValue(
        default = #_
↳ default value for argument,
```

(continues on next page)

(continued from previous page)

```
↪  
↪     domain = # condition  
↪input must satisfy,  
    description = "short  
↪description of argument",  
    doc = "longer  
↪description of argument"))
```

Note: Configuration arguments are set when an instance of a model is created and are generally only used at build-time. That is, once a model has been constructed changing a configuration argument has no effect on the model structure.

The *build* Method

Finally, the core of any IDAES model class is the *build* method, which contains the set of instructions to be executed when a model is created. The *build* method acts as the rule for constructing the resulting Pyomo *Block*, and needs to contain the instructions necessary for constructing the variable, expressions and constraints which describe the model. The *build* method is written in Python code and should construct the necessary Pyomo components, and may make use of sub-methods to modularize the model construction.

In almost all cases, the first instruction in a *build* method should be to call the *build* method of the inherited (base) class. This is necessary to execute the instructions in the base class, and can be done with the following line of code:

```
super().build()
```

Types of Models

Custom Unit Models

- *UnitModelBlockData*
- *Unit Model Configuration*
- *Unit Model build Method*
- *Control Volumes*

- *Unit Model Initialization*
- *Unit Model Report*
- *Tutorials*

UnitModelBlockData

The starting point for all unit models within the IDAES Process Modeling Framework is the *UnitModelBlockData* base class. This class contains a number of methods to assist users with creating new unit models including:

- checking and validating the “dynamic” and “has_holdup” configuration arguments to ensure consistency,
- adding *Port* objects to the model,
- creating simple material balances between states (equal flow of each component in each phase), and,
- a method for initializing simple unit models.

More details on the *UnitModelBlockData* class can be found in the [technical specifications](#).

Unit Model Configuration

Configuration arguments in unit models allow model developers to provide the end-user with the ability to configure the model to suit the needs of the flowsheet they are simulating. The most common aspects that need to be configured are:

- whether the model should be dynamic or steady-state, and
- what property package to use when calculating thermophysical properties.

The *UnitModelBlockData* class contains a simple configuration block which includes two configuration arguments; “dynamic” and “has_holdup”. These arguments are required for any model which is expected to be used in both steady-state and dynamic flowsheets and are used to determine

whether accumulation and holdup terms should be constructed and included in the material balance equations. There are some situations where a model is inherently steady-state (even if it is included in a dynamic flowsheet), notably those where outlet conditions are a function solely of the inlet conditions. Examples of these include:

- unit operations involving equilibrium where the outlet condition can be calculated directly from the inlet condition.
- unit operations with negligible holdup where (total) flow out of the unit is always equal to the (total) flow in.

In general, a unit model is not written with a specific flowsheet or set of thermophysical property calculations in mind, thus it is necessary to provide a configuration argument (or arguments in cases where multiple streams interact) to allow the end-user to specify a property package to use with the model. The example below shows how to declare a configuration argument for a single property package, along with a second argument that allows users to pass configuration arguments to the instances of the property packages when they are created.

```
@declare_process_  
↳ block_class("NewUnit")  
class_  
↳ NewUnitDataData(UnitModelBlockData):  
  
    CONFIG =_  
↳ UnitModelBlockData.CONFIG()  
  
    CONFIG.declare("property_  
↳ package", ConfigValue(  
        default=useDefault,  
        domain=is_  
↳ physical_parameter_block,  
        description=  
↳ "Property package_  
↳ to use for control volume",  
  
        doc="""Property parameter_  
↳ object used to define_  
↳ property calculations.""")  
    CONFIG.declare("property_  
↳ package_args", ConfigBlock(  
        implicit=True,
```

(continues on next page)

(continued from previous page)

```

    ↪ description="Arguments
    ↪to use for constructing
    ↪property packages",
        doc="""A
    ↪ConfigBlock with arguments
    ↪to be passed to a property
    ↪block(s) and used when
    ↪constructing these."""))

```

For unit models involving multiple property packages, or those that include reaction packages, additional pairs of configuration arguments are required for each of these. Model developers must provide unique names for each configuration argument, and are encouraged to use meaningful names to assist end-users in understanding what package should be linked to each argument.

Model developers may also declare additional configuration arguments to give end-users the ability to change the behavior of different parts of the model. For example, the core IDAES Unit Model Library makes use of these to provide flexibility in the form of the balance equations. Use of additional configuration arguments is entirely optional.

Unit Model *build* Method

The *build* method for a unit model must include all the instructions necessary for constructing the representation of the unit operation. This generally involves the following steps:

1. Calling *super().build()* to trigger the behind-the-scenes code.
2. Adding any variables and constraints required to describe the system geometry.
3. Adding *State Blocks* to the model to represent each of the material states in the system.
4. Adding the necessary material balances and associated variable to describe the flow of material between each state.
5. Adding the necessary energy balances

and associated variable to describe the flow of energy between each state.

6. Adding the necessary momentum balances and associated variable to describe the flow of momentum between each state.
7. Adding any additional performance equations and associated variable that govern the behavior of the unit operation.
8. Adding the required inlet and outlet Ports to allow the unit model to be included in a flowsheet.

For some applications, not all of these steps will be required (e.g. a process in which pressure drop is negligible may be able to skip adding momentum balances).

The above steps represent a significant amount of work, and in many cases require a detailed understanding of how the IDAES framework is structured. To reduce the effort and knowledge required to create new models, the framework provides a number of tools to automate these steps for common cases. Users are encouraged to familiarize themselves with the methods available in *UnitModelBlockData* and the use of control volumes.

Control Volumes

The IDAES Process Modeling Framework includes tools to assist users with creating new models in the form of the Control Volume libraries. These libraries contain methods for performing the common task associated with building unit models, such as creating material, energy and momentum balances. Users are free to choose whether or not to use these libraries, but are encouraged to understand what is available in these as they can greatly reduce the amount of effort required by the user.

The IDAES Process Modeling Framework currently includes two types of Control Volumes:

1. *ControlVolume0D* for inlet-outlet type models where spatial variation are not significant.
2. *ControlVolume1D* for models where spatial variation in one-dimension are required.

Unit Model Initialization

Whilst the *UnitModelBlockData* class contains a pre-built *initialize* method, this method is relatively simple and is unlikely to work for more complex models. For these situations, model developers will need to write their own *initialize* methods as part of their new unit model.

To create a custom initialization routine, model developers must create an *initialize* method as part of their model, and provide a sequence of steps intended to build up a feasible solution. Developing initialization routines is one of the hardest aspects of model development, and generally involves starting with a simplified form of the model and progressively adding complexity. Initialization routines generally make use of Pyomo's tools for activating and deactivating constraints and often involve solving multiple sub-problems whilst building up an initial state.

The example below shows the general form used when declaring a new initialization method:

```
def initialize(blk,
    ↪ state_args=None,
    ↪ outlvl=idaeslog.NOTSET,
        solver='ipopt
    ↪ ', optarg={'tol': 1e-6}):
```

- blk – local name for the block to be initialized.
- state_args – initial guesses for the state variables. The form of this may vary depending on the number and type of inlets to the unit model.
- outlvl – optional argument to allow users to control the amount of diagnostic output from the initialization process.

dure. His requires the use of the IDAES logger tools to function.

- `solver` – allows the user to set a solver to use for initialization.
- `optarg` – dict of options to pass to the solver; used to adjust solver behavior.

Unit Model Report

Users are likely already aware of the *report* method which is available in all IDAES models and prints a summary of the current state of a given model. This functionality is also part of *UnitModelBlockData* and is thus included in all custom unit models, however model developers need to define what information should be included in the output.

The *report* method will automatically search for and identify all *Ports* in the model to be included in the summary stream table, however model developers must identify any performance variables they wish to include in the summary. This is done by declaring a `_get_performance_contents` method as shown in the example below:

```
def _get_  
↳ performance_contents(self,  
↳ time_point=0):  
    var_dict = {"display name"  
↳ ": self.var[time_point]}  
  
    ↳  
    ↳ expr_dict = {"display name"  
↳ ": self.expr[time_point]}  
  
    ↳  
    ↳ param_dict = {"display name"  
↳ ": self.param[time_point]}  
  
    return {"vars"  
↳ ": var_dict, "exprs": expr_  
↳ dict, "params": param_dict}
```

The `_get_performance_contents` method should take two arguments, the first being the model object and the second being a time point at which to report the model state. The method should return a dictionary-of-dictionaries with one to three keys; “vars”, “exprs” and “params”. The entries from these will be included in the

model summary under the headings of Variables, Expressions and Parameters respectively.

Tutorials

Tutorials demonstrating how to create custom unit models are found [here](#).

Custom Property Packages

- *Property Package Classes*
- *Build-on-Demand Properties*
- *The Physical Parameter Block*
- *Physical Parameter Block Configuration*
- *The Physical Parameter Block build Method*
- *Defining Property Metadata*
- * *Setting Default Units*
- * *Setting Properties Metadata*
- *The State Block*
- *The State Block Data class*
- * *State Block Data Configuration Arguments*
- * *The State Block build Method*
- * *State Variables and Properties*
- * *Required Methods*
- *The State Block Methods class*
- * *Declaration and Base Class*
- * *The initialize and release_state Methods*
- *Tutorials*

Warning: This section is currently being developed

Physical property packages form the core of all IDAES process models, and the ability for users to develop their

own property formulations is a key aspect of the IDAES modeling paradigm. In order to support the flexibility of the IDAES Process Modeling Framework, property packages define a number of key aspects that inform the eventual structure of the final process model. This however places the burden of making these decisions on the developer of each property package, which are implemented as part of the property package classes.

Property Package Classes

As users of the IDAES Process Modeling Framework, you are likely already aware that property packages (of both types) consist of two related model components; in this case the Physical Parameter Block and the State Block. However, when creating a new thermo-physical property package, developers need to define three (rather than two) new classes.

The first of these classes is a *PhysicalParameterBlock* class, which is responsible for constructing the Physical Parameter Block. However, two classes are required for defining the State Block; a *StateBlockMethods* class and a *StateBlockData* class. The reason for this is because State Blocks are always indexed (by time and occasionally by space) . The *StateBlockData* class represents an individual state at a point in space and time (i.e. one element of the indexed *StateBlock*), and as such contains a set of state variables and the constraints necessary for calculating the desired thermophysical properties at that state. However, we often want to perform actions on the entire set of states (i.e. *StateBlockDatas*) in one go, such as during initialization. Whilst we could initialize each state individually, as the process for each state is generally identical except for values, it is much more efficient to perform the same set of instructions on all state simultaneously. The *StateBlockMethods* serves this purpose by defining the methods that should be applied to multiple *StateBlockDatas* simultaneously. When a model requires a State Block, these two classes are combined to produce the final model. The distinction and use of the *StateBlockMethods* and *StateBlockData* classes will be discussed further later in this documentation.

Build-on-Demand Properties

Before moving onto a discussion of the contents of each of the three classes, it is important to introduce the concept of build-on-demand properties. Property packages generally tend to include methods for a number of properties, but not all of these will be required by every unit model. In order to reduce model complexity and avoid calculating properties which are not required for a given unit operation, the IDAES framework supports the concept of build-on-demand properties, where the variables and constraints related to a given property are only constructed if called for in a given state.

It must be noted that this is an advanced feature and is entirely optional. Whilst it can reduce the complexity of individual models, it also increases the complexity of the model instructions and can increase the chance of errors during model constructions. Property package developers should decide up front if they wish to implement build-on-demand properties for their property packages, and which properties this will be implemented for (i.e. it is possible to use the build-on-demand infrastructure) for a subset of the properties within a package.

The Physical Parameter Block

The first part of the physical property package is the *PhysicalParameterBlock*, which defines the global parameters and components of the property package. This includes:

- a reference to the *StateBlock* class associated with the *PhysicalParameterBlock*,
- the chemical species or components in the material,
- the thermodynamic phases of interest,
- the base units of measurement for the property packages,

- the thermophysical properties supported by the property package, and
- the parameters required to calculate the thermophysical properties.

The starting point for creating a new *PhysicalParameterBlock* is shown in the example below. The model developer needs to declare a new class which inherits from the *PhysicalParameterBlock* base class, decorated using the *declare_process_block_class* decorator.

```
@declare_process_block_class(
    ↪ "NewPhysicalParameterBlock"
    ↪ ")
class_
    ↪ NewPhysicalParameterData(PhysicalParameterBlock):

    def build(self):
        super().build()

    @classmethod
    def_
    ↪ define_metadata(cls, obj):
        obj.add_
    ↪ properties({# properties})
        obj.add_
    ↪ default_units({# units})
```

The *NewPhysicalParameterData* class needs to contain a *build* method, and may also include a configuration block and a *define_metadata* classmethod as shown above. These methods and their contents will be explained below.

Physical Parameter Block Configuration

Like all IDAES models, Physical Parameter Blocks can have configuration arguments which can be used to adjust the form of the resulting model. The default configuration block which comes from the *PhysicalParameterBlock* base class contains a single configuration argument:

- “default_arguments” - this configuration argument allows users to specify a set of default configuration arguments that will be passed to all *StateBlocks* created from an instance of a parameter block.

The Physical Parameter Block *build* Method

The *build* method in the *NewPhysicalParameterBlock* class is responsible for constructing the various modeling components that will be required by the associated *StateBlocks*, such as the sets components and phases that make up the material, and the various parameters required by the property calculations. The *build* method is also responsible for setting up the underlying infrastructure of the property package and making a link to the associated *StateBlock* class so that the modeling framework can automate the construction and linking of these.

The first step in the *build* method is to call *super().build()* to trigger the construction of the underlying infrastructure using the base class' *build* method.

Next, the user must declare an attribute named “_state_block_class” which is a pointer to the associated *StateBlock* class (creation of this will be discussed later). An example of this is shown below, where the associated State Block is named *NewStateBlock*.

```
def build(self):

    super().build()
    self._state_
    ↪block_class = NewStateBlock
```

The next step in the *build* method is to define the chemical species and phases necessary to describe the material of interest. This is done by adding *Component* and *Phase* objects, as shown below.

```
def build(self):

    ↪
    ↪ self.benzene = Component()
    ↪
    ↪ self.toluene = Component()

    ↪
    ↪ self.liquid = LiquidPhase()
    self.vapor = VaporPhase()
```

Note: The IDAES Process Modeling Framework supports a number of different types of *Component* and *Phases* ob-

jects, as discussed in the associated documentation. Users should use the type most appropriate for their applications. Also note that whilst *Component* and *Phase* objects contain configuration arguments, these are primarily for use by the Generic Property Package framework, and are not required for custom property packages.

Finally, the *build* method needs to declare all the global parameters that will be used by the property calculations. By declaring these in a single central location rather than in each State Block, this reduces the number of parameters present in the model (thus reducing memory requirements) and also facilitates parameter estimation studies using these parameters.

Note: Whilst we generally use the term “parameters” to describe these global coefficients used in property correlations, it is often better to declare these as Pyomo *Var* objects with fixed values (rather than as *Param* objects). The reason for this is because, despite the name, it is not possible to estimate the value of *Params* using parameter estimation tools (as their value is concrete and cannot be changed).

Defining Property Metadata

The last part of creating a new Physical Parameter block is to define the metadata associated with it. The properties metadata serves three purposes:

1. The default units metadata is used by the framework to automatically determine the units of measurement of the resulting property model, and automatically convert between different unit sets where appropriate.
2. The properties metadata is used to set up any build-on-demand properties,
3. The metadata is also used by the Data Management Framework to index the available property packages to create a searchable index for users.

Setting Default Units

The most important part of defining the metadata for a property package is to set the default units of measurement for each of the 7 base quantities (time, length, mass, amount, temperature, current (optional) and luminous intensity (optional)). These units are used by the modeling framework to determine the units of measurement for all other quantities in the process that are related to this property package. Units must be defined using Pyomo *Units* components, as shown in the example below:

```
from_
↳ pyomo.environ import units

@classmethod
def_
↳ define_metadata(cls, obj):
    obj.add_default_
↳ units({'time': units.s,

↳         'length': units.m,

↳         'mass': units.kg,

↳         'amount': units.mol,

↳         'temperature': units.K})
```

Setting Properties Metadata

The primary purpose of the properties metadata is to set up the build-on-demand system used to selectively construct only those properties required by a given unit operation. In order to do this, the user needs to add each property they wish to build-on-demand along with the name of a method that will be called whenever the property is required (this method will be created later as part of the *StateBlockData* class). Users are also encouraged to list *all* properties supported by their property packages here, setting *None* as the method associated with the property for those which are always constructed. An example for both uses is shown below:

```
@classmethod
def
↳define_metadata(cls, obj):
    obj.add_properties({
        'property_1': {
↳'method': method_name}, #
↳a build-on-demand property

        'property_2': {'method':
↳None}}) # a property that
↳will always be constructed
```

Note: The name of a property in the metadata dictionary must match the name of the property component (normally a variable) that will be called for. These names should be drawn from the *standard naming conventions*.

The State Block

The second part of a thermophysical property package is the *StateBlock* class, which as mentioned earlier is defined using two user-written classes; the *StateBlockData* class and the *StateBlockMethods* class. Declaration of the *StateBlock* class is similar to that of other modeling classes, but makes use of a special aspect of the *declare_process_block_class* decorator as shown in the example below.

```
@declare_process_block_
↳class("NewStateBlock",

    block_
↳class=NewStateBlockMethods)
class
↳NewStateBlockData(StateBlockData):

    def build(self):
        super().build()
```

As can be seen, the declaration of the new *StateBlock* class (*NewStateBlock*) looks similar to that of other modeling class declarations, where the *declare_process_block_class* is applied to a user defined *NewStateBlockData* class. However, in this case we also provide an additional argument to the decorator; the “block_class” argument allows us to attach a set of methods declared in a user-defined class (in this case *NewStateBlockMethods*) to the

NewStateBlock class, which can be applied across all members of an indexed *NewStateBlock* (methods in the *NewStateBlockData* class can only be applied to a single indexed element).

The State Block Data class

As part of the core of the IDAES Process Modeling Framework, the *StateBlockData* class is responsible not only for defining the variables, expressions and constraints which describe the thermophysical properties of the material in question, but also providing information to the rest of the Process Modeling Framework on how the higher levels models should be formulated. As such, *StateBlockData* classes need to define more methods than any other component class. The base class for developing new *StateBlockData* classes is *StateBlockData*, which includes a configuration block with a number of critical configuration arguments as well as the code necessary for supporting “build-on-demand properties”.

State Block Data Configuration Arguments

The *StateBlockData* base class configuration contains three configuration arguments that are expected by the modeling framework and must be included in and user defined *StateBlockData*. These configuration arguments are:

- “parameters” – this argument is used to provide a link back to the associated *PhysicalParameterBlock*, and is generally automatically passed to the *StateBlock* when it is constructed.
- “defined_state” – this argument is used to indicate whether this state represents a point in the process where all state variables are defined. The most common case for this is for inlets to unit models, where all inlet states are known from the outlet of the previous unit model. In these cases, it is not possible to write certain constraints, such as the sum of mole fractions, without over

specifying the system of equations; this argument identifies these cases so that generation of these constraints can be automatically skipped.

- “has_phase_equilibrium” – this argument indicates whether phase equilibrium will be considered for this state. Phase equilibrium constraints decrease the degrees of freedom in the system thus it is important to determine when and where these constraints should be written. Note that equilibrium constraints can never be written for cases where the state is fully defined (as above), thus both this and the *defined_state* arguments must be considered when determining whether to include equilibrium constraints.

The State Block *build* Method

As with all IDAES components, the *build* method forms the core of a *StateBlockData* class, and contains the instructions on how to construct the variables, expressions and constraints required by the thermophysical model. As usual, the first step in the *build* method should be to call *super().build()* to trigger the construction of the underlying components required for State Blocks to function.

State Variables and Properties

The most important part of the construction of a State Block is defining the necessary set of variables, expression and constraints that make up the property model. There are many different ways in which these can be defined and formulated, and there is no single “best” way to do this; different approaches may work better for different applications. However, there are some general rules that should be followed when defining the variables which make up a State Block.

1. All state variables and properties should use the IDAES naming conventions. Standard names allow linking between

different types of models to be automated, as no cross-referencing of names is required.

2. All properties within a property package should use a consistent set of base units. This is most easily accomplished by selecting a set of units for the 7 base SI quantities (time, length, mass, amount, temperature, current and luminous intensity) and deriving units for all quantities from these. Modelers should also select units based solely on convenience or ease of use – scaling of variables and equations is better handled separately using the *IDAES scaling tools*.

Beyond these requirements, modelers are free to choose the form of their model to best suit their needs and make the most tractable problem possible. Modelers are also free to combine variable and constraints with expression for some quantities as needed. The IDAES Process Modeling Framework is concerned only that the expected quantities are present (i.e. the expected variable/expression names), not their exact form or how they are calculated.

As described throughout this page, IDAES supports “build-on-demand” for property correlations. Details on how to define methods for building properties on demand is demonstrated in the tutorials (see link at bottom of page).

Required Methods

As the foundation of the entire Process Modeling Framework, the definition of a new *StateBlockData* class needs to include a number of methods that the framework relies on for determining the formulation of the higher level models.

Below is a list of the required methods, along with a short description.

- *get_material_flow_basis(block)* – this method is used to define the basis on which material balance terms will be expressed. This is used by the framework to automatically convert between

mass and mole basis if required, and the method needs to return a *MaterialFlowBasis Enum*.

- *get_material_flow_terms(block, phase, component)* – this method is used to determine the form of the material flow terms that are constructed as part of the material balance equations in each unit model. This method needs to take three arguments; a reference to the current state block, a phase name and a component name, and must return an expression for the material flow term for the given phase and component.
- *get_material_density_terms(block, phase, component)* – similar to the *get_material_flow_terms* method, this method is used to determine the form of the density term which should be used when constructing material holdup terms in the material balances. This method also needs to take three arguments; a reference to the current state block, a phase name and a component name, and must return an expression for the material density term for the given phase and component.
- *get_material_diffusion_terms(block, phase, component)* – Support for this is not currently implemented.
- *get_enthalpy_flow_terms(block, phase)* – this method is used to determine the form of the enthalpy flow terms that are constructed as part of the energy balance equations in each unit model. This method needs to take two arguments; a reference to the current state block and a phase name, and must return an expression for the enthalpy flow term for the given phase and component.
- *get_energy_density_terms(block, phase)* – this method is used to determine the form of the energy density terms that are required for the holdup terms in the energy balance equations. This method needs to take two arguments; a reference to the current state block and a phase name, and must return an expression for the energy density term for the given phase and component. Note that the

holdup/density term needs to be in terms of internal energy, not enthalpy.

- *get_energy_diffusion_terms(block, phase)* – Support for this is not currently implemented.
- *default_material_balance_type(block)* – this method is used to set a default for the type of material balance to be written by a Control Volume if the user does not specify which type to use. This method needs to return a *MaterialBalanceType Enum*.
- *default_energy_balance_type(block)* – this method is used to set a default for the type of energy balance to be written by a Control Volume if the user does not specify which type to use. This method needs to return a *EnergyBalanceType Enum*.
- *define_state_vars(block)* – this method is used to define the set of state variables which should be considered the state variables for the property package, and is used in a number of methods associated with model initialization to determine which variables should be fixed. This method must return a Python dict, where the keys are the variable name as a string, and the values are the variables.
- *define_port_members(block)* – similar to the *define_state_vars* method, this method is used to define what variables should be part of the inlet/outlet ports of a unit model. In many cases, these variables are equivalent to the state variables of the property package and if so this method can be skipped (if undefined *define_state_vars* is called instead). This method is similar to the one in the above method, however in this case the key names can be defined by the user for improved readability (instead of having to be the variable name).
- *define_display_vars(block)* – similar again to the *define_state_vars* method, this method is used to define a set of variables which should be used when generating the output of the *report* method for this property package. Again, this is often the same as the state variables, but allows modelers to

include additional variables beyond just the state variables (or port members). Similarly to the *define_port_members* method, this method can be skipped (in which case it defaults to *define_state_vars*) and the key names in the dict can be defined by the user.

The State Block Methods class

The purpose of the *StateBlockMethods* class is to define methods which can be applied to an entire set of indexed *StateBlocks* simultaneously. Whilst the *StateBlockData* class contains the instructions for how to build the variables and constraints that describe the state of a material at a single point in space and time, the *StateBlockMethods* class defines methods for interacting with multiple states across space and time simultaneously. The most common application for this is during initialization of *StateBlocks*, where the same set of instructions needs to be performed on each indexed state; whilst this could be done by iterating over each state and performing the set of instructions, it is generally more efficient to apply the instructions simultaneously across all states.

Declaration and Base Class

Due to the way the *StateBlockMethods* class is provided to the *declare_process_block_class* decorator on the *NewStateBlockData* class, this is one of the few cases where the decorator is not required when declaring a class within IDAES. An example of declaring a new *StateBlockMethods* class is shown below, using the *StateBlock* base class:

```
class   
     NewStateBlockMethods(StateBlock):
```

As the *StateBlockMethods* class is designed to contain methods that can be applied to multiple existing *StateBlockData* objects, rather than construct

a model itself, the *StateBlockMethods* class does not need a *build* method either, nor is it necessary to call *super().build()* as is normal for other modeling components.

Instead, the *StateBlockMethods* class should contain a set of methods which can be called and applied to an indexed *StateBlock* as required. The two methods that must be declared are:

- *initialize*
- *release_state*

The *initialize* and *release_state* Methods

When initializing a unit model, most IDAES models use a hierarchical approach where each state in the model (i.e. each *StateBlockData*) is first initialized at some initial state, after which the unit model attempts to build up and solve the material, energy and momentum balances, etc. The purpose of the *initialize* method is to provide a set of instructions which can take a state from its initial state to a solvable final state at the set of initial conditions (provided as arguments to the *initialize* method). This is generally done by:

1. fixing the state variables at the initial conditions,
2. performing a series of steps to build up the final solution,
3. solving the full state model, and
4. unfixing the state variables (unless they were already fixed when the process began).

However, in order to fully initialize the unit operation (which contains these material state) it is necessary for the unit model to be fully defined (with zero degrees of freedom, i.e. a square model). In order for this to be true however, it is necessary for the inlet states to remain fixed until the unit model has finished initializing. This requires step 4 above to be postponed for inlet states until the unit model has finished initial-

izing, thus the above process is broken into two methods.

1. The *initialize* method covers steps 1-3 above, and is called at the beginning of the unit model initialization process.
2. The *release_state* method covers step 4; for inlet states this is called when the unit model has finished initialization, whilst for all other states it is called immediately by the *initialize* methods when it finishes.

More details on writing initialization methods will be provided elsewhere in the documentation of tutorials.

Tutorials

Tutorials demonstrating how to create custom property packages are being developed. Once they are created, they will be found [here](#).

Custom Reaction Packages

- *What Belongs in a Reaction Package?*
- *Reaction Package Classes*
- *Build-on-Demand Properties*
- *The Reaction Parameter Block*
- *Reaction Parameter Block Configuration*
- *The Reaction Parameter Block build Method*
- *Defining Reaction Metadata*
- * *Setting Default Units*
- * *Setting Reaction Metadata*
- *The Reaction Block*
- *The Reaction Block Data Class*
- * *Reaction Block Data Configuration Arguments*
- * *The Reaction Block build Method*
- * *Variables and Properties*
- * *Required Methods*

- *The Reaction Block Methods Class*
- * *The initialize Method*
- *Tutorials*

Warning: This section is currently being developed

Chemical reactions are a fundamental part of most processes, and models for these come in a wide range of different forms. Much like thermophysical property packages, the ability for users to define custom reaction formulations is a key aspect of the IDAES modeling paradigm.

Reaction packages within IDAES share many similarities with thermophysical property packages, both in form and content. Rather than repeat much of that documentation here, users should start by reading the *thermophysical property package documentation*, as this document will focus on the content of the reaction package.

What Belongs in a Reaction Package?

Chemical reactions are fundamentally governed by the same laws of thermodynamics as thermophysical properties, thus the separation of these into thermophysical and reaction packages is somewhat arbitrary. In IDAES, this separation between thermophysical and reaction packages was based on their expected frequency of use. All unit operations require some set of property calculations (e.g. enthalpy) and these types of calculations were grouped in the thermophysical package, whereas only a small subset of unit operations have chemical reactions and these types of calculations were grouped in the reaction package. This separation benefits the user in that they only need to be concerned about reactions in the unit operations that require them.

Important: For the context of IDAES, chemical reactions are defined as phenomena where one chemical species is

converted into another. This includes both rate limited and equilibrium reactions.

On the other hand, phase equilibrium phenomena (where a chemical species changes phase) are handled via the thermophysical property package.

However, users should note that reaction properties are fundamentally linked to the thermophysical properties, and that a reaction package should only be used with the thermophysical property package they were developed with (in theory at least). Due to this, when a reactions package is added to a model it must be coupled to a thermophysical property package. The modeling framework performs some limited checks to ensure the two packages are compatible (e.g. same set of base units) and that each reaction packages is only used in conjunction with its coupled thermophysical property package in unit models.

Reaction Package Classes

Like thermophysical property packages, reaction property packages consist of two related model components; the Reaction Parameter Block and the Reaction Block, which are analogous to the Physical Parameter Block and State Block components. Similarly, when creating a custom reaction package users need to declare three new classes; the Reaction Parameter Block Data class, the Reaction Block Data class and the Reaction Block Methods class.

Build-on-Demand Properties

IDAES reaction packages also support build-on-demand properties using the same approach as for thermophysical properties.

The Reaction Parameter Block

The first part of the reaction package is the *ReactionParameterBlock*, which defines the global parameters and components of the property package. This includes:

- a reference to the *ReactionBlock* class associated with the *ReactionParameterBlock*,
- a Pyomo Set listing names for all rate-based reactions,
- a dict defining stoichiometric coefficients for all rate-based reactions,
- a Pyomo Set listing names for all equilibrium reactions,
- a dict defining stoichiometric coefficients for all equilibrium reactions,
- the base units of measurement for the property packages,
- the reaction properties supported by the property package, and
- the parameters required to calculate the reaction properties.

The starting point for creating a new *ReactionParameterBlock* is shown in the example below. The model developer needs to declare a new class which inherits from the *ReactionParameterBlock* base class, decorated using the *declare_process_block_class* decorator.

```
@declare_process_block_class(
    ↪ "NewReactionParameterBlock"
    ↪ ")
class_
    ↪ NewReactionParameterData (ReactionParameterBlock):

    def build(self):
        super().build()

    @classmethod
    def_
    ↪ define_metadata(cls, obj):
        obj.add_
    ↪ properties({# properties}))
        obj.add_
    ↪ default_units({# units}))
```

The *NewReactionParameterData* class needs to contain a *build* method, and may also include a configuration block and a *define_metadata* classmethod as shown above. These methods and their contents will be explained below.

Reaction Parameter Block Configuration

The *ReactionParameterBlock* configuration block must contain the following two arguments:

- “property_package” - this configuration argument contains a pointer to the associated thermophysical property package (via an instance of a *PhysicalParameterBlock*), and is used for validating the link between thermophysical and reaction properties (e.g. confirming that both packages use the same set of base units).
- “default_arguments” - this configuration argument allows users to specify a set of default configuration arguments that will be passed to all *ReactionBlocks* created from an instance of a parameter block.

The Reaction Parameter Block *build* Method

The *build* method in the *NewReactionParameterBlock* class is responsible for constructing the various modeling components that will be required by the associated *ReactionBlocks*. This includes the indexing sets which will be used to identify individual reactions and the stoichiometry of each of these. The *build* method is also responsible for setting up the underlying infrastructure of the property package and making a link to the associated *ReactionBlock* class so that the modeling framework can automate the construction and linking of these.

The first step in the *build* method is to call *super().build()* to trigger the construction of the underlying infrastructure using the base class’ *build* method.

Next, the user must declare an attribute named “_reaction_block_class” which is a pointer to the associated *Reaction-Block* class (creation of this will be discussed later). An example of this is shown below, where the associated Reaction Block class is named *NewReactionBlock*.

```
def build(self):
    super().build()
    self._reaction_block_
    ↪class = NewReactionBlock
```

Next, the *build* method must create two indexing sets which provide names for the rate- and equilibrium-based reaction respectively. These indexing sets must be named *rate_reaction_idx* and *equilibrium_reaction_idx*. These indexing sets will be used by the unit models and control volumes when creating reaction terms in material balance equations.

```
self.rate_reaction_idx_
    ↪= Set(initialize=["rate_
    ↪rxn_1", "rate_rxn_2"])
self.
    ↪equilibrium_reaction_idx_
    ↪= Set(initialize=["equil_
    ↪rxn_1", "equil_rxn_2"])
```

Note: Users only need to define indexing sets and stoichiometry dicts for the types of reaction which they wish to model. E.g. users do not need to declare *rate_reaction_idx* and *rate_reaction_stoichiometry* if there are no rate-based reactions in their system.

The *build* method also needs to create stoichiometry *dicts* for the rate- and equilibrium-based reactions present in the system. These *dicts* should be named “rate_reaction_stoichiometry” and “equilibrium_reaction_stoichiometry” and have keys with the form (reaction_index, phase, component) and values equal to the stoichiometric coefficient for the given reaction, phase and component. A positive stoichiometric coefficient indicates a product of the reaction (i.e. generation) whilst a negative coefficient indicates a reactant (i.e. consumption). An example for defining the stoichiometry

for rate-based reactions is shown below.

```
self.rate_  
↳ reaction_stoichiometry = {  
    ("rate_rxn_  
↳ 1", "phase_1", "component_  
↳ 1"): -1, # Component_  
↳ 1 in phase 1 is a reactant  
    ("rate_rxn_  
↳ 1", "phase_2", "component_  
↳ 1"): 0, # Reaction_  
↳ 1 does no occur is phase 2  
    ("rate_rxn_  
↳ 1", "phase_1", "component_  
↳ 2"): 2, # Component_  
↳ 2 in phase 1 is a product  
    ("rate_rxn_1", "phase_  
↳ 2", "component_2"): 0,  
    ("rate_rxn_2", "phase_  
↳ 1", "component_1"): 0,  
    ("rate_rxn_2", "phase_  
↳ 2", "component_1"): -1,  
    ("rate_rxn_2", "phase_  
↳ 1", "component_2"): 0,  
      
↳  
↳ ("rate_rxn_2", "phase_2",   
↳ "component_2"): -1} # etc.
```

Important: Stoichiometry *dicts* must contain a key for every reaction-phase-component combination, even if the stoichiometric coefficient is zero.

Finally, the *build* method needs to declare all the global parameters that will be used by the reaction calculations. Similar to thermophysical property parameters, users are encouraged to declare these as Pyomo *Vars* rather than *Params* to facilitate parameter estimation studies.

Defining Reaction Metadata

The last part of creating a new Reaction Parameter block is to define the metadata associated with it. The reactions metadata serves four purposes:

1. The default units metadata is used by the framework to automatically determine the units of measurement of the resulting property model, and automatically convert between different unit sets where appropriate.

2. The properties metadata is used to set up any build-on-demand properties,
3. The metadata is also used by the Data Management Framework to index the available property packages to create a searchable index for users.
4. The units metadata is compared to that of the associated thermophysical property package (when an instance of the Reaction Parameter Block is declared), and an exception is raised if they do not match.

Setting Default Units

As with thermophysical property packages, the most important part of defining the metadata for a property package is to set the default units of measurement for each of the 7 base quantities (time, length, mass, amount, temperature, current (optional) and luminous intensity (optional)). These units are used by the modeling framework to determine the units of measurement for all other quantities in the process that are related to this property package. More importantly, the units metadata is used to determine if a reaction package is comparable with a given thermophysical property package when they are declared – if the units metadata does not match, an exception will be raised and the two packages cannot be used together.

Units must be defined using Pyomo *Units* components, as shown in the example below:

```
from_
↳ pyomo.environ import units

@classmethod
def_
↳ define_metadata(cls, obj):
    obj.add_default_
↳ units({'time': units.s,

↳         'length': units.m,

↳         'mass': units.kg,

↳         'amount': units.mol,
```

(continues on next page)

(continued from previous page)

```
→ 'temperature': units.K))
```

Setting Reaction Metadata

Similar to thermophysical property packages, reaction packages allow users to specify the set of reaction properties supported by a given reaction package. This is also used to set up the build-on-demand properties system in the same way as thermophysical properties. For more information, see the documentation for *thermophysical properties metadata*.

The Reaction Block

The second part of a reaction property package is the *ReactionBlock* class. Similarly to *StateBlock* classes this is defined using two user-written classes; the *ReactionBlockData* class and the *ReactionBlockMethods* class.

[illegible]

(continues on next page)

(continued from previous page)

```
def build(self):  
    super().build()
```

The Reaction Block Data Class

One important difference between Reaction Blocks and State Blocks is that while State Blocks are fully self-contained and can be solved in isolation, Reaction Blocks depend upon the State Block for the definition of the state variables. This means that Reaction Blocks do not need to redefine the state variables (which are needed for the reaction properties), but at the cost of not being independent, self-contained models. This is one of the reasons why reaction packages are so closely tied to thermophysical property packages within IDAES.

The purpose of the Reaction Block Data class is to define the reaction properties that will be required by the unit models using this package. The three main properties required for material and energy balances are:

- rate terms for rate-based reactions,
- equilibrium constraints for equilibrium-based reactions, and
- heats of reaction (if required, see note below).

These properties may in turn depend on other reaction properties such as equilibrium and rate constants. All of these properties may be constructed using the build-on-demand framework.

All reaction properties depend upon the state of the material, which is defined in the State Block; thus it is necessary to reference the associated State Block whenever these are needed. In order to facilitate this, the ReactionBlockData base class establishes a reference to the associated State Block which users can use to obtain state variables and properties from the State Block. For example,

temperature can be referenced from the state block as shown below:

```
temperature = _  
↪ self.state_ref.temperature
```

Note: There are multiple ways in which heat of reaction may be included in a model, and users should consider which is most suitable for their application. The two most common approaches are to include an explicit heat of reaction term in the energy balance equations, or to incorporate heat of reaction into the specific enthalpy terms (generally via heats of formation). The IDAES Process Modeling Framework supports both of these approaches.

Reaction Block Data Configuration Arguments

The `ReactionBlockData` base class defines three configuration arguments that are required for all Reaction Block Data classes.

- “parameters” – this argument is used to provide a link back to the associated *ReactionParameterBlock*, and is generally automatically passed to the *ReactionBlock* when it is constructed.
- “state_block” – this argument is used to provide a link to the State Block associated with this Reaction Block, as is generally passed to the *ReactionBlock* by the unit model when it is constructed. This argument is used to the *state_ref* attribute shown above for referencing properties from the State Block.
- “has_equilibrium” – this argument indicates whether equilibrium reaction will be considered for this state. In most cases, this argument will always be `True`, however this allows users the ability to turn off equilibrium reactions if they desire.

The Reaction Block *build* Method

As with all IDAES components, the *build* method forms the core of a *ReactionBlockData* class, and contains the instructions on how to construct the variables, expressions and constraints required by the reaction model. As usual, the first step in the *build* method should be to call *super().build()* to trigger the construction of the underly-

ing components required for Reaction Blocks to function.

Variables and Properties

The same set of guidelines for defining thermophysical properties apply to reaction properties, *which can be found [here](#)*.

Required Methods

In addition to the *build* method, Reaction Blocks require one additional method which is used to define the basis for the reaction terms.

- *get_reaction_rate_basis* - must return a *MaterialFlowBasis Enum*, and is used to automatically convert reaction terms between mass and mole basis in control volumes.

The Reaction Block Methods Class

The Reaction Block Methods class is very similar to the *State Block Methods class*. The Reaction Block Methods class needs to contain an *initialize* method (however a *release_state* method is not required as Reaction Blocks do not contain state variables).

The *initialize* Method

Initialization of Reaction Blocks is complicated by the fact that they depend upon the State Block for the state variables, and thus cannot be solved as a stand-alone model. Within the wider IDAES modeling framework, this is handled by initializing the Reaction Block after the State Block *initialization* method has been called (and thus all state variables and properties are initialized) but before the *release_State* method is called (thus all state variables are fixed). Thus, the Reaction Block can assume that the state is fully defined and

initialized (although it may not be possible to use a solver as part of the Reaction Block's initialization procedure).

However, Reaction Blocks also tend to be much simpler than State Blocks, involving fewer properties which are generally much less tightly coupled (most reaction properties are functions solely of the state variables), which simplifies the requirements of initializing the sub-model.

Tutorials

Tutorials demonstrating how to create custom reaction packages are being developed. Once they are created, they will be found [here](#).

4.4 Tutorials and Examples

4.4.1 Viewing online

Tutorials and examples for IDAES are located on the [examples online documentation page](#). If you are new to IDAES, it is strongly recommended to start with the tutorials. There are also pre-recorded tutorial videos on the [IDAES page on YouTube](#).

4.4.2 Running locally

To run and use the examples on your own computer, once you have installed IDAES, you should run `idaes get-examples` in a command-line shell. Please see [this page](#) for details on how to use this program.

Once you have installed the examples, change the directory where you downloaded them and run the following command¹

```
jupyter notebook ↵  
↪ notebook_index.ipynb
```

This will open a descriptive Jupyter Notebook with links that allow you to open and run any of the other tutorial and example notebooks. Happy modeling!

¹ If you have installed [JupyterLab](#), you can use it by simply substituting “jupyter lab” for “jupyter notebook” in the given command.

4.4.3 Additional information

The sources for the tutorials and examples are maintained on the [IDAES examples repository](#).

If you want to develop custom unit and property models, please refer to the *advanced user guide*.

4.5 Technical Specifications

4.5.1 Core

Process Block

Example

ProcessBlock is used to simplify inheritance of Pyomo’s Block. The code below provides an example of how a new ProcessBlock class can be implemented. The new ProcessBlock class has a ConfigBlock that allows each element of the block to be passed configuration options that affect how a block is built. ProcessBlocks have a rule set by default that calls the build method of the contained ProcessBlockData class.

```
from pyomo.environ import *
from pyomo.common.
↳ config import ConfigValue
from idaes.core_
↳ import ProcessBlockData, _
↳ declare_process_block_class

@declare_process_
↳ block_class("MyBlock")
```

(continues on next page)

(continued from previous page)

```

class_
↳ MyBlockData (ProcessBlockData) :
    CONFIG_
↳ = ProcessBlockData.CONFIG()
    CONFIG.declare("xinit",
↳ ConfigValue(default=1001,
↳ domain=float))
    CONFIG.declare("yinit",
↳ ConfigValue(default=1002,
↳ domain=float))
    def build(self):
        super(MyBlockData,
↳ self).build()
        self.
↳ x = Var(initialize=self.
↳ config.xinit)
        self.
↳ y = Var(initialize=self.
↳ config.yinit)

```

The following example demonstrates creating a scalar instance of the new class. The default key word argument is used to pass information on the the MyBlockData ConfigBlock.

```

m = ConcreteModel()
m.b = MyBlock(default=
↳ {"xinit":1, "yinit":2})

```

The next example creates an indexed MyBlock instance. In this case, each block is configured the same, using the default argument.

```

m = ConcreteModel()
m.b = MyBlock([0,
↳ 1,2,3,4], default=
↳ {"xinit":1, "yinit":2})

```

The next example uses the initialize argument to override the configuration of the first block. Initialize is a dictionary of dictionaries where the key of the top level dictionary is the block index and the second level dictionary is arguments for the config block.

```

m = ConcreteModel()
m.b = MyBlock([0,
↳ 1,2,3,4], default=
↳ {"xinit":1, "yinit":2},
    initialize=
↳ {0:{"xinit":1, "yinit":2}})

```

The next example shows a more complicated configuration where there are three configurations, one for the first block, one for the last block, and one for the interior blocks. This is accomplished by providing the `idx_map` argument to `MyBlock`, which is a function that maps a block index to a index in the initialize dictionary. In this case 0 is mapped to 0, 4 is mapped to 4, and all elements between 0 and 4 are mapped to 1. A lambda function is used to convert the block index to the correct index in initialize.

```
m = ConcreteModel()
m.b = MyBlock(
    [0,1,2,3,4],
    idx_map = lambda i: 1_
↪ if i > 0 and i < 4 else i,
    initialize={0:{"xinit
↪ ":2001, "yinit":2002},
                1:{"xinit
↪ ":5001, "yinit":5002},
                4:{"xinit
↪ ":7001, "yinit":7002}})
```

The build method

The core part of any IDAES Block is the build method, which contains the instructions on how to construct the variables, constraints and other components that make up the model. The build method serves as the default rule for constructing an instance of an IDAES Block, and is triggered automatically whenever an instance of an IDAES Block is created unless a custom rule is provided by the user.

ProcessBlock Class

```
idaes.core.process_block.declare_process_block_class(name,      block_class=<class
                                                         'idaes.core.process_block.ProcessBlock'>,
                                                         doc="")
```

Declare a new ProcessBlock subclass.

This is a decorator function for a class definition, where the class is derived from Pyomo's `_BlockData`. It creates a ProcessBlock subclass to contain the decorated class. The only requirement is

that the subclass of `_BlockData` contain a `build()` method. The purpose of this decorator is to simplify subclassing Pyomo's block class.

Parameters

- **name** – name of class to create
- **block_class** – `ProcessBlock` or a subclass of `ProcessBlock`, this allows you to use a subclass of `ProcessBlock` if needed. The typical use case for Subclassing `ProcessBlock` is to impliment methods that operate on elements of an indexed block.
- **doc** – Documentation for the class. This should play nice with sphinx.

Returns Decorator function

```
class idaes.core.process_block.ProcessBlock(*args, **kws)
```

`ProcessBlock` is a Pyomo Block that is part of a system to make Pyomo Block easier to subclass. The main difference between a Pyomo Block and `ProcessBlock` from the user perspective is that a `ProcessBlock` has a rule assigned by default that calls the `build()` method for the contained `ProcessBlockData` objects. The default rule can be overridden, but the new rule should always call `build()` for the `ProcessBlockData` object.

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - "Block"
- **default** (*dict*) – Default `ProcessBlockData` config
- **initialize** (*dict*) – `ProcessBlockData` config for individual elements. Keys are `BlockData` indexes and values are dictionaries described under the "default" argument above.
- **idx_map** (*function*) – Function to take the index of a `BlockData` element and return the index in the initialize dict from which to read arguments. This can

be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (ProcessBlock) New instance

classmethod base_class_module()
Return module of the associated ProcessBase class.

Returns (str) Module of the class.

Raises `AttributeError`, if no base class module was set, e.g. this class – was *not* wrapped by the `declare_process_block_class` decorator.

classmethod base_class_name()
Name given by the user to the ProcessBase class.

Returns (str) Name of the class.

Raises `AttributeError`, if no base class name was set, e.g. this class – was *not* wrapped by the `declare_process_block_class` decorator.

class `idaes.core.process_base.ProcessBlockData` (*component*)
Base class for most IDAES process models and classes.

The primary purpose of this class is to create the local config block to handle arguments provided by the user when constructing an object and to ensure that these arguments are stored in the config block.

Additionally, this class contains a number of methods common to all IDAES classes.

build()

The build method is called by the default ProcessBlock rule. If a rule is specified other than the default it is important to call ProcessBlockData's build method to put information from the "default" and "initialize" arguments to a ProcessBlock derived class into the BlockData object's ConfigBlock.

The the build method should usually be overloaded in a subclass derived from ProcessBlockData. This method would generally add Pyomo components such as variables, expressions,

and constraints to the object. It is important for `build()` methods implemented in derived classes to call `build()` from the super class.

Parameters `None` –

Returns `None`

`fix_initial_conditions` (*state='steady-state'*)

This method fixes the initial conditions for dynamic models.

Parameters `state` – initial state to use for simulation (default = 'steady-state')

Returns : `None`

`flowsheet` ()

This method returns the components parent flowsheet object, i.e. the flowsheet component to which the model is attached. If the component has no parent flowsheet, the method returns `None`.

Parameters `None` –

Returns Flowsheet object or `None`

`serialize_contents` (*time_point=0*)

Return the performance contents and stream table

NOTE: There is the possibility of a `ConfigurationError` because the names of the inlets and outlets of the unit model may not be standard. If this occurs then return an empty dataframe

Parameters `time_point` – The time

Returns Pandas dataframe with the performance contents `stream_table`: Pandas dataframe with the stream table for a unit model

Return type `performance_contents`

`unfix_initial_conditions` ()

This method unfixed the initial conditions for dynamic models.

Parameters `None` –

Returns : `None`

Flowsheet Block

Default Property Packages

Flowsheet Blocks may assign a property package to use as a default for all UnitModels within the Flowsheet. If a specific property package is not provided as an argument when constructing a UnitModel, the UnitModel will search up the model tree until it finds a default property package declared. The UnitModel will use the first default property package it finds during the search, and will return an error if no default is found.

Flowsheet Configuration Arguments

Flowsheet blocks have three configuration arguments which are stored within a `Config` block (`flowsheet.config`). These arguments can be set by passing arguments when instantiating the class, and are described below:

- `dynamic` - indicates whether the flowsheet should be dynamic or steady-state. If `dynamic = True`, the flowsheet is declared to be a dynamic flowsheet, and the time domain will be a `Pyomo ContinuousSet`. If `dynamic = False`, the flowsheet is declared to be steady-state, and the time domain will be an ordered `Pyomo Set`. For top level Flowsheets, `dynamic` defaults to `False` if not provided. For lower level Flowsheets, the `dynamic` will take the same value as that of the parent model if not provided. It is possible to declare steady-state sub-Flowsheets as part of dynamic Flowsheets if desired, however the reverse is not true (cannot have dynamic Flowsheets within steady-state Flowsheets).
- `time` - a reference to the time domain for the flowsheet. During flowsheet creation, users may provide a `Set` or `ContinuousSet` that the flowsheet should use as the time domain. If not provided, then the flowsheet will look for a parent flowsheet and set this equal to the parent's time domain, otherwise a new

time domain will be created and assigned here.

- `time_units` - used to specify the units of the time domain, and must be a Pyomo Unit object (cannot be a compound unit). This is necessary for dynamic flowsheets, but can be neglected in steady-state cases. In cases where the time domain is inherited from a parent flowsheet, the time units will also be inherited.
- `time_set` - used to initialize the time domain in top-level Flowsheets. When constructing the time domain in top-level Flowsheets, `time_set` is used to initialize the ContinuousSet or Set created. This can be used to set start and end times, and to establish points of interest in time (e.g. times when disturbances will occur). If `dynamic = True`, `time_set` defaults to `[0.0, 1.0]` if not provided, if `dynamic = False` `time_set` defaults to `[0.0]`. `time_set` is not used in sub-Flowsheets and will be ignored.
- `default_property_package` - can be used to assign the default property package for a Flowsheet. Defaults to `None` if not provided.

Flowsheet Classes

class `idaes.core.flowsheet_model.FlowsheetBlockData` (*component*)

The `FlowsheetBlockData` Class forms the base class for all IDAES process flowsheet models. The main purpose of this class is to automate the tasks common to all flowsheet models and ensure that the necessary attributes of a flowsheet model are present.

The most significant role of the `FlowsheetBlockData` class is to automatically create the time domain for the flowsheet.

build()

General build method for `FlowsheetBlockData`. This method calls a number of sub-methods which automate the construction of expected attributes of flowsheets.

Inheriting models should call `super().build`.

Parameters None –

Returns None

```
get_costing (module=<module 'idaes.core.util.unit_costing' from
                '/home/docs/checkouts/readthedocs.org/user_builds/idaes-
                pse/checkouts/1.9.0/idaes/core/util/unit_costing.py'>, year=None, inte-
                ger_n_units=False)
```

Creates a new block called 'costing' at the flowsheet level. This block builds global parameters used in costing methods (power plant costing and generic costing).

Parameters

- **idaes flowsheet** (*self*) –
- **year** – used to build parameter CE_index (Chemical Engineering),
- **parameter is the same for all costing blocks in the flowsheet** (*this*) –
- **integer_n_units** – flag to define variable domain (True: domain is
- **Integer numbers** (*within*) – domain is NonNegativeReals).
- **False** – domain is NonNegativeReals).

Returns None

```
is_flowsheet ()
```

Method which returns True to indicate that this component is a flowsheet.

Parameters None –

Returns True

```
model_check ()
```

This method runs model checks on all unit models in a flowsheet.

This method searches for objects which inherit from UnitModelBlockData and executes the model_check method if it exists.

Parameters None –

Returns None

```
stream_table (true_state=False, time_point=0, orient='columns')
```

Method to generate a stream table by iterating over all Arcs in the flowsheet.

Parameters

- **true_state** – whether the state variables (True) or display variables (False, default) from the StateBlocks should be used in the stream table.
- **time_point** – point in the time domain at which to create stream table (default = 0)
- **orient** – whether stream should be shown by columns (“columns”) or rows (“index”)

Returns A pandas dataframe containing stream table information

visualize (*model_name*, ***kwargs*)

Starts up a flask server that serializes the model and pops up a webpage with the visualization

Parameters **model_name** – The name of the model that flask will use as an argument for the webpage

Keyword Arguments ****kwargs** – Additional keywords for `idaes.ui.fsvis.visualize()`

Returns None

class `idaes.core.flowsheet_model.FlowsheetBlock` (**args*, ***kws*)

FlowsheetBlock is a specialized Pyomo block for IDAES flowsheet models, and contains instances of FlowsheetBlockData.

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default ProcessBlockData config

Keys

dynamic Indicates whether this model will be dynamic, **default** - `useDefault`.
Valid values: { **useDefault** - get flag from parent or False, **True** - set as a dynamic model, **False** - set as a steady-state model. }

time Pointer to the time domain for the flowsheet. Users may provide an existing time domain from another flowsheet, otherwise the flowsheet will search for a parent with a time domain or create a new time domain and reference it here.

time_set Set of points for initializing time domain. This should be a list of floating point numbers, **default** - [0].

time_units Pyomo Units object describing the units of the time domain. This must be defined for dynamic simulations, **default** = None.

default_property_package Indicates the default property package to be used by models within this flowsheet if not otherwise specified, **default** - None. **Valid values:** { **None** - no default property package, a **ParameterBlock** object. }

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (FlowsheetBlock) New instance

0D Control Volume Class

Contents

- *0D Control Volume Class*
- *ControlVolume0DBlock Equations*

The ControlVolume0DBlock block is the most commonly used Control Volume class, and is used for systems where there is a well-mixed volume of fluid, or where variations in spatial domains are considered to be negligible.

ControlVolume0DBlock blocks generally contain two *StateBlocks* - one for the incoming material and one for the material within and leaving the volume - and one *StateBlocks*.

class `idaes.core.control_volume0d.ControlVolume0DBlock` (*args, **kws)

ControlVolume0DBlock is a specialized Pyomo block for IDAES non-discretized control volume blocks, and contains instances of ControlVolume0DBlockData.

ControlVolume0DBlock should be used for any control volume with a defined volume and distinct inlets and outlets which does not require spatial discretization. This encompasses most basic unit models used in process modeling.

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic, **default** - useDefault.

Valid values: { **useDefault** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **True** - construct holdup terms, **False** - do not construct holdup terms }

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object.}

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

auto_construct If set to True, this argument will trigger the auto_construct method which will attempt to construct a set of material, energy and momentum balance equations based on the parent unit's config block. The parent unit must have a config block which derives from CONFIG_Base, **default** - False. **Valid values:** { **True** - use automatic construction, **False** - do not use automatic construction.}

- **initialize** (*dict*) - ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the "default" argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (ControlVolume0DBlock) New instance

class `idaes.core.control_volume0d.ControlVolume0DBlockData` (*component*)
0-Dimensional (Non-Discretised) ControlVolume Class

This class forms the core of all non-discretized IDAES models. It provides methods to build property and reaction blocks, and add mass, energy and mo-

mentum balances. The form of the terms used in these constraints is specified in the chosen property package.

add_geometry()

Method to create volume Var in ControlVolume.

Parameters None –

Returns None

add_phase_component_balances (*has_rate_reactions=False, has_equilibrium_reactions=False, has_phase_equilibrium=False, has_mass_transfer=False, custom_molar_term=None, custom_mass_term=None*)

This method constructs a set of 0D material balances indexed by time, phase and component.

Parameters

- **has_rate_reactions** – whether default generation terms for rate reactions should be included in material balances
- **has_equilibrium_reactions** – whether generation terms should for chemical equilibrium reactions should be included in material balances
- **has_phase_equilibrium** – whether generation terms should for phase equilibrium behaviour should be included in material balances
- **has_mass_transfer** – whether generic mass transfer terms should be included in material balances
- **custom_molar_term** – a Pyomo Expression representing custom terms to be included in material balances on a molar basis. Expression must be indexed by time, phase list and component list
- **custom_mass_term** – a Pyomo Expression representing custom terms to be included in material balances on a mass basis. Expression must be indexed by time, phase list and component list

Returns Constraint object representing material balances

add_phase_energy_balances (**args, **kwargs*)

Method for adding energy balances (including kinetic energy) indexed by phase to the control volume.

See specific control volume documentation for details.

add_phase_enthalpy_balances (*args, **kwargs)

Method for adding enthalpy balances indexed by phase to the control volume.

See specific control volume documentation for details.

add_phase_momentum_balances (*args, **kwargs)

Method for adding momentum balances indexed by phase to the control volume.

See specific control volume documentation for details.

add_phase_pressure_balances (*args, **kwargs)

Method for adding pressure balances indexed by phase to the control volume.

See specific control volume documentation for details.

add_reaction_blocks (has_equilibrium=None)

This method constructs the reaction block for the control volume.

Parameters

- **has_equilibrium** – indicates whether equilibrium calculations will be required in reaction block
- **package_arguments** – dict-like object of arguments to be passed to reaction block as construction arguments

Returns None

add_state_blocks (information_flow=<FlowDirection.forward:
has_phase_equilibrium=None) l>,

This method constructs the inlet and outlet state blocks for the control volume.

Parameters

- **information_flow** – a FlowDirection Enum indicating whether information flows from inlet-to-outlet or outlet-to-inlet
- **has_phase_equilibrium** – indicates whether equilibrium calculations will be required in state blocks
- **package_arguments** – dict-like object of arguments to be passed to state blocks as construction arguments

Returns None

add_total_component_balances (*has_rate_reactions=False, has_equilibrium_reactions=False, has_phase_equilibrium=False, has_mass_transfer=False, custom_molar_term=None, custom_mass_term=None*)

This method constructs a set of 0D material balances indexed by time and component.

Parameters

- **- whether default generation terms for rate** (*has_rate_reactions*) – reactions should be included in material balances
- **- whether generation terms should for** (*has_equilibrium_reactions*) – chemical equilibrium reactions should be included in material balances
- **- whether generation terms should for phase** (*has_phase_equilibrium*) – equilibrium behaviour should be included in material balances
- **- whether generic mass transfer terms should be** (*has_mass_transfer*) – included in material balances
- **- a Pyomo Expression representing custom terms to** (*custom_mass_term*) – be included in material balances on a molar basis. Expression must be indexed by time, phase list and component list
- **- a Pyomo Expression representing custom terms to** – be included in material balances on a mass basis. Expression must be indexed by time, phase list and component list

Returns Constraint object representing material balances

add_total_element_balances (*has_rate_reactions=False, has_equilibrium_reactions=False, has_phase_equilibrium=False, has_mass_transfer=False, custom_elemental_term=None*)

This method constructs a set of 0D element balances indexed by time.

Parameters

- **- whether default generation terms for rate**

(*has_rate_reactions*) – reactions should be included in material balances

- – **whether generation terms should for**
(*has_equilibrium_reactions*) – chemical equilibrium reactions should be included in material balances
- – **whether generation terms should for phase**
(*has_phase_equilibrium*) – equilibrium behaviour should be included in material balances
- – **whether generic mass transfer terms should be**
(*has_mass_transfer*) – included in material balances
- – **a Pyomo Expression representing custom**
(*custom_elemental_term*) – terms to be included in material balances on a molar elemental basis. Expression must be indexed by time and element list

Returns Constraint object representing material balances

add_total_energy_balances (*args, **kwargs)

Method for adding a total energy balance (including kinetic energy) to the control volume.

See specific control volume documentation for details.

add_total_enthalpy_balances (*has_heat_of_reaction=False*, *has_heat_transfer=False*,
has_work_transfer=False, *has_enthalpy_transfer=False*,
custom_term=None)

This method constructs a set of 0D enthalpy balances indexed by time and phase.

Parameters

- – **whether terms for heat of reaction should**
(*has_heat_of_reaction*) – be included in enthalpy balance
- – **whether terms for heat transfer should be**
(*has_heat_transfer*) – included in enthalpy balances

- - **whether terms for work transfer should be**
(*has_work_transfer*) - included in enthalpy balances
- - **whether terms for enthalpy transfer due to**
(*has_enthalpy_transfer*) - mass transfer should be included in enthalpy balance. This should generally be the same as the *has_mas_trasnfer* argument in the material balance methods
- - **a Pyomo Expression representing custom terms**
to (*custom_term*) - be included in enthalpy balances. Expression must be indexed by time and phase list

Returns Constraint object representing enthalpy balances

add_total_material_balances (**args, **kwargs*)

Method for adding a total material balance to the control volume.

See specific control volume documentation for details.

add_total_momentum_balances (**args, **kwargs*)

Method for adding a total momentum balance to the control volume.

See specific control volume documentation for details.

add_total_pressure_balances (*has_pressure_change=False, custom_term=None*)

This method constructs a set of 0D pressure balances indexed by time.

Parameters

- - **whether terms for pressure change should be** (*has_pressure_change*) - included in enthalpy balances
- - **a Pyomo Expression representing custom terms**
to (*custom_term*) - be included in pressure balances. Expression must be indexed by time

Returns Constraint object representing pressure balances

build()

Build method for ControlVolume0DBlock blocks.

Returns None

initialize (*state_args=None, outlvl=0, optarg={}, solver=None, hold_state=True*)

Initialization routine for OD control
volume (default solver ipopt)

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output log level of initialization routine
- **optarg** – solver options dictionary object (default={})
- **solver** – str indicating which solver to use during initialization (default = None)
- **hold_state** – flag indicating whether the initialization routine should unfix any state variables fixed during initialization, **default** - True. **Valid values:** **True** - states variables are not unfixed, and a dict of returned containing flags for which states were fixed during initialization, **False** - state variables are unfixed after initialization by calling the `release_state` method.

Returns If `hold_states` is True, returns a dict containing flags for which states were fixed during initialization.

model_check ()

This method executes the `model_check` methods on the associated state blocks (if they exist). This method is generally called by a unit model as part of the unit's `model_check` method.

Parameters None –

Returns None

release_state (*flags, outlvl=0*)

Method to release state variables fixed during initialization.

Keyword Arguments

- **flags** – dict containing information of which state variables were fixed during initialization, and should now be unfixed. This dict is returned by `initialize` if `hold_state = True`.

- `outlvl` – sets output level of logging

Returns None

ControlVolume0DBlock Equations

This section documents the variables and constraints created by each of the methods provided by the `ControlVolume0DBlock` class.

- t indicates time index
- p indicates phase index
- j indicates component index
- e indicates element index
- r indicates reaction name index

`add_geometry`

The `add_geometry` method creates a single variable within the control volume named *volume* indexed by time (allowing for varying volume over time). A number of other methods depend on this variable being present, thus this method should generally be called first.

Variables

Variable Name	Symbol	Indices	Conditions
volume	V_t	t	None

Constraints

No additional constraints

`add_phase_component_balances`

Material balances are written for each component in each phase (e.g. separate balances for liquid water and steam). Physical property packages may include information to indicate that certain species do not appear in all phases, and material balances will not be written in these cases (if *has_holdup* is True holdup terms will still appear for these species, however these will be set to 0).

Variables

Variable Name	Symbol	Indices	Conditions
material_holdup	$M_{t,p,j}$	t, p, j	has_holdup = True
phase_fraction	$\phi_{t,p}$	t, p	has_holdup = True
material_accumulation	$\frac{\partial M_{t,p,j}}{\partial t}$	t, p, j	dynamic = True
rate_reaction_generation	$N_{kinetic,t,p,j}$	t, p, j	has_rate_reactions = True
rate_reaction_extent	$X_{kinetic,t,r}$	t, r	has_rate_reactions = True
equilibrium_reaction_generation	$N_{equilibrium,t,p,j}$	t, p, j	has_equilibrium_reactions = True
equilibrium_reaction_extent	$X_{equilibrium,t,r}$	t, r	has_equilibrium_reactions = True
phase_equilibrium_generation	$N_{pe,t,p,j}$	t, p, j	has_phase_equilibrium = True
mass_transfer_term	$N_{transfer,t,p,j}$	t, p, j	has_mass_transfer = True

Constraints

material_balances(t, p, j):

$$\frac{\partial M_{t,p,j}}{\partial t} = F_{in,t,p,j} - F_{out,t,p,j} + N_{kinetic,t,p,j} + N_{equilibrium,t,p,j} + N_{pe,t,p,j} + N_{transfer,t,p,j} + N_{custom,t,p,j}$$

The $N_{custom,t,p,j}$ term allows the user to provide custom terms (variables or expressions) in both mass and molar basis which will be added into the material balances, which will be converted as necessary to the same basis as the material balance (by multiplying or dividing by the component molecular weight). The basis of the material balance is determined by the physical property package, and if undefined (or not mass or mole basis), an Exception will be returned.

If *has_holdup* is True, *material_holdup_calculation*(t, p, j):

$$M_{t,p,j} = \rho_{t,p,j} \times V_t \times \phi_{t,p}$$

where $\rho_{t,p,j}$ is the density of component j in phase p at time t

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial M_{t,p,j}}{\partial t}$, will be performed by Pyomo.DAE.

If *has_rate_reactions* is True, *rate_reaction_stoichiometry_constraint*(t, p, j):

$$N_{kinetic,t,p,j} = \alpha_{r,p,j} \times X_{kinetic,t,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component j in phase p for reaction r (as defined in the PhysicalParameterBlock).

If *has_equilibrium_reactions*
argument
is True,
equilibrium_reaction_stoichiometry_constraint(*t*,
p, *j*):

$$N_{equilibrium,t,p,j} = \alpha_{r,p,j} \times X_{equilibrium,t,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

add_total_component_balances

Material balances are written for each component across all phases (e.g. one balance for both liquid water and steam). Most terms in the balance equations are still indexed by both phase and component however. Physical property packages may include information to indicate that certain species do not appear in all phases, and material balances will not be written in these cases (if *has_holdup* is True holdup terms will still appear for these species, however these will be set to 0).

Variables

Variable Name	Symbol	Indices	Conditions
material_holdup	$M_{t,p,j}$	t, p, j	has_holdup = True
phase_fraction	$\phi_{t,p}$	t, p	has_holdup = True
material_accumulation	$\frac{\partial M_{t,p,j}}{\partial t}$	t, p, j	dynamic = True
rate_reaction_generation	$N_{kinetic,t,p,j}$	t, p, j	has_rate_reactions = True
rate_reaction_extent	$X_{kinetic,t,r}$	t, r	has_rate_reactions = True
equilibrium_reaction_generation	$N_{equilibrium,t,p,j}$	t, p, j	has_equilibrium_reactions = True
equilibrium_reaction_extent	$X_{equilibrium,t,r}$	t, r	has_equilibrium_reactions = True
mass_transfer_term	$N_{transfer,t,p,j}$	t, p, j	has_mass_transfer = True

Constraints

material_balances(*t*, *j*):

$$\sum_p \frac{\partial M_{t,p,j}}{\partial t} = \sum_p F_{in,t,p,j} - \sum_p F_{out,t,p,j} + \sum_p N_{kinetic,t,p,j} + \sum_p N_{equilibrium,t,p,j} + \sum_p N_{pe,t,p,j} + \sum_p N_{transfer,t,p,j} + N_{custom,t,j}$$

The $N_{custom,t,j}$ term allows the user to provide custom terms (variables or expressions) in both mass and molar basis which will be added into the material balances, which will be converted as

necessary to the same basis as the material balance (by multiplying or dividing by the component molecular weight). The basis of the material balance is determined by the physical property package, and if undefined (or not mass or mole basis), an Exception will be returned.

If *has_holdup* is True, *material_holdup_calculation*(*t*, *p*, *j*):

$$M_{t,p,j} = \rho_{t,p,j} \times V_t \times \phi_{t,p}$$

where $\rho_{t,p,j}$ is the density of component *j* in phase *p* at time *t*

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial M_{t,p,j}}{\partial t}$, will be performed by Pyomo.DAE.

If *has_rate_reactions* is True,,
rate_reaction_stoichiometry_constraint(*t*, *p*, *j*):

$$N_{kinetic,t,p,j} = \alpha_{r,p,j} \times X_{kinetic,t,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

If *has_equilibrium_reactions* argument is True,
equilibrium_reaction_stoichiometry_constraint(*t*, *p*, *j*):

$$N_{equilibrium,t,p,j} = \alpha_{r,p,j} \times X_{equilibrium,t,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

add_total_element_balances

Material balances are written for each element in the mixture.

Variables

Variable Name	Symbol	Indices	Conditions
element_holdup	$M_{t,e}$	t, e	has_holdup = True
phase_fraction	$\phi_{t,p}$	t, p	has_holdup = True
element_accumulation	$\frac{\partial M_{t,e}}{\partial t}$	t, e	dynamic = True
elemental_mass_transfer_term	$N_{transfer,t,e}$	t, e	has_mass_transfer = True

Expressions

elemental_flow_in(t, p, e):

$$F_{in,t,p,e} = \sum_j F_{in,t,p,j} \times n_{j,e}$$

elemental_flow_out(t, p, e):

$$F_{out,t,p,e} = \sum_j F_{out,t,p,j} \times n_{j,e}$$

where $n_{j,e}$ is the number of moles of element e in component j .

Constraints

element_balances(t, e):

$$\frac{\partial M_{t,e}}{\partial t} = \sum_p F_{in,t,p,e} - \sum_p F_{out,t,p,e} + \sum_p N_{transfer,t,e} + N_{custom,t,e}$$

The $N_{custom,t,e}$ term allows the user to provide custom terms (variables or expressions) which will be added into the material balances.

If *has_holdup* is True, *elemental_holdup_calculation*(t, e):

$$M_{t,e} = V_t \times \sum_{p,j} \phi_{t,p} \times \rho_{t,p,j} \times n_{j,e}$$

where $\rho_{t,p,j}$ is the density of component j in phase p at time t

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial M_{t,e}}{\partial t}$, will be performed by Pyomo.DAE.

add_total_enthalpy_balances

A single enthalpy balance is written for the entire mixture.

Variables

Variable Name	Symbol	Indices	Conditions
energy_holdup	$E_{t,p}$	t, p	has_holdup = True
phase_fraction	$\phi_{t,p}$	t, p	has_holdup = True
energy_accumulation	$\frac{\partial E_{t,p}}{\partial t}$	t, p	dynamic = True
heat	Q_t	t	has_heat_transfer = True
work	W_t	t	has_work_transfer = True
enthalpy_transfer	$H_{transfer,t}$	t	has_enthalpy_transfer = True

Expressions

heat_of_reaction(t):

$$Q_{rxn,t} = \sum_r X_{kinetic,t,r} \times \Delta H_{rxn,r} + \sum_r X_{equilibrium,t,r} \times \Delta H_{rxn,r}$$

where $Q_{rxn,t}$ is the total enthalpy released by both kinetic and equilibrium reactions, and $\Delta H_{rxn,r}$ is the specific heat of reaction for reaction r .

Parameters

Parameter Name	Symbol	Default Value
scaling_factor_energy	s_{energy}	1E-6

Constraints

enthalpy_balance(t):

$$s_{energy} \times \sum_p \frac{\partial E_{t,p}}{\partial t} = s_{energy} \times \sum_p H_{in,t,p} - s_{energy} \times \sum_p H_{out,t,p} + s_{energy} \times Q_t + s_{energy} \times W_t + s_{energy} \times H_{transfer,t} +$$

The $E_{custom,t}$ term allows the user to provide custom terms which will be added into the energy balance.

If *has_holdup* is True, *enthalpy_holdup_calculation(t, p)*:

$$E_{t,p} = u_{t,p} \times V_t \times \phi_{t,p}$$

where $u_{t,p}$ is the internal energy density (specific internal energy) of phase p at time t

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial E_{t,p}}{\partial t}$, will be performed by Pyomo.DAE.

add_total_pressure_balances

A single pressure balance is written for the entire mixture.

Variables

Variable Name	Symbol	Indices	Conditions
deltaP	ΔP_t	t	has_pressure_change = True

Parameters

Parameter Name	Symbol	Default Value
scaling_factor_pressure	$s_{pressure}$	1E-4

Constraints

pressure_balance(t):

$$0 = s_{pressure} \times P_{in,t} - s_{pressure} \times P_{out,t} + s_{pressure} \times \Delta P_t + s_{pressure} \times \Delta P_{custom,t}$$

The $\Delta P_{custom,t}$ term allows the user to provide custom terms which will be added into the pressure balance.

1D Control Volume Class

Contents

- [1D Control Volume Class](#)
- [ControlVolume1DBlock Equations](#)

The ControlVolume1DBlock block is used for systems with one spatial dimension where material flows parallel to the spatial domain. Examples of these types of unit operations include plug flow reactors and pipes. ControlVolume1DBlock blocks are discretized along the length domain and contain one StateBlock and one ReactionBlock (if applicable) at each point in the domain (including the inlet and outlet).

```
class idaes.core.control_volume1d.ControlVolume1DBlock (*args, **kws)
```

ControlVolume1DBlock is a specialized Pyomo block for IDAES control volume blocks discretized in one spatial direction, and contains instances of ControlVolume1DBlockData.

ControlVolume1DBlock should be used for any control volume with a defined volume and distinct inlets and outlets where there is a single spatial domain parallel to the material flow direction. This encompasses unit operations such as plug flow reactors and pipes.

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic, **default** - useDefault. **Valid values:** { **useDefault** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **True** - construct holdup terms, **False** - do not construct holdup terms }

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object. }

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

auto_construct If set to True, this argument will trigger the auto_construct method which will attempt to construct a set of material, energy and momentum balance equations based on the parent unit's config block. The parent unit must have a config block which derives from CONFIG_Base, **default** - False. **Valid values:** { **True** - use automatic construction, **False** - do not use automatic construction.}

area_definition Argument defining whether area variable should be spatially variant or not. **default** - DistributedVars.uniform. **Valid values:** { DistributedVars.uniform - area does not vary across spatial domain, DistributedVars.variant - area can vary over the domain and is indexed by time and space.}

transformation_method Method to use to transform domain. Must be a method recognised by the Pyomo TransformationFactory.

transformation_scheme Scheme to use when transforming domain. See Pyomo documentation for supported schemes.

finite_elements Number of finite elements to use in transformation (equivalent to Pyomo nfe argument).

collocation_points Number of collocation points to use (equivalent to Pyomo ncp argument).

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default be-

havior of matching the BlockData index exactly to the index in initialize.

Returns (ControlVolume1DBlock) New instance

class `idaes.core.control_volume1d.ControlVolume1DBlockData` (*component*)
1-Dimensional ControlVolume Class

This class forms the core of all 1-D IDAES models. It provides methods to build property and reaction blocks, and add mass, energy and momentum balances. The form of the terms used in these constraints is specified in the chosen property package.

add_geometry (*length_domain=None*, *length_domain_set=[0.0, 1.0]*,
flow_direction=<FlowDirection.forward: 1>)
Method to create spatial domain and volume Var in ControlVolume.

Parameters

- *length_domain_set* – domain for the ControlVolume. If not provided, a new ContinuousSet will be created (default=None). ContinuousSet should be normalized to run between 0 and 1.
- *length_domain* – a new ContinuousSet if *length_domain* is not provided (default = [0.0, 1.0]).
- *flow_direction* – argument indicating direction of material flow (*flow_direction*) –

relative to length domain. Valid values:

- FlowDirection.forward (default), flow goes from 0 to 1.
- FlowDirection.backward, flow goes from 1 to 0

Returns None

add_phase_component_balances (*has_rate_reactions=False*, *has_equilibrium_reactions=False*,
has_phase_equilibrium=False, *has_mass_transfer=False*,
custom_molar_term=None, *custom_mass_term=None*)

This method constructs a set of 1D material balances indexed by time, length, phase and component.

Parameters

- **has_rate_reactions** – whether default generation terms for rate reactions should be included in material balances

- **has_equilibrium_reactions** – whether generation terms should for chemical equilibrium reactions should be included in material balances
- **has_phase_equilibrium** – whether generation terms should for phase equilibrium behaviour should be included in material balances
- **has_mass_transfer** – whether generic mass transfer terms should be included in material balances
- **custom_molar_term** – a Pyomo Expression representing custom terms to be included in material balances on a molar basis. Expression must be indexed by time, length domain, phase list and component list
- **custom_mass_term** – a Pyomo Expression representing custom terms to be included in material balances on a mass basis. Expression must be indexed by time, length domain, phase list and component list

Returns Constraint object representing material balances

add_phase_energy_balances (**args*, ***kwargs*)

Method for adding energy balances (including kinetic energy) indexed by phase to the control volume.

See specific control volume documentation for details.

add_phase_enthalpy_balances (**args*, ***kwargs*)

Method for adding enthalpy balances indexed by phase to the control volume.

See specific control volume documentation for details.

add_phase_momentum_balances (**args*, ***kwargs*)

Method for adding momentum balances indexed by phase to the control volume.

See specific control volume documentation for details.

add_phase_pressure_balances (**args*, ***kwargs*)

Method for adding pressure balances indexed by phase to the control volume.

See specific control volume documentation for details.

add_reaction_blocks (*has_equilibrium=None*)

This method constructs the reaction block for the control volume.

Parameters

- **has_equilibrium** – indicates whether equilibrium calculations will be required in reaction block
- **package_arguments** – dict-like object of arguments to be passed to reaction block as construction arguments

Returns None

add_state_blocks (*information_flow=<FlowDirection.forward: 1>*,
has_phase_equilibrium=None)

This method constructs the state blocks for the control volume.

Parameters

- **information_flow** – a FlowDirection Enum indicating whether information flows from inlet-to-outlet or outlet-to-inlet
- **has_phase_equilibrium** – indicates whether equilibrium calculations will be required in state blocks
- **package_arguments** – dict-like object of arguments to be passed to state blocks as construction arguments

Returns None

add_total_component_balances (*has_rate_reactions=False, has_equilibrium_reactions=False,*
has_phase_equilibrium=False, has_mass_transfer=False,
custom_molar_term=None, custom_mass_term=None)

This method constructs a set of 1D material balances indexed by time length and component.

Parameters

- **has_rate_reactions** – whether default generation terms for rate reactions should be included in material balances
- **has_equilibrium_reactions** – whether generation terms should for chemical equilibrium reactions should be included in material balances
- **has_phase_equilibrium** – whether generation terms should for phase equilibrium behaviour should be included in material balances

- **has_mass_transfer** – whether generic mass transfer terms should be included in material balances
- **custom_molar_term** – a Pyomo Expression representing custom terms to be included in material balances on a molar basis. Expression must be indexed by time, length domain and component list
- **custom_mass_term** – a Pyomo Expression representing custom terms to be included in material balances on a mass basis. Expression must be indexed by time, length domain and component list

Returns Constraint object representing material balances

add_total_element_balances (*has_rate_reactions=False, has_equilibrium_reactions=False, has_phase_equilibrium=False, has_mass_transfer=False, custom_elemental_term=None*)

This method constructs a set of 1D element balances indexed by time and length.

Parameters

- – **whether default generation terms for rate** (*has_rate_reactions*) – reactions should be included in material balances
- – **whether generation terms should for** (*has_equilibrium_reactions*) – chemical equilibrium reactions should be included in material balances
- – **whether generation terms should for phase** (*has_phase_equilibrium*) – equilibrium behaviour should be included in material balances
- – **whether generic mass transfer terms should be** (*has_mass_transfer*) – included in material balances
- – **a Pyomo Expression representing custom** (*custom_elemental_term*) – terms to be included in material balances on a molar elemental basis.

Expression must be indexed by time,
length and element list

Returns Constraint object representing
material balances

add_total_energy_balances (*args, **kwargs)

Method for adding a total energy balance (including kinetic energy) to the control volume.

See specific control volume documentation for details.

add_total_enthalpy_balances (*has_heat_of_reaction=False*, *has_heat_transfer=False*,
has_work_transfer=False, *has_enthalpy_transfer=False*,
custom_term=None)

This method constructs a set of 1D enthalpy balances indexed by time and phase.

Parameters

- - **whether terms for heat of reaction should**
(*has_heat_of_reaction*) - be included in enthalpy balance
- - **whether terms for heat transfer should be**
(*has_heat_transfer*) - included in enthalpy balances
- - **whether terms for work transfer should be**
(*has_work_transfer*) - included in enthalpy balances
- - **whether terms for enthalpy transfer due to**
(*has_enthalpy_transfer*) - mass transfer should be included in enthalpy balance. This should generally be the same as the *has_mas_trasnfer* argument in the material balance methods
- - **a Pyomo Expression representing custom terms to** (*custom_term*) - be included in enthalpy balances. Expression must be indexed by time, length and phase list

Returns Constraint object representing enthalpy balances

add_total_material_balances (*args, **kwargs)

Method for adding a total material balance to the control volume.

See specific control volume documentation for details.

add_total_momentum_balances (**args, **kwargs*)

Method for adding a total momentum balance to the control volume.

See specific control volume documentation for details.

add_total_pressure_balances (*has_pressure_change=False, custom_term=None*)

This method constructs a set of 1D pressure balances indexed by time.

Parameters

- **whether terms for pressure change should be** (*has_pressure_change*) – included in enthalpy balances
- **a Pyomo Expression representing custom terms to** (*custom_term*) – be included in pressure balances. Expression must be indexed by time and length domain

Returns Constraint object representing pressure balances

apply_transformation ()

Method to apply DAE transformation to the Control Volume length domain. Transformation applied will be based on the Control Volume configuration arguments.

build ()

Build method for ControlVolume1DBlock blocks.

Returns None

initialize (*state_args=None, outlvl=0, optarg={}, solver=None, hold_state=True*)

Initialization routine for 1D control volume (default solver ipopt)

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={})

- **solver** – str indicating which solver to use during initialization (default = None)
- **hold_state** – flag indicating whether the initialization routine should unfix any state variables fixed during initialization, **default** - True. **Valid values:** **True** - states variables are not unfixed, and a dict of returned containing flags for which states were fixed during initialization, **False** - state variables are unfixed after initialization by calling the `release_state` method.

Returns If `hold_states` is True, returns a dict containing flags for which states were fixed during initialization else the release state is triggered.

model_check()

This method executes the `model_check` methods on the associated state blocks (if they exist). This method is generally called by a unit model as part of the unit's `model_check` method.

Parameters None –

Returns None

release_state (*flags, outlvl=0*)

Method to release state variables fixed during initialization.

Keyword Arguments

- **flags** – dict containing information of which state variables were fixed during initialization, and should now be unfixed. This dict is returned by `initialize` if `hold_state = True`.
- **outlvl** – sets output level of logging

Returns None

report (*time_point=0, dof=False, ostream=None, prefix=""*)

No report method defined for `ControlVolume1D` class. This is due to the difficulty of presenting spatially discretized data in a readable form without plotting.

ControlVolume1DBlock Equations

This section documents the variables and constraints created by each of the methods provided by the ControlVolume0DBlock class.

- t indicates time index
- x indicates spatial (length) index
- p indicates phase index
- j indicates component index
- e indicates element index
- r indicates reaction name index

Most terms within the balance equations written by ControlVolume1DBlock are on a basis of per unit length (e.g. $\text{mol}/m \cdot s$).

add_geometry

The add_geometry method creates the normalized length domain for the control volume (or a reference to an external domain). All constraints in ControlVolume1DBlock assume a normalized length domain, with values between 0 and 1.

This method also adds variables and constraints to describe the geometry of the control volume. ControlVolume1DBlock does not support varying dimensions of the control volume with time at this stage.

Variables

Variable Name	Symbol	Indices	Conditions
length_domain	x	None	None
volume	V	None	None
area	A	None	None
length	L	None	None

Constraints

geometry_constraint:

$$V = A \times L$$

add_phase_component_balances

Material balances are written for each component in each phase (e.g. separate balances for liquid water and steam). Physical property packages may include information to indicate that certain species do not appear in all phases, and material balances will not be written in these cases (if *has_holdup* is True holdup terms will still appear for these species, however these will be set to 0).

Variables

Variable Name	Symbol	Indices	Conditions
material_holdup	$M_{t,x,p,j}$	t, x, p, j	has_holdup = True
phase_fraction	$\phi_{t,x,p}$	t, x, p	has_holdup = True
material_accumulation	$\frac{\partial M_{t,x,p,j}}{\partial t}$	t, x, p, j	dynamic = True
_flow_terms	$F_{t,x,p,j}$	t, x, p, j	None
material_flow_dx	$\frac{\partial F_{t,x,p,j}}{\partial x}$	t, x, p, j	None
rate_reaction_generation	$N_{kinetic,t,x,p,j}$	t, x, p, j	has_rate_reactions = True
rate_reaction_extent	$X_{kinetic,t,x,r}$	t, x, r	has_rate_reactions = True
equilibrium_reaction_generation	$N_{equilibrium,t,x,p,j}$	t, x, p, j	has_equilibrium_reactions = True
equilibrium_reaction_extent	$X_{equilibrium,t,x,r}$	t, x, r	has_equilibrium_reactions = True
phase_equilibrium_generation	$N_{pe,t,x,p,j}$	t, x, p, j	has_phase_equilibrium = True
mass_transfer_term	$N_{transfer,t,x,p,j}$	t, x, p, j	has_mass_transfer = True

Constraints

material_balances(t, x, p, j):

$$L \times \frac{\partial M_{t,x,p,j}}{\partial t} = fd \times \frac{\partial F_{t,x,p,j}}{\partial x} + L \times N_{kinetic,t,x,p,j} + L \times N_{equilibrium,t,x,p,j} + L \times N_{pe,t,x,p,j} + L \times N_{transfer,t,x,p,j} + L \times N_{custom,t,x,p,j}$$

fd is a flow direction term, which allows for material flow to be defined in either direction. If material flow is defined as *forward*, *fd* = -1, otherwise *fd* = 1.

The $N_{custom,t,x,p,j}$ term allows the user to provide custom terms (variables or expressions) in both mass and molar basis which will be added into the material balances, which will be converted as necessary to the same basis as the material balance (by multiplying or dividing by the component molecular weight). The basis of the material balance is determined by the physical property package, and if undefined (or not mass or mole basis), an Exception will be returned.

material_flow_linking_constraints(*t*, *x*, *p*, *j*):

This constraint is an internal constraint used to link the extensive material flow terms in the StateBlocks into a single indexed variable. This is required as Pyomo.DAE requires a single indexed variable to create the associated DerivativeVars and their numerical expansions.

If *has_holdup* is True, *material_holdup_calculation*(*t*, *x*, *p*, *j*):

$$M_{t,x,p,j} = \rho_{t,x,p,j} \times A \times \phi_{t,x,p}$$

where $\rho_{t,x,p,j}$ is the density of component *j* in phase *p* at time *t* and location *x*.

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial M_{t,x,p,j}}{\partial t}$, will be performed by Pyomo.DAE.

If *has_rate_reactions* is True, *rate_reaction_stoichiometry_constraint*(*t*, *x*, *p*, *j*):

$$N_{kinetic,t,x,p,j} = \alpha_{r,p,j} \times X_{kinetic,t,x,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

If *has_equilibrium_reactions* argument is True, *equilibrium_reaction_stoichiometry_constraint*(*t*, *x*, *p*, *j*):

$$N_{equilibrium,t,x,p,j} = \alpha_{r,p,j} \times X_{equilibrium,t,x,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

add_total_component_balances

Material balances are written for each component across all phases (e.g. one balance for both liquid water and steam). Physical property packages may include information to indicate that certain species do not appear in all phases, and material balances will not be written in these cases (if *has_holdup* is True holdup terms will still appear for these species, however these will be set to 0).

Variables

Variable Name	Symbol	Indices	Conditions
material_holdup	$M_{t,x,p,j}$	t, x, p, j	has_holdup = True
phase_fraction	$\phi_{t,x,p}$	t, x, p	has_holdup = True
material_accumulation	$\frac{\partial M_{t,x,p,j}}{\partial t}$	t, x, p, j	dynamic = True
_flow_terms	$F_{t,x,p,j}$	t, x, p, j	None
material_flow_dx	$\frac{\partial F_{t,x,p,j}}{\partial x}$	t, x, p, j	None
rate_reaction_generation	$N_{kinetic,t,x,p,j}$	t, x, p, j	has_rate_reactions = True
rate_reaction_extent	$X_{kinetic,t,x,r}$	t, x, r	has_rate_reactions = True
equilibrium_reaction_generation	$N_{equilibrium,t,x,p,j}$	t, x, p, j	has_equilibrium_reactions = True
equilibrium_reaction_extent	$X_{equilibrium,t,x,r}$	t, x, r	has_equilibrium_reactions = True
mass_transfer_term	$N_{transfer,t,x,p,j}$	t, x, p, j	has_mass_transfer = True

Constraints

material_balances(t, x, p, j):

$$L \times \sum_p \frac{\partial M_{t,x,p,j}}{\partial t} = fd \times \sum \frac{\partial F_{t,x,p,j}}{\partial x} + L \times \sum_p N_{kinetic,t,x,p,j} + L \times \sum_p N_{equilibrium,t,x,p,j} + L \times \sum_p N_{transfer,t,x,p,j} + L \times \sum_p N_{custom,t,x,p,j}$$

fd is a flow direction term, which allows for material flow to be defined in either direction. If material flow is defined as *forward*, *fd* = -1, otherwise *fd* = 1.

The $N_{custom,t,x,j}$ term allows the user to provide custom terms (variables or expressions) in both mass and molar basis which will be added into the material balances, which will be converted as necessary to the same basis as the material balance (by multiplying or dividing by the component molecular weight). The basis of the material balance is determined by the physical property package, and if undefined (or not mass or mole basis), an Exception will be returned.

material_flow_linking_constraints(*t*, *x*, *p*, *j*):

This constraint is an internal constraint used to link the extensive material flow terms in the StateBlocks into a single indexed variable. This is required as Pyomo.DAE requires a single indexed variable to create the associated DerivativeVars and their numerical expansions.

If *has_holdup* is True, *material_holdup_calculation*(*t*, *x*, *p*, *j*):

$$M_{t,x,p,j} = \rho_{t,x,p,j} \times A \times \phi_{t,x,p}$$

where $\rho_{t,x,p,j}$ is the density of component *j* in phase *p* at time *t* and location *x*.

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial M_{t,x,p,j}}{\partial t}$, will be performed by Pyomo.DAE.

If *has_rate_reactions* is True, *rate_reaction_stoichiometry_constraint*(*t*, *x*, *p*, *j*):

$$N_{kinetic,t,x,p,j} = \alpha_{r,p,j} \times X_{kinetic,t,x,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

If *has_equilibrium_reactions* argument is True, *equilibrium_reaction_stoichiometry_constraint*(*t*, *x*, *p*, *j*):

$$N_{equilibrium,t,x,p,j} = \alpha_{r,p,j} \times X_{equilibrium,t,x,r}$$

where $\alpha_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* for reaction *r* (as defined in the PhysicalParameterBlock).

add_total_element_balances

Material balances are written for each element in the mixture.

Variables

Variable Name	Symbol	Indices	Conditions
element_holdup	$M_{t,x,e}$	t, x, e	has_holdup = True
phase_fraction	$\phi_{t,x,p}$	t, x, p	has_holdup = True
element_accumulation	$\frac{\partial M_{t,x,e}}{\partial t}$	t, x, e	dynamic = True
elemental_mass_transfer_term	$N_{transfer,t,x,e}$	t, x, e	has_mass_transfer = True
elemental_flow_term	$F_{t,x,e}$	t, x, e	None

Constraints

elemental_flow_constraint(t, x, e):

$$F_{t,x,e} = \sum_p \sum_j F_{t,x,p,j} \times n_{j,e}$$

where $n_{j,e}$ is the number of moles of element e in component j .

element_balances(t, x, e):

$$L \times \frac{\partial M_{t,x,e}}{\partial t} = fd \times \frac{\partial F_{t,x,e}}{\partial x} + L \times N_{transfer,t,p,j} + L \times N_{custom,t,e}$$

fd is a flow direction term, which allows for material flow to be defined in either direction. If material flow is defined as *forward*, $fd = -1$, otherwise $fd = 1$.

The $N_{custom,t,x,e}$ term allows the user to provide custom terms (variables or expressions) which will be added into the material balances.

If *has_holdup* is True, *elemental_holdup_calculation*(t, x, e):

$$M_{t,x,e} = \rho_{t,x,p,j} \times A \times \phi_{t,x,p}$$

where $\rho_{t,x,p,j}$ is the density of component j in phase p at time t and location x .

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial M_{t,x,p,j}}{\partial t}$, will be performed by Pyomo.DAE.

add_total_enthalpy_balances

A single enthalpy balance is written for the entire mixture at each point in the spatial domain.

Variables

Variable Name	Symbol	Indices	Conditions
energy_holdup	$E_{t,x,p}$	t, x, p	has_holdup = True
phase_fraction	$\phi_{t,x,p}$	t, x, p	has_holdup = True
energy_accumulation	$\frac{\partial E_{t,x,p}}{\partial t}$	t, x, p	dynamic = True
_enthalpy_flow	$H_{t,x,p}$	t, x, p	None
enthalpy_flow_dx	$\frac{\partial H_{t,x,p}}{\partial x}$	t, x, p	None
heat	$Q_{t,x}$	t, x	has_heat_transfer = True
work	$W_{t,x}$	t, x	has_work_transfer = True
enthalpy_transfer	$H_{transfer,t,x}$	t, x	has_enthalpy_transfer = True

Expressions

heat_of_reaction(t, x):

$$Q_{rxn,t,x} = \sum_r X_{kinetic,t,x,r} \times \Delta H_{rxn,r} + \sum_r X_{equilibrium,t,x,r} \times \Delta H_{rxn,r}$$

where $Q_{rxn,t,x}$ is the total enthalpy released by both kinetic and equilibrium reactions, and $\Delta H_{rxn,r}$ is the specific heat of reaction for reaction r .

Parameters

Parameter Name	Symbol	Default Value
scaling_factor_energy	s_{energy}	1E-6

Constraints

enthalpy_balance(t):

$$s_{energy} \times L \times \sum_p \frac{\partial E_{t,x,p}}{\partial t} = s_{energy} \times fd \times \sum_p \frac{\partial H_{t,x,p}}{\partial x} + s_{energy} \times L \times Q_{t,x} + s_{energy} \times L \times W_{t,x} + s_{energy} \times L \times H_{transfer,t,x}$$

fd is a flow direction term, which allows for material flow to be defined in either direction. If material flow is defined as *forward*, $fd = -1$, otherwise $fd = 1$.

The $E_{custom,t,x}$ term allows the user to provide custom terms which will be added into the energy balance.

enthalpy_flow_linking_constraints(t, x, p):

This constraint is an internal constraint used to link the extensive enthalpy flow

terms in the StateBlocks into a single indexed variable. This is required as Pyomo.DAE requires a single indexed variable to create the associated DerivativeVars and their numerical expansions.

If *has_holdup* is True, *enthalpy_holdup_calculation*(*t*, *x*, *p*):

$$E_{t,x,p} = u_{t,x,p} \times A \times \phi_{t,x,p}$$

where $u_{t,x,p}$ is the internal density (specific internal energy) of phase *p* at time *t* and location *x*.

If *dynamic* is True:

Numerical discretization of the derivative terms, $\frac{\partial E_{t,x,p}}{\partial t}$, will be performed by Pyomo.DAE.

add_total_pressure_balances

A single pressure balance is written for the entire mixture at all points in the spatial domain.

Variables

Variable Name	Symbol	Indices	Conditions
pressure	$P_{t,x}$	t, x	None
pressure_dx	$\frac{\partial P_{t,x}}{\partial x}$	t, x	None
deltaP	$\Delta P_{t,x}$	t, x	has_pressure_change = True

Parameters

Parameter Name	Symbol	Default Value
scaling_factor_pressure	$s_{pressure}$	1E-4

Constraints

pressure_balance(*t*, *x*):

$$0 = s_{pressure} \times fd \times \frac{\partial P_{t,x}}{\partial x} + s_{pressure} \times L \times \Delta P_{t,x} + s_{pressure} \times L \times \Delta P_{custom,t,x}$$

fd is a flow direction term, which allows for material flow to be defined in either direction. If material flow is defined as *forward*, *fd* = -1, otherwise *fd* = 1.

The $\Delta P_{custom,t,x}$ term allows the user to provide custom terms which will be added into the pressure balance.

pressure_linking_constraint(t, x):

This constraint is an internal constraint used to link the pressure terms in the StateBlocks into a single indexed variable. This is required as Pyomo.DAE requires a single indexed variable to create the associated DerivativeVars and their numerical expansions.

Physical Property Package Classes

Contents

- *Physical Property Package Classes*
- *Physical Parameter Blocks*
- *State Blocks*

Physical property packages represent a collection of calculations necessary to determine the state properties of a given material. Property calculations form a critical part of any process model, and thus property packages form the core of the IDAES modeling framework.

Physical property packages consist of two parts:

- PhysicalParameterBlocks, which contain a set of parameters associated with the specific material(s) being modeled, and
- StateBlocks, which contain the actual calculations of the state variables and functions.

Physical Parameter Blocks

Physical Parameter blocks serve as a central location for linking to a property package, and contain all the parameters and indexing sets used by a given property package.

PhysicalParameterBlock Class

The role of the `PhysicalParameterBlock` class is to set up the references required by the rest of the IDAES framework for constructing instances of `StateBlocks` and attaching these to the `PhysicalParameter` block for ease of use. This allows other models to be pointed to the `PhysicalParameter` block in order to collect the necessary information and to construct the necessary `StateBlocks` without the need for the user to do this manually.

Physical property packages form the core of any process model in the IDAES modeling framework, and are used by all of the other modeling components to inform them of what needs to be constructed. In order to do this, the IDAES modeling framework looks for a number of attributes in the `PhysicalParameter` block which are used to inform the construction of other components.

- `state_block_class` - a pointer to the associated class that should be called when constructing `StateBlocks`. This should only be set by the property package developer.
- `phase_list` - a Pyomo Set object defining the valid phases of the mixture of interest.
- `component_list` - a Pyomo Set defining the names of the chemical species present in the mixture.
- `element_list` - (optional) a Pyomo Set defining the names of the chemical elements that make up the species within the mixture. This is used when doing elemental material balances.
- `element_comp` - (optional) a dict-like object which defines the elemental composition of each species in `component_list`. Form: `component: {element_1: value, element_2: value, ...}`.
- `supported_properties_metadata` - a list of supported physical properties that the property package supports, along with instruction to the framework on how to construct the associated variables and

constraints, and the units of measurement used for the property. This information is set using the `add_properties` attribute of the `define_metadata` class method.

Physical Parameter Configuration Arguments

Physical Parameter blocks have one standard configuration argument:

- `default_arguments` - this allows the user to provide a set of default values for construction arguments in associated StateBlocks, which will be passed to all StateBlocks when they are constructed.

class `idaes.core.property_base.PhysicalParameterBlock` (*component*)

This is the base class for thermophysical parameter blocks. These are blocks that contain a set of parameters associated with a specific thermophysical property package, and are linked to by all instances of that property package.

build ()

General build method for Property-ParameterBlocks. Inheriting models should call `super().build`.

Parameters None –

Returns None

build_state_block (**args, **kwargs*)

Methods to construct a StateBlock associated with this PhysicalParameterBlock. This will automatically set the parameters construction argument for the StateBlock.

Returns StateBlock

get_component (*comp*)

Method to retrieve a Component object based on a name from the `component_list`.

Parameters `comp` – name of Component object to retrieve

Returns Component object

get_default_scaling (*attribute, index=None*)

Returns a default scale factor for a property

Parameters

- **attribute** – property attribute name
- **index** – optional index for indexed properties

Returns None

get_phase (*phase*)

Method to retrieve a Phase object based on a name from the phase_list.

Parameters **phase** – name of Phase object to retrieve

Returns Phase object

get_phase_component_set ()

Method to get phase-component set for property package. If a phase-component set has not been constructed yet, this method will construct one.

Parameters None –

Returns Phase-Component Set object

set_default_scaling (*attribute, value, index=None*)

Set a default scaling factor for a property.

Parameters

- **attribute** – property attribute name
- **value** – default scaling factor
- **index** – for indexed properties, if this is not provided the scaling factor default applies to all indexed elements where specific indexes are not specifically specified.

Returns None

unset_default_scaling (*attribute, index=None*)

Remove a previously set default value

Parameters

- **attribute** – property attribute name
- **index** – optional index for indexed properties

Returns None

State Blocks

State Blocks are used within all IDAES Unit models (generally within ControlVolume Blocks) in order to calculate physical properties given the state of the material. State Blocks are notably different to other types of Blocks within IDAES as they are always indexed by time (and possibly space as well). There are two base Classes associated with State Blocks:

- `StateBlockData` forms the base class for all `StateBlockData` objects, which contain the instructions on how to construct each instance of a State Block.
- `StateBlock` is used for building classes which contain methods to be applied to sets of Indexed State Blocks (or to a subset of these). See the documentation on `declare_process_block_class` and the IDAES tutorials and examples for more information.

State Block Construction Arguments

State Blocks have the following construction arguments:

- `parameters` - a reference to the associated Physical Parameter block which will be used to make references to all necessary parameters.
- `defined_state` - this argument indicates whether the State Block should expect the material state to be fully defined by another part of the flowsheet (such as by an upstream unit operation). This argument is used to determine whether constraints such as sums of mole fractions should be enforced.
- `has_phase_equilibrium` - indicates whether the associated Control Volume or Unit model expects phase equilibrium to be enforced (if applicable).

Constructing State Blocks

State Blocks can be constructed directly from the associated Physical Parameter Block by calling the *build_state_block()* method on the Physical Parameter Block. The *parameters* construction argument will be automatically set, and any other arguments (including indexing sets) may be provided to the *build_state_block* method as usual.

StateBlockData Class

StateBlockData contains the code necessary for implementing the as needed construction of variables and constraints.

class `idaes.core.property_base.StateBlockData (*args, **kwargs)`

This is the base class for state block data objects. These are blocks that contain the Pyomo components associated with calculating a set of thermophysical and transport properties for a given material.

build()

General build method for StateBlock-Datas.

Parameters None –

Returns None

calculate_bubble_point_pressure (*args, **kwargs)

Method which computes the bubble point pressure for a multi- component mixture given a temperature and mole fraction.

calculate_bubble_point_temperature (*args, **kwargs)

Method which computes the bubble point temperature for a multi- component mixture given a pressure and mole fraction.

calculate_dew_point_pressure (*args, **kwargs)

Method which computes the dew point pressure for a multi- component mixture given a temperature and mole fraction.

calculate_dew_point_temperature (*args, **kwargs)

Method which computes the dew point temperature for a multi- component

mixture given a pressure and mole fraction.

define_display_vars()

Method used to specify components to use to generate stream tables and other outputs. Defaults to `define_state_vars`, and developers should overload as required.

define_port_members()

Method used to specify components to populate Ports with. Defaults to `define_state_vars`, and developers should overload as required.

define_state_vars()

Method that returns a dictionary of state variables used in property package. Implement a placeholder method which returns an Exception to force users to overload this.

get_energy_density_terms(*args, **kwargs)

Method which returns a valid expression for enthalpy density to use in the energy balances.

get_energy_diffusion_terms(*args, **kwargs)

Method which returns a valid expression for energy diffusion to use in the energy balances.

get_enthalpy_flow_terms(*args, **kwargs)

Method which returns a valid expression for enthalpy flow to use in the energy balances.

get_material_density_terms(*args, **kwargs)

Method which returns a valid expression for material density to use in the material balances .

get_material_diffusion_terms(*args, **kwargs)

Method which returns a valid expression for material diffusion to use in the material balances.

get_material_flow_basis(*args, **kwargs)

Method which returns an Enum indicating the basis of the material flow term.

get_material_flow_terms(*args, **kwargs)

Method which returns a valid expression for material flow to use in the material balances.

is_property_constructed(attr)

Returns True if the attribute `attr` already exists, or false if it would be added in `__getattr__`, or does not exist.

Parameters `attr` (*str*) – Attribute name to check

Returns True if the attribute is already constructed, False otherwise

lock_attribute_creation_context ()

Returns a context manager that does not allow attributes to be created while in the context and allows attributes to be created normally outside the context.

StateBlock Class

class `idaes.core.property_base.StateBlock` (*args, **kws)

This is the base class for state block objects. These are used when constructing the SimpleBlock or IndexedBlock which will contain the PropertyData objects, and contains methods that can be applied to multiple StateBlockData objects simultaneously.

initialize (*args, **kwargs)

This is a default initialization routine for StateBlocks to ensure that a routine is present. All StateBlockData classes should overload this method with one suited to the particular property package

Parameters None –

Returns None

report (index=0, true_state=False, dof=False, ostream=None, prefix="")

Default report method for StateBlocks. Returns a Block report populated with either the display or state variables defined in the StateBlockData class.

Parameters

- **index** – tuple of Block indices indicating which point in time (and space if applicable) to report state at.
- **true_state** – whether to report the display variables (False default) or the actual state variables (True)
- **dof** – whether to show local degrees of freedom in the report (default=False)

- **ostream** – output stream to write report to
- **prefix** – string to append to the beginning of all output lines

Returns Printed output to ostream

Reaction Property Package Classes

Contents

- *Reaction Property Package Classes*
- *Consistency with Thermophysical Properties*
- *Reaction Parameter Blocks*
- *Reaction Blocks*

Reaction property packages represent a collection of calculations necessary to determine the reaction behavior of a mixture at a given state. Reaction properties depend upon the state and physical properties of the material, and thus must be linked to a StateBlock which provides the necessary state and physical property information.

Reaction property packages consist of two parts:

- ReactionParameterBlocks, which contain a set of parameters associated with the specific reaction(s) being modeled, and
- ReactionBlocks, which contain the actual calculations of the reaction behavior.

Consistency with Thermophysical Properties

Within the IDAES modeling framework, all reaction packages are coupled with a thermophysical property package. The thermophysical property package contains the state variables required for calculating reaction properties, and in some cases may also provide thermophysical properties required by reaction

calculations. Due to this, reaction packages must be consistent with the thermophysical property package they are linked to and the modeling framework performs some checks to ensure this. Notably, the default units of measurement defined for the reaction package and the thermophysical property package must match.

Reaction Parameter Blocks

Reaction Parameter blocks serve as a central location for linking to a reaction property package, and contain all the parameters and indexing sets used by a given reaction package.

ReactionParameterBlock Class

The role of the ReactionParameterBlock class is to set up the references required by the rest of the IDAES framework for constructing instances of ReactionBlocks and attaching these to the ReactionParameter block for ease of use. This allows other models to be pointed to the ReactionParameter block in order to collect the necessary information and to construct the necessary ReactionBlocks without the need for the user to do this manually.

Reaction property packages are used by all of the other modeling components to inform them of what needs to be constructed when dealing with chemical reactions. In order to do this, the IDAES modeling framework looks for a number of attributes in the ReactionParameter block which are used to inform the construction of other components.

- `reaction_block_class` - a pointer to the associated class that should be called when constructing ReactionBlocks. This should only be set by the property package developer.
- `phase_list` - a Pyomo Set object defining the valid phases of the mixture of interest.

- `component_list` - a Pyomo Set defining the names of the chemical species present in the mixture.
- `rate_reaction_idx` - a Pyomo Set defining a list of names for the kinetically controlled reactions of interest.
- `rate_reaction_stoichiometry` - a dict-like object defining the stoichiometry of the kinetically controlled reactions. Keys should be tuples of (`rate_reaction_idx`, `phase_list`, `component_list`) and values equal to the stoichiometric coefficient for that index.
- `equilibrium_reaction_idx` - a Pyomo Set defining a list of names for the equilibrium controlled reactions of interest.
- `equilibrium_reaction_stoichiometry` - a dict-like object defining the stoichiometry of the equilibrium controlled reactions. Keys should be tuples of (`equilibrium_reaction_idx`, `phase_list`, `component_list`) and values equal to the stoichiometric coefficient for that index.
- `supported properties metadata` - a list of supported reaction properties that the property package supports, along with instruction to the framework on how to construct the associated variables and constraints, and the units of measurement used for the property. This information is set using the `add_properties` attribute of the `define_metadata` class method.
- `required properties metadata` - a list of physical properties that the reaction property calculations depend upon, and must be supported by the associated StateBlock. This information is set using the `add_required_properties` attribute of the `define_metadata` class method.

Reaction Parameter Configuration Arguments

Reaction Parameter blocks have two standard configuration arguments:

- `property_package` - a pointer to a `PhysicalParameterBlock` which will be used to construct the `StateBlocks` to which associated `ReactionBlocks` will be linked. Reaction property packages must be tied to a single Physical property package, and this is used to validate the connections made later when constructing `ReactionBlocks`.
- `default_arguments` - this allows the user to provide a set of default values for construction arguments in associated `ReactionBlocks`, which will be passed to all `ReactionBlocks` when they are constructed.

class `idaes.core.reaction_base.ReactionParameterBlock(*args, **kwargs)`

This is the base class for reaction parameter blocks. These are blocks that contain a set of parameters associated with a specific reaction package, and are linked to by all instances of that reaction package.

build()

General build method for `ReactionParameterBlocks`. Inheriting models should call `super().build`.

Parameters None –

Returns None

build_reaction_block(*args, **kwargs)

Methods to construct a `ReactionBlock` associated with this `ReactionParameterBlock`. This will automatically set the parameters construction argument for the `ReactionBlock`.

Returns `ReactionBlock`

Reaction Blocks

Reaction Blocks are used within IDAES Unit models (generally within ControlVolume Blocks) in order to calculate reaction properties given the state of the material (provided by an associated StateBlock). Reaction Blocks are notably different to other types of Blocks within IDAES as they are always indexed by time (and possibly space as well), and are also not fully self contained (in that they depend upon the associated state block for certain variables). There are two base Classes associated with Reaction Blocks:

- ReactionBlockDataBase forms the base class for all ReactionBlockData objects, which contain the instructions on how to construct each instance of a Reaction Block.
- ReactionBlockBase is used for building classes which contain methods to be applied to sets of Indexed Reaction Blocks (or to a subset of these). See the documentation on `declare_process_block_class` and the IDAES tutorials and examples for more information.

Reaction Block Construction Arguments

Reaction Blocks have the following construction arguments:

- `parameters` - a reference to the associated Reaction Parameter block which will be used to make references to all necessary parameters.
- `state_block` - a reference to the associated StateBlock which will provide the necessary state and physical property information.
- `has_equilibrium` - indicates whether the associated Control Volume or Unit model expects chemical equilibrium to be enforced (if applicable).

Constructing Reaction Blocks

Reaction Blocks can be constructed directly from the associated Reaction Parameter Block by calling the *build_reaction_block()* method on the Reaction Parameter Block. The *parameters* construction argument will be automatically set, and any other arguments (including indexing sets) may be provided to the *build_reaction_block* method as usual.

ReactionBlockDataBase Class

ReactionBlockDataBase contains the code necessary for implementing the as needed construction of variables and constraints.

```
class idaes.core.reaction_base.ReactionBlockDataBase(*args, **kwargs)
```

This is the base class for reaction block data objects. These are blocks that contain the Pyomo components associated with calculating a set of reaction properties for a given material.

build()

General build method for Property-BlockDatas. Inheriting models should call *super().build*.

Parameters None –

Returns None

get_reaction_rate_basis()

Method which returns an Enum indicating the basis of the reaction rate term.

is_property_constructed(attr)

Returns True if the attribute *attr* already exists, or false if it would be added in *__getattr__*, or does not exist.

Parameters *attr* (*str*) – Attribute name to check

Returns True if the attribute is already constructed, False otherwise

lock_attribute_creation_context()

Returns a context manager that does not allow attributes to be created while

in the context and allows attributes to be created normally outside the context.

ReactionBlockBase Class

class `idaes.core.reaction_base.ReactionBlockBase(*args, **kws)`

This is the base class for reaction block objects. These are used when constructing the SimpleBlock or IndexedBlock which will contain the PropertyData objects, and contains methods that can be applied to multiple ReactionBlockData objects simultaneously.

initialize(*args)

This is a default initialization routine for ReactionBlocks to ensure that a routine is present. All ReactionBlockData classes should overload this method with one suited to the particular reaction package

Parameters None –

Returns None

Unit Model Class

The UnitModelBlock class is designed to form the basis of all IDAES Unit Models, and contains a number of methods which are common to all Unit Models.

UnitModelBlock Construction Arguments

The UnitModelBlock class by default has only one construction argument, which is listed below. However, most models inheriting from UnitModelBlock should declare their own set of configuration arguments which contain more information on how the model should be constructed.

- **dynamic** - indicates whether the Unit model should be dynamic or steady-state, and if `dynamic = True`, the unit is declared to be a dynamic model. `dynamic` defaults to `useDefault` if not provided when instantiating the Unit model (see below for more details). It

is possible to declare steady-state Unit models as part of dynamic Flowsheets if desired, however the reverse is not true (cannot have dynamic Unit models within steady-state Flowsheets).

Collecting Time Domain

The next task of the `UnitModelBlock` class is to establish the time domain for the unit by collecting the necessary information from the parent Flowsheet model. If the dynamic construction argument is set to *useDefault* then the Unit model looks to its parent model for the dynamic argument, otherwise the value provided at construction is used.

Finally, if the Unit model has a construction argument named “has_holdup” (not part of the base class), then this is checked to ensure that if `dynamic = True` then `has_holdup` is also `True`. If this check fails then a `ConfigurationError` exception will be thrown.

Modeling Support Methods

The `UnitModelBlock` class also contains a number of methods designed to facilitate the construction of common components of a model, and these are described below.

Build Inlets Method

All (or almost all) Unit Models will have inlets and outlets which allow material to flow in and out of the unit being modeled. In order to save the model developer from having to write the code for each inlet themselves, `UnitModelBlock` contains a method named *build_inlet_port* which can automatically create an inlet to a specified `ControlVolume` block (or linked to a specified `StateBlock`). The *build_inlet_port* method is described in more detail in the documentation below.

Build Outlets Method

Similar to *build_inlet_port*, `UnitModelBlock` also has a method named *build_outlet_port* for constructing outlets from Unit models. The *build_outlet_port* method is described in more detail in the documentation below.

Model Check Method

In order to support the IDAES Model Check tools, `UnitModelBlock` contains a simple *model_check* method which assumes a single Holdup block and calls the *model_check* method on this block. Model developers are encouraged to create their own *model_check* methods for their particular applications.

Initialization Routine

All Unit Models need to have an initialization routine, which should be customized for each Unit model. In order to ensure that all Unit models have at least a basic initialization routine, `UnitModelBlock` contains a generic initialization procedure which may be sufficient for simple models with only one Holdup Block. Model developers are strongly encouraged to write their own initialization routines rather than relying on the default method.

UnitModelBlock Classes

```
class idaes.core.unit_model.UnitModelBlockData (component)
```

This is the class for process unit operations models. These are models that would generally appear in a process flowsheet or superstructure.

```
add_inlet_port (name=None, block=None, doc=None)
```

This is a method to build inlet Port objects in a unit model and connect these to a specified control volume or state block.

The name and block arguments are optional, but must be used together. i.e.

either both arguments are provided or neither.

Keyword Arguments

- **name** – name to use for Port object (default = “inlet”).
- **block** – an instance of a ControlVolume or StateBlock to use as the source to populate the Port object. If a ControlVolume is provided, the method will use the inlet state block as defined by the ControlVolume. If not provided, method will attempt to default to an object named control_volume.
- **doc** – doc string for Port object (default = “Inlet Port”)

Returns A Pyomo Port object and associated components.

add_outlet_port (*name=None, block=None, doc=None*)

This is a method to build outlet Port objects in a unit model and connect these to a specified control volume or state block.

The name and block arguments are optional, but must be used together. i.e. either both arguments are provided or neither.

Keyword Arguments

- **name** – name to use for Port object (default = “outlet”).
- **block** – an instance of a ControlVolume or StateBlock to use as the source to populate the Port object. If a ControlVolume is provided, the method will use the outlet state block as defined by the ControlVolume. If not provided, method will attempt to default to an object named control_volume.
- **doc** – doc string for Port object (default = “Outlet Port”)

Returns A Pyomo Port object and associated components.

add_port (*name=None, block=None, doc=None*)

This is a method to build Port objects in a unit model and connect these to a specified StateBlock.

Keyword Arguments

- **name** – name to use for Port object.
- **block** – an instance of a StateBlock to use as the source to populate the Port object
- **doc** – doc string for Port object

Returns A Pyomo Port object and associated components.

add_state_material_balances (*balance_type, state_1, state_2*)

Method to add material balances linking two State Blocks in a Unit Model. This method is not intended to replace Control Volumes, but to automate writing material balances linking isolated State Blocks in those models where this is required.

Parameters

- – **a MaterialBalanceType Enum indicating the type** (*balance_type*) – of material balances to write
- – **first State Block to be linked by balances** (*state_1*)
–
- – **second State Block to be linked by balances** (*state_2*)
–

Returns None

build()

General build method for UnitModel-BlockData. This method calls a number of sub-methods which automate the construction of expected attributes of unit models.

Inheriting models should call *super().build*.

Parameters None –

Returns None

initialize (*state_args=None, outlvl=0, solver=None, optarg={}*)

This is a general purpose initialization routine for simple unit models. This method assumes a single ControlVolume block called controlVolume, and first initializes this and then attempts to solve the entire unit.

More complex models should overload this method with their own initialization

routines,

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={})
- **solver** – str indicating which solver to use during initialization (default = None, use default IDAES solver)

Returns None

`model_check()`

This is a general purpose initialization routine for simple unit models. This method assumes a single ControlVolume block called `controlVolume` and tries to call the `model_check` method of the `controlVolume` block. If an `AttributeError` is raised, the check is passed.

More complex models should overload this method with a `model_check` suited to the particular application, especially if there are multiple `ControlVolume` blocks present.

Parameters None –

Returns None

```
class idaes.core.unit_model.UnitModelBlock(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** =

`useDefault`. **Valid values:** { `useDefault` - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if `dynamic` = True, **default** - False. **Valid values:** { `useDefault` - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (UnitModelBlock) New instance

Component Class

This is a general purpose Component object, and is suitable for general cases where the user is not concerned about distinguishing solutes from solvents (`is_solute()` and `is_solvent()` will both raise *TypeError*s). This also forms the base class for all other Component types.

```
class idaes.core.components.Component (*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

valid_phase_types List of valid Phase-Types (Enums) for this Component.

elemental_composition Dict containing elemental composition in the form element : stoichiometry

henry_component Dict indicating phases in which component follows Henry's Law (keys) with values indicating form of law.

dens_mol_liq_comp Method to use to calculate liquid phase molar density

cp_mol_liq_comp Method to calculate liquid component specific heats

cp_mol_ig_comp Method to calculate ideal gas component specific heats

enth_mol_liq_comp Method to calculate liquid component molar enthalpies

enth_mol_ig_comp Method to calculate ideal gas component molar enthalpies

entr_mol_liq_comp Method to calculate liquid component molar entropies

entr_mol_ig_comp Method to calculate ideal gas component molar entropies

pressure_sat_comp Method to use to calculate saturation pressure

phase_equilibrium_form Form of phase equilibrium constraints for component

parameter_data Dict containing initialization data for parameters

_component_list_exists Internal config argument indicating whether component_list needs to be populated.

_electrolyte Internal config argument indicating whether electrolyte component_lists needs to be populated.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can

be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Component) New instance

Solute Class

The component object is suitable for species which should be treated as solutes in a *LiquidPhase*. The only difference between this and a general *Component* is that *is_solute()* returns *True* and *is_solvent()* returns *False*.

Solvent Class

The component object is suitable for species which should be treated as solvents in a *LiquidPhase*. The only difference between this and a general *Component* is that *is_solute()* returns *False* and *is_solvent()* returns *True*.

Ion Class

The *Ion* class is suitable for ionic species which appear in *LiquidPhases*. This is similar to the *Solute* class, in that *is_solute()* returns *True* and *is_solvent()* returns *False*. Additionally, *Ion* objects have a *charge* configuration argument for recording the charge on the ion (must be an integer) and do not have a *valid_phase_types* argument (as it is assumed they can only exist in *LiquidPhases*).

Note: Users are encouraged to use the *Anion* and *Cation* classes instead of the generic *Ion* class, as these validate that sign of the *charge* configuration argument.

Anion Class

The *Anion* class is suitable for anionic species (i.e. negatively charged) which appear in *LiquidPhases*. This is a subclass of *Ion*, which enforces that the sign on the *charge* configuration argument be negative.

Cation Class

The *Cation* class is suitable for cationic species (i.e. positively charged) which appear in *LiquidPhases*. This is a subclass of *Ion*, which enforces that the sign on the *charge* configuration argument be positive.

Phase Class

```
class idaes.core.phases.Phase(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

component_list List of components which are present in phase. This is used to construct the phase-component Set for the property package.

equation_of_state A valid Python class with the necessary methods for constructing the desired equation of state (or similar model).

equation_of_state_options A dict or ConfigBlock of options to be used when setting up equation of state for phase.

parameter_data Dict containing initialization data for parameters

_phase_list_exists Internal config argument indicating whether `phase_list` needs to be populated.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Phase) New instance

Phase Type Enum

In some cases, it is useful to be able to indicate a given type of phase, rather than an instance specific *Phase* class; an example would be indicating the set of valid phases for a given chemical species. In these cases, the *PhaseType Enum* can be used, which enumerates the different types of phases recognized by the IDAES framework.

The *PhaseType Enum* has the following possible values:

- *liquidPhase* (1)
- *vaporPhase* (2)
- *solidPhase* (3)

Utility Methods

Utilities for Dynamic Flowsheets

These are utility functions for working with dynamic IDAES flowsheets.

Methods

This module contains utility functions for dynamic IDAES models.

`idaes.core.util.dyn_utils.copy_non_time_indexed_values` (*fs_tgt*, *fs_src*, *copy_fixed=True*, *outlvl=0*)

Function to set the values of all variables that are not (implicitly or explicitly) indexed by time to their values in a different flowsheet.

Parameters

- **fs_tgt** – Flowsheet into which values will be copied.
- **fs_src** – Flowsheet from which values will be copied.
- **copy_fixed** – Bool marking whether or not to copy over fixed variables in the target flowsheet.
- **outlvl** – Outlevel for the IDAES logger.

Returns

None

`idaes.core.util.dyn_utils.copy_values_at_time` (*fs_tgt*, *fs_src*, *t_target*, *t_source*, *copy_fixed=True*, *outlvl=0*)

Function to set the values of all (explicitly or implicitly) time-indexed variables in a flowsheet to similar values (with the same name) but at different points in time and (potentially) in different flowsheets.

Parameters

- **fs_tgt** – Target flowsheet, whose variables' values will get set
- **fs_src** – Source flowsheet, whose variables' values will be used to set those of the target flowsheet. Could be the target flowsheet
- **t_target** – Target time point
- **t_source** – Source time point
- **copy_fixed** – Bool of whether or not to copy over fixed variables in target model
- **outlvl** – IDAES logger output level

Returns

None

`idaes.core.util.dyn_utils.deactivate_constraints_unindexed_by(b, time)`

Searches block `b` for and constraints not indexed by time and deactivates them.

Parameters

- **b** – Block to search
- **time** – Set with respect to which to find unindexed constraints

Returns List of constraints deactivated

`idaes.core.util.dyn_utils.deactivate_model_at(b, cset, pts, outlvl=0)`

Finds any block or constraint in block `b`, indexed explicitly (and not implicitly) by `cset`, and deactivates it at points specified. Implicitly indexed components are excluded because one of their parent blocks will be deactivated, so deactivating them too would be redundant.

Parameters

- **b** – Block to search
- **cset** – ContinuousSet of interest
- **pts** – Value or list of values, in ContinuousSet, to deactivate at

Returns A dictionary mapping points in `pts` to lists of component data that have been deactivated there

`idaes.core.util.dyn_utils.find_comp_in_block(tgt_block, src_block, src_comp, allow_miss=False)`

This function finds a component in a source block, then uses the same local names and indices to try to find a corresponding component in a target block. This is used when we would like to verify that a component of the same name exists in the target block, as in model predictive control where certain variables must be correlated between plant and controller model.

Parameters

- **tgt_block** – Target block that will be searched for component
- **src_block** – Source block in which the original component is located
- **src_comp** – Component whose name will be searched for in target block
- **allow_miss** – If True, will ignore attribute and key errors due to searching

for non-existent components in the target model

Returns Component with the same name in the target block

```
idaes.core.util.dyn_utils.find_comp_in_block_at_time(tgt_block, src_block, src_comp,  
time, t0, allow_miss=False)
```

This function finds a component in a source block, then uses the same local names and indices to try to find a corresponding component in a target block, with the exception of time index in the target component, which is replaced by a specified time point. This is used for validation of a component by its name in the case where blocks may differ by at most time indices, for example validating a steady-state model or a model with a different time discretization.

Parameters

- **tgt_block** – Target block that will be searched for component
- **src_block** – Source block in which the original component is located
- **src_comp** – Component whose name will be searched for in target block
- **time** – Set whose index will be replaced in the target component
- **t0** – Index of the time set that will be used in the target component
- **allow_miss** – If True, will ignore attribute and key errors due to searching for non-existent components in the target model

```
idaes.core.util.dyn_utils.fix_vars_unindexed_by(b, time)
```

Searches block *b* for variables not indexed by time and fixes them.

Parameters

- **b** – Block to search
- **time** – Set with respect to which to find unindexed variables

Returns List of variables fixed

```
idaes.core.util.dyn_utils.get_activity_dict(b)
```

Function that builds a dictionary telling whether or not each ConstraintData and BlockData object in a model is active. Uses the objects' ids as the hash.

Parameters **b** – A Pyomo Block to be searched for active components

Returns A dictionary mapping id of constraint and block data objects to a bool indicating if they are active

`idaes.core.util.dyn_utils.get_derivatives_at(b, time, pts)`

Finds derivatives with respect to time at points specified. No distinction made for multiple derivatives or mixed partials.

Parameters

- **b** – Block to search for derivatives
- **time** – ContinuousSet to look for derivatives with respect to
- **pts** – Value or list of values in time set at which to return derivatives

Returns Dictionary mapping time points to lists of derivatives at those points

`idaes.core.util.dyn_utils.get_fixed_dict(b)`

Function that builds a dictionary telling whether or not each VarData object in a model is fixed. Uses the objects' ids as the hash.

Parameters **b** – A Pyomo block to be searched for fixed variables

Returns A dictionary mapping id of VarData objects to a bool indicating if they are fixed

`idaes.core.util.dyn_utils.get_implicit_index_of_set(comp, wrt)`

For some data object contained (at some level of the hierarchy) in a block indexed by wrt, returns the index corresponding to wrt in that block.

Parameters

- **comp** – Component data object whose (parent blocks') indices will be searched
- **wrt** – Set whose index will be searched for

Returns Value of the specified set

`idaes.core.util.dyn_utils.get_index_of_set(comp, wrt)`

For some data object of an indexed component, gets the value of the index corresponding to some 1-dimensional pyomo set.

Parameters

- **comp** – Component data object whose index will be searched
- **wrt** – Set whose index will be searched for

Returns Value of the specified set in the component data object

`idaes.core.util.dyn_utils.get_location_of_coordinate_set(setprod, subset)`

For a SetProduct and some 1-dimensional coordinate set of that SetProduct, returns the location of an index of the coordinate set within the index of the setproduct.

Parameters

- **setprod** – SetProduct containing the subset of interest
- **subset** – 1-dimensional set whose location will be found in the SetProduct

Returns Integer location of the subset within the SetProduct

`idaes.core.util.dyn_utils.path_from_block(comp, blk, include_comp=False)`

Returns a list of tuples with (local_name, index) pairs required to locate comp from blk

Parameters

- **comp** – Component(Data) object to locate
- **blk** – Block(Data) to locate comp from
- **include_comp** – Bool of whether or not to include the local_name, index of the component itself

Returns A list of string, index tuples that can be used to locate comp from blk

Homotopy Meta-Solver

The IDAES homotopy meta-solver is useful for cases where a user has a feasible solution to a well-defined (i.e. square) problem at one set of conditions (i.e. value of fixed variables), and wishes to find a feasible solution to the same problem at a different set of conditions. In many situations this can be achieved by directly changing the values of the fixed variables to their

new values and solving the problem, but cases exist where this is challenging. Homotopy solvers try to find a feasible path to the new solution by taking smaller steps in the value of the fixed variables to progressively find a solution at the new point.

Note: A homotopy solver should not be considered a fix to a poorly posed or ill-conditioned problem, and users should first consider whether their problem can be reformulated for better performance.

Homotopy Routine

The IDAES homotopy routine starts from a feasible solution to the problem at the initial values for the fixed variables (v_0) and a set of target values for these (t). The routine then calculates a set of new values for the fixed variables during the first homotopy evaluation based on an initial step size s_0 such that:

$$v_1 = t \times s_0 + v_0 \times (1 - s_0)$$

The problem is then passed to Ipopt to try to find a solution at the current values for the fixed variables. Based on the success or failure of the solver step, the following occurs:

1. If the solver returns an optimal solution, the step is accepted and the solution to the current state of the model is saved (to provide a feasible point to revert to in case a future step fails). If the current meta-solver progress is 1 (i.e. it has converged to the target values), the meta-solver terminates otherwise the meta-solver progress (p_i) is then updated, $p_i = p_{i-1} + s_i$, and the size of the next homotopy step is then calculated based on an adaptive step size method such that:

$$s_{i+1} = s_i \times \left(1 + a \times \left[\frac{I_t}{I_a} - 1 \right] \right)$$

where I_a is the number of solver iterations required in the current homotopy step, I_t is the desired number of solver iterations per homotopy step (an input parameter to the homotopy routine) and a is a step size acceleration

factor (another input parameter). As such, the size of the homotopy step is adjusted to try to achieve a desired number of solver iterations per step as a proxy for difficulty in solving each step. If new step would overshoot the target values, then the step size is cut back to match the target values. The user can also specify a maximum and/or minimum size for the homotopy which can be used to limit the homotopy step.

A new set of values for the fixed variables is calculated using $v_{i+1} = t \times (p_i + s_{i+1}) + v_0 \times (1 - (p_i + s_{i+1}))$ and the process repeated.

2. If the solver fails to find an optimal solution (for any reason), the current step is rejected and solution to the previous successful step is reloaded. If the last homotopy step was equal to the minimum homotopy step size, the meta-solver terminates, otherwise, a reduced homotopy step is calculated using:

$$s_{i+1} = s_i \times c$$

where c is a step cut factor (an input parameter between 0.1 and 0.9). If the new step homotopy step is less than the minimum homotopy step size, the minimum step is used instead.

A new set of fixed variable values are then calculated and another attempt to solve the problem is made.

Possible Termination Conditions

The homotopy meta-solver has the following possible termination conditions (using the Pyomo *TerminationCondition* Enum):

- *TerminationCondition.optimal* - meta-solver successfully converged at the target values for the fixed variables.
- *TerminationCondition.other* - the meta-solver successfully converged at the target values for the fixed variables, but with regularization of during final step. Users are recommended to discard this solution.

- *TerminationCondition.minStepLength* - the meta-solver was unable to find a feasible path to the target values, as the solver failed to find a solution using the minimum homotopy step size.
- *TerminationCondition.maxEvaluations* - the meta-solver terminated due to reaching the maximum allowed number of attempted homotopy steps
- *TerminationCondition.infeasible* - could not find feasible solution to the problem at the initial values for the fixed variables.

Available Methods

IDAES Homotopy meta-solver routine.

```
idaes.core.util.homotopy.homotopy(model, variables, targets, max_solver_iterations=50,  
                                   max_solver_time=10, step_init=0.1, step_cut=0.5,  
                                   iter_target=4, step_accel=0.5, max_step=1,  
                                   min_step=0.05, max_eval=200)
```

Homotopy meta-solver routine using Ipopt as the non-linear solver. This routine takes a model along with a list of fixed variables in that model and a list of target values for those variables. The routine then tries to iteratively move the values of the fixed variables to their target values using an adaptive step size.

Parameters

- **model** – model to be solved
- **variables** – list of Pyomo Var objects to be varied using homotopy. Variables must be fixed.
- **targets** – list of target values for each variable
- **max_solver_iterations** – maximum number of solver iterations per homotopy step (default=50)
- **max_solver_time** – maximum cpu time for the solver per homotopy step (default=10)
- **step_init** – initial homotopy step size (default=0.1)
- **step_cut** – factor by which to reduce step size on failed step (default=0.5)

- **step_accel** – acceleration factor for adjusting step size on successful step (default=0.5)
- **iter_target** – target number of solver iterations per homotopy step (default=4)
- **max_step** – maximum homotopy step size (default=1)
- **min_step** – minimum homotopy step size (default=0.05)
- **max_eval** – maximum number of homotopy evaluations (both successful and unsuccessful) (default=200)

Returns

A Pyomo TerminationCondition Enum indicating

how the meta-solver terminated (see documentation)

Solver Progress [a fraction indication how far the solver progressed] from the initial values to the target values

Number of Iterations [number of homotopy evaluations before solver] terminated

Return type Termination Condition

Initialization Methods

The IDAES toolset contains a number of utility functions to assist users with initializing models.

Available Methods

This module contains utility functions for initialization of IDAES models.

`idaes.core.util.initialization.fix_state_vars(blk, state_args={})`

Method for fixing state variables within StateBlocks. Method takes an optional argument of values to use when fixing variables.

Parameters

- **blk** – An IDAES StateBlock object in which to fix the state variables
- **state_args** – a dict containing values to use when fixing state variables.

Keys must match with names used in the `define_state_vars` method, and indices of any variables must agree.

Returns A dict keyed by block index, state variable name (as defined by `define_state_variables`) and variable index indicating the fixed status of each variable before the `fix_state_vars` method was applied.

`idaes.core.util.initialization.initialize_by_time_element` (*fs, time, **kwargs*)

Function to initialize Flowsheet *fs* element-by-element along ContinuousSet *time*. Assumes sufficient initialization/correct degrees of freedom such that the first finite element can be solved immediately and each subsequent finite element can be solved by fixing differential and derivative variables at the initial time point of that finite element.

Parameters

- **fs** – Flowsheet to initialize
- **time** – Set whose elements will be solved for individually
- **solver** – Pyomo solver object initialized with user’s desired options
- **outlvl** – IDAES logger outlvl
- **ignore_dof** – Bool. If True, checks for square problems will be skipped.

Returns None

`idaes.core.util.initialization.propagate_state` (*stream, direction='forward'*)

This method propagates values between Ports along Arcs. Values can be propagated in either direction using the *direction* argument.

Parameters

- **stream** – Arc object along which to propagate values
- **direction** – direction in which to propagate values. Default = ‘forward’
Valid value: ‘forward’, ‘backward’.

Returns None

`idaes.core.util.initialization.revert_state_vars` (*blk, flags*)

Method to revert the fixed state of the state variables within an IDAES State-

Block based on a set of flags of the previous state.

Parameters

- **blk** – an IDAES StateBlock
- **flags** – a dict of bools indicating previous state with keys in the form (StateBlock index, state variable name (as defined by `define_state_vars`), var indices).

Returns None

`idaes.core.util.initialization.solve_indexed_blocks(solver, blocks, **kws)`

This method allows for solving of Indexed Block components as if they were a single Block. A temporary Block object is created which is populated with the contents of the objects in the `blocks` argument and then solved.

Parameters

- **solver** – a Pyomo solver object to use when solving the Indexed Block
- **blocks** – an object which inherits from Block, or a list of Blocks
- **kws** – a dict of arguments to be passed to the solver

Returns A Pyomo solver results object

Phase Equilibria

The IDAES toolset contains methods for generating and displaying phase equilibria data.

Available Methods

The phase equilibria methods include methods to calculate bubble and dew temperatures (`Txy_data()`) and include this data in a readable Class (`TXYDataClass`). This class can be used in the method `build_txy_diagrams()` to create T-x-y diagrams.

`idaes.core.util.phase_equilibria.Txy_data(model, component_1, component_2, pressure, num_points=20, temperature=298.15, print_level=0, solver_op={'tol': 1e-06})`

Function to generate T-x-y data. The

function builds a state block and extracts bubble and dew temperatures at P pressure for N number of compositions. As N is increased increase the time of the calculation will increase and create a smoother looking plot.

Parameters

- **component_1** – Component 1
- **component_2** – Component 2
- **pressure** – Pressure at which the bubble and dew temperatures will be calculates
- **temperature** – Temperature at which to initialize state block
- **num_points** – Number of data point to be calculated
- **model** – Model wit intialized Property package which contains data to calculate
- **and dew temperatures for component 1 and component 2 (bubble)** –
- **print_level** – printing level from initialization
- **solver_op** – solver options

Returns A class containing the T-x-y data

Return type (Class)

```
idaes.core.util.phase_equilibria.TXYDataClass(component_1, component_2, Punits, Tunits, pressure)
```

Write data needed for build_txy_diagrams() into a class. The class can be obtained by running Txy_data() or by assigning values to the class.

```
idaes.core.util.phase_equilibria.build_txy_diagrams(txy_data, figure_name=None, print_legend=True, include_pressure=False)
```

Parameters

- **txy_data** – Txy data class includes components bubble and dew
- **temperatures** –
- **compositions** –
- **components** –
- **pressure** –

- **units.** (*and*) –
- **figure_name** – if a figure name is included the plot will save with the name
- **figure_name.png** –
- **print_legend** (*bool*) – = If True, include legend to distinguish between Bubble and dew temperature. The default is True.
- **include_pressure** (*bool*) –
- **in legends. The default is False.** (*calculated*) –

Returns t-x-y plot

The methods also include `Txy_diagram()` which is a method that calculates the data and creates the plots automatically.

```
idaes.core.util.phase_equilibria.Txy_diagram(model, component_1, component_2, pressure, num_points=20, temperature=298.15, figure_name=None, print_legend=True, include_pressure=False, print_level=0, solver_op={'tol': 1e-06})
```

This method generates T-x-y plots. Given the components, pressure and property dictionary this function calls `Txy_data()` to generate T-x-y data and once the data has been generated calls `build_txy_diagrams()` to create a plot.

Parameters

- **component_1** – Component which composition will be plotted in x axis
- **component_2** – Component which composition will decrease in x axis
- **pressure** – Pressure at which the bubble and dew temperatures will be calculated
- **temperature** – Temperature at which to initialize state block
- **num_points** – Number of data point to be calculated
- **properties** – property package which contains parameters to calculate bubble
- **dew temperatures for the mixture of the components specified.** (*and*) –

- **figure_name** – if a figure name is included the plot will save with the name
- **figure_name.png** –
- **print_legend** (*bool*) – If True, include legend to distinguish between Bubble and dew temperature. The default is True.
- **include_pressure** (*bool*) –
- **in legends. The default is False.** (*calculated*) –
- **print_level** – printing level from initialization
- **solver_op** – solver options

Returns Plot

Model State Serialization

The IDAES framework has some utility functions for serializing the state of a Pyomo model. These functions can save and load attributes of Pyomo components, but cannot reconstruct the Pyomo objects (it is not a replacement for pickle). It does have some advantages over pickle though. Not all Pyomo models are picklable. Serialization and deserialization of the model state to/from json is more secure in that it only deals with data and not executable code. It should be safe to use the `from_json()` function with data from untrusted sources, while, unpickling an object from an untrusted source is not secure. Storing a model state using these functions is also probably more robust against Python and Python package version changes, and possibly more suitable for long-term storage of results.

Below are a few example use cases for this module.

- Some models are very complex and may take minutes to initialize. Once a model is initialized it's state can be saved. For future runs, the initialized state can be reloaded instead of rerunning the initialization procedure.

- Results can be stored for later evaluation without needing to rerun the model. These results can be archived in a data management system if needed later.
- These functions may be useful in writing initialization procedures. For example, a model may be constructed and ready to run but first it may need to be initialized. Which components are active and which variables are fixed can be stored. The initialization can change which variables are fixed and which components are active. The original state can be read back after initialization, but where only values of variables that were originally fixed are read back in. This is an easy way to ensure that whatever the initialization procedure may do, the result is exactly the same problem (with only better initial values for unfixed variables).
- These functions can be used to send and receive model data to/from JavaScript user interface components.

Examples

This section provides a few very simple examples of how to use these functions.

Example Models

This section provides some boilerplate and functions to create a couple simple test models. The second model is a little more complicated and includes suffixes.

```
from pyomo.environ import *
from_
↳ idaes.core.util import to_
↳ json, from_json, StoreSpec

def setup_model01():
    model = ConcreteModel()
    model.b = Block([1,2,3])
    a = model.
    ↳ b[1].a = Var(bounds=(-
    ↳ 100, 100), initialize=2)
    b = model.
    ↳ b[1].b = Var(bounds=(-
    ↳ 100, 100), initialize=20)
    model.b[1].c_
    ↳ = Constraint(expr=b==10*a)
```

(continues on next page)

(continued from previous page)

```

    a.fix(2)
    return model

def setup_model02():
    model = ConcreteModel()
    a = _
    ↪model.a = Param(default=1,
    ↪mutable=True)
    b = _
    ↪model.b = Param(default=2,
    ↪mutable=True)
    c = model.
    ↪c = Param(initialize=4)
    x = model.x = _
    ↪Var([1,2], initialize={1:1.
    ↪5, 2:2.5}, bounds=(-10,10))
    model.
    ↪f = Objective(expr=(x[1]_
    ↪- a)**2 + (x[2] - b)**2)
    model.
    ↪g = Constraint(expr=x[1]_
    ↪+ x[2] - c >= 0)
    model.dual_
    ↪= Suffix(direction=Suffix.
    ↪IMPORT)
    model.ipopt_zL_out_
    ↪= Suffix(direction=Suffix.
    ↪IMPORT)
    model.ipopt_zU_out_
    ↪= Suffix(direction=Suffix.
    ↪IMPORT)
    return model

```

Serialization

These examples can be appended to the boilerplate code above.

The first example creates a model, saves the state, changes a value, then reads back the initial state.

```

model = setup_model01()
to_json(model,
    ↪ fname="ex.json.gz",
    ↪ gz=True, human_read=True)
model.b[1].a = 3000.4
from_json(model, fname=
    ↪ "ex.json.gz", gz=True)
print(value(model.b[1].a))

```

2

This next example show how to save only suffixes.

```

model = setup_model02()
# Suffixes here_
↪are read back from solver,
↪ so to have suffix data,
# need to solve first
solver_
↪= SolverFactory("ipopt")
solver.solve(model)
store_
↪spec = StoreSpec.suffix()
to_json(model, fname=
↪"ex.json", wts=store_spec)
# Do something and_
↪now I want my suffixes back
from_json(model, fname=
↪"ex.json", wts=store_spec)

```

to_json

Despite the name of the `to_json` function it is capable of creating Python dictionaries, json files, gzipped json files, and json strings. The function documentation is below. A *StoreSpec* object provides the function with details on what to store and how to handle special cases of Pyomo component attributes.

```

idaes.core.util.model_serializer.to_json(o, fname=None, human_read=False, wts=None,
                                         metadata={}, gz=None, return_dict=False, re-
                                         turn_json_string=False)

```

Save the state of a model to a Python dictionary, and optionally dump it to a json file. To load a model state, a model with the same structure must exist. The model itself cannot be recreated from this.

Parameters

- **o** – The Pyomo component object to save. Usually a Pyomo model, but could also be a subcomponent of a model (usually a sub-block).
- **fname** – json file name to save model state, if None only create python dict
- **gz** – If fname is given and gv is True gzip the json file. The default is True if the file name ends with '.gz' otherwise False.
- **human_read** – if True, add indents and spacing to make the json file more

readable, if false cut out whitespace and make as compact as possible

- **metadata** – A dictionary of additional metadata to add.
- **wt** – is What To Save, this is a Store-Spec object that specifies what object types and attributes to save. If None, the default is used which saves the state of the complete model state.
- **metadata** – additional metadata to save beyond the standard format_version, date, and time.
- **return_dict** – default is False if true returns a dictionary representation
- **return_json_string** – default is False returns a json string

Returns If return_dict is True returns a dictionary serialization of the Pyomo component. If return_dict is False and return_json_string is True returns a json string dump of the dict. If fname is given the dictionary is also written to a json file. If gz is True and fname is given, writes a gzipped json file.

from_json

The `from_json` function puts data from Python dictionaries, json files, gzipped json files, and json strings back into a Pyomo model. The function documentation is below. A *StoreSpec* object provides the function with details on what to read and how to handle special cases of Pyomo component attributes.

```
idaes.core.util.model_serializer.from_json(o, sd=None, fname=None, s=None,
                                             wts=None, gz=None, root_name=None)
```

Load the state of a Pyomo component state from a dictionary, json file, or json string. Must only specify one of sd, fname, or s as a non-None value. This works by going through the model and loading the state of each sub-component of o. If the saved state contains extra information, it is ignored. If the save state doesn't contain an entry for a model component that is to be loaded an error will be raised, unless ignore_missing = True.

Parameters

- **o** – Pyomo component to for which to load state
- **sd** – State dictionary to load, if None, check fname and s
- **fname** – JSON file to load, only used if sd is None
- **s** – JSON string to load only used if both sd and fname are None
- **wt** **s** – StoreSpec object specifying what to load
- **gz** – If True assume the file specified by fname is gzipped. The default is True if fname ends with ‘.gz’ otherwise False.

Returns Dictionary with some performance information. The keys are “etime_load_file”, how long in seconds it took to load the json file “etime_read_dict”, how long in seconds it took to read models state “etime_read_suffixes”, how long in seconds it took to read suffixes

StoreSpec

StoreSpec is a class for objects that tell the `to_json()` and `from_json()` functions how to read and write Pyomo component attributes. The default initialization provides an object that would load and save attributes usually needed to save a model state. There are several other class methods that provide canned objects for specific uses. Through initialization arguments, the behavior is highly customizable. Attributes can be read or written using callback functions to handle attributes that can not be directly read or written (e.g. a variable lower bound is set by calling `setlb()`). See the class documentation below.

```
class idaes.core.util.model_serializer.StoreSpec (classes=((<class
    'pyomo.core.base.param.Param'>,
    ('_mutable', )), (<class 'pyomo.core.base.var.Var'>,
    ()), (<class 'pyomo.core.base.expression.Expression'>,
    ()), (<class 'pyomo.core.base.component.Component'>,
    ('active', )), data_classes=((<class 'pyomo.core.base.var._VarData'>,
    ('fixed', 'stale', 'value', 'lb', 'ub')), (<class 'pyomo.core.base.param._ParamData'>,
    ('value', )), (<class 'int'>,
    ('value', )), (<class 'float'>,
    ('value', )), (<class 'pyomo.core.base.expression._ExpressionData'>,
    ()), (<class 'pyomo.core.base.component.ComponentData'>,
    ('active', ))), skip_classes=(<class 'pyomo.core.base.external.ExternalFunction'>,
    <class 'pyomo.core.base.set.Set'>,
    <class 'pyomo.network.port.Port'>,
    <class 'pyomo.core.base.expression.Expression'>,
    <class 'pyomo.core.base.set.RangeSet'>),
    ignore_missing=True, suffix=True,
    suffix_filter=None)
```

A StoreSpec object tells the serializer functions what to read or write. The default settings will produce a StoreSpec configured to load/save the typical attributes required to load/save a model state.

Parameters

- **classes** – A list of classes to save. Each class is represented by a list (or tuple) containing the following elements: (1) class (compared using `isinstance`) (2) attribute list or `None`, an empty list store the object, but none of its attributes, `None` will not store objects of this class type (3) optional load filter function. The load filter function returns a list of attributes to read based on the state of an object and its saved state. The allows, for example, loading values for unfixed variables, or only loading values whoes current value is less than one. The filter function only applies to load not save. Filter functions take two

arguments (a) the object (current state) and (b) the dictionary containing the saved state of an object. More specific classes should come before more general classes. For example if an object is a HeatExchanger and a UnitModel, and HeatExchanger is listed first, it will follow the HeatExchanger settings. If UnitModel is listed first in the classes list, it will follow the UnitModel settings.

- **data_classes** – This takes the same form as the classes argument. This is for component data classes.
- **skip_classes** – This is a list of classes to skip. If a class appears in the skip list, but also appears in the classes argument, the classes argument will override skip_classes. The use for this is to specifically exclude certain classes that would get caught by more general classes (e.g. UnitModel is in the class list, but you want to exclude HeatExchanger which is derived from UnitModel).
- **ignore_missing** – If True will ignore a component or attribute that exists in the model, but not in the stored state. If false an exception will be raised for things in the model that should be loaded but aren't in the stored state. Extra items in the stored state will not raise an exception regardless of this argument.
- **suffix** – If True store suffixes and component ids. If false, don't store suffixes.
- **suffix_filter** – None to store all suffixes if suffix=True, or a list of suffixes to store if suffix=True

classmethod bound()

Returns a StoreSpec object to store variable bounds only.

get_class_attr_list(o)

Look up what attributes to save/load for an Component object. :param o: Object to look up attribute list for.

Returns A list of attributes and a filter function for object type

get_data_class_attr_list(o)

Look up what attributes to save/load for an ComponentData object. :param o:

Object to look up attribute list for.

Returns A list of attributes and a filter function for object type

classmethod `isfixed()`

Returns a StoreSpec object to store if variables are fixed.

set_read_callback (*attr, cb=None*)

Set a callback to set an attribute, when reading from json or dict.

set_write_callback (*attr, cb=None*)

Set a callback to get an attribute, when writing to json or dict.

classmethod `value()`

Returns a StoreSpec object to store variable values only.

classmethod `value_isfixed(only_fixed)`

Return a StoreSpec object to store variable values and if fixed.

Parameters `only_fixed` – Only load fixed variable values

classmethod `value_isfixed_isactive(only_fixed)`

Return a StoreSpec object to store variable values, if variables are fixed and if components are active.

Parameters `only_fixed` – Only load fixed variable values

Structure

Python dictionaries, json strings, or json files are generated, in any case the structure of the data is the same. The current data structure version is 3.

The example json below shows the top-level structure. The "top_level_component" would be the name of the Pyomo component that is being serialized. The top level component is the only place where the component name does not matter when reading the serialized data.

```
{
  "__metadata__": {
    "format_version": 3,
    "date": "2018-12-21",
    "time": "11:34:39.714323",
```

(continues on next page)

(continued from previous page)

```

        "other": {
        },
        "__performance__": {
        ↪     "n_components": 219,
        ↪     "etime_make_dict": 0.003}
        },
        "top_level_component":{
            "...": "..."
        },
    }

```

The data structure of a Pyomo component is shown below. Here "attribute_1" and "attribute_2" are just examples the actual attributes saved depend on the "wts" argument to `to_json()`. Scalar and indexed components have the same structure. Scalar components have one entry in "data" with an index of "None". Only components derived from Pyomo's `_BlockData` have a `"__pyomo_components__"` field, and components appearing there are keyed by their name. The data structure duplicates the hierarchical structure of the Pyomo model.

Suffixes store extra attributes for Pyomo components that are not stored on the components themselves. Suffixes are a Pyomo structure that comes from the AMPL solver interface. If a component is a suffix, keys in the data section are the serial integer component IDs generated by `to_json()`, and the value is the value of the suffix for the corresponding component.

```

{
    "__type_"
    ↪_: "<class 'some.class'>",
    "__id__": 0,
    "data":{
        "index_1":{
            "__type_"
            ↪_: "<usually a component_
            ↪class but for params_
            ↪could be float, int, ...>",
            "__id__": 1,

            ↪     "__pyomo_components__":{
            ↪
            ↪     "child_component_1": {

```

(continues on next page)

(continued from previous page)

```

        }
    },
    "attribute_
→1": "... could be_
→any number of attributes_
→like 'value': 1.0," ,
    "attribute_2": "...",
    }
},
"attribute_
→1": "... could be_
→any number of attributes_
→like 'active': true," ,
    "attribute_2": "...",
}

```

As a more concrete example, here is the json generated for example model 2 in *Examples*. This code can be appended to the *example boilerplate above*. To generate the example json shown.

```
model = setup_model02()
solver = SolverFactory("ipopt")
solver.solve(model)
to_json(model,
        fname="ex.json")
```

The resulting json is shown below. The top-level component in this case is given as “unknown,” because the model was not given a name. The top level object name is not needed when reading back data, since the top level object is specified in the call to `from_json()`. Types are not used when reading back data, they may have some future application, but at this point they just provide a little extra information.

```
{
  "__metadata__": {
    "format_version": 3,
    "date": "2019-01-02",
    "time": "10:22:25.833501",
    "other": {
    },
    "__performance__": {
      "n_components": 18,
      "etime_make_
→ dict": 0.000955581665039062
    }
  }
}
```

(continues on next page)

```

},
"unknown": {
  ↵
  ↪ "__type__": "<class 'pyomo.
  ↪ core.base.PyomoModel.
  ↪ ConcreteModel'>",
    "__id__": 0,
    "active": true,
    "data": {
      "None": {
        ↵
        ↪ "__type__": "<class 'pyomo.
        ↪ core.base.PyomoModel.
        ↪ ConcreteModel'>",
          "__id__": 1,
          "active": true,
        ↵
        ↪ "__pyomo_components__": {
          "a": {
            "__type__
            ↪ _": "<class 'pyomo.core.
            ↪ base.param.SimpleParam'>",
              "__id__": 2,
              "_mutable": true,
              "data": {
                "None": {
                  "__type__
                  ↪ _": "<class 'pyomo.core.
                  ↪ base.param.SimpleParam'>",
                    "__id__": 3,
                    "value": 1
                }
              }
            },
          },
          "b": {
            "__type__
            ↪ _": "<class 'pyomo.core.
            ↪ base.param.SimpleParam'>",
              "__id__": 4,
              "_mutable": true,
              "data": {
                "None": {
                  "__type__
                  ↪ _": "<class 'pyomo.core.
                  ↪ base.param.SimpleParam'>",
                    "__id__": 5,
                    "value": 2
                }
              }
            },
          },
          "c": {
            "__type__
            ↪ _": "<class 'pyomo.core.
            ↪ base.param.SimpleParam'>",
              "__id__": 6,
              "_mutable": false,
              "data": {

```

(continued from previous page)

```

        "None": {
            "__type__
↪_": "<class 'pyomo.core.
↪base.param.SimpleParam'>",
            "__id__": 7,
            "value": 4
        }
    },
    "x": {
        "__type__
↪_": "<class 'pyomo.core.
↪base.var.IndexedVar'>",
        "__id__": 8,
        "data": {
            "1": {
                "__type__
↪_": "<class 'pyomo.core.base.
↪var._GeneralVarData'>",
                "__id__": 9,
                "fixed": false,
                "stale": false,
                "value": 1.5,
                "lb": -10,
                "ub": 10
            },
            "2": {
                "__type__
↪_": "<class 'pyomo.core.base.
↪var._GeneralVarData'>",
                "__id__": 10,
                "fixed": false,
                "stale": false,
                "value": 2.5,
                "lb": -10,
                "ub": 10
            }
        }
    },
    "f": {
        "__type__": "<class_
↪_pyomo.core.base.objective.
↪SimpleObjective'>",
        "__id__": 11,
        "active": true,
        "data": {
            "None": { "__type__": "<class_
↪_pyomo.core.base.objective.
↪SimpleObjective'>",
                "__id__": 12,
                "active": true

```

(continues on next page)


```

    }
    },
    "g": {
        ↪ "__type__": "<class 'pyomo.
        ↪ core.base.constraint.
        ↪ SimpleConstraint'>",
        "___id___": 13,
        "active": true,
        "data": {
            "None": {
                ↪ "__type__": "<class 'pyomo.
                ↪ core.base.constraint.
                ↪ SimpleConstraint'>",
                "___id___": 14,
                "active": true
            }
        }
    },
    "dual": {
        ↪ "__type__": "<class 'pyomo.
        ↪ core.base.suffix.Suffix'>",
        "___id___": 15,
        "active": true,
        "data": {
            ↪ "14": 0.9999999626149493
        }
    },
    "ipopt_zL_out": {
        ↪ "__type__": "<class 'pyomo.
        ↪ core.base.suffix.Suffix'>",
        "___id___": 16,
        "active": true,
        "data": {
            ↪ "9": 2.1791814146763388e-10,
            ↪ "10": 2.004834508495852e-10
        }
    },
    "ipopt_zU_out": {
        ↪ "__type__": "<class 'pyomo.
        ↪ core.base.suffix.Suffix'>",
        "___id___": 17,
        "active": true,
        "data": {
            ↪ "9": -2.947875485096964e-10,
            ↪ "10": -3.3408951850535573e-10
        }
    }
}

```

(continued from previous page)

```
}  
    }  
  }  
}
```

Model Statistics Methods

The IDAES toolset contains a number of utility functions which are useful for quantifying model statistics such as the number of variable and constraints, and calculating the available degrees of freedom in a model. These methods can be found in `idaes.core.util.model_statistics`.

The most commonly used methods are `degrees_of_freedom` and `report_statistics`, which are described below.

Degrees of Freedom Method

The `degrees_of_freedom` method calculates the number of degrees of freedom available in a given model. The calculation is based on the number of unfixed variables which appear in active constraints, minus the number of active equality constraints in the model. Users should note that this method does not consider inequality or deactivated constraints, or variables which do not appear in active equality constraints.

```
idaes.core.util.model_statistics.degrees_of_freedom(block)
```

Method to return the degrees of freedom of a model.

Parameters `block` – model to be studied

Returns Number of degrees of freedom in `block`.

Report Statistics Method

The `report_statistics` method provides the user with a summary of the contents of their model, including the degrees of freedom and a break down of the different Variables, Constraints, Objectives, Blocks and Expressions. This method also includes numbers of deactivated components for the user to use in debugging complex models.

Note: This method only considers Pyomo components in activated Blocks. The number of deactivated Blocks is reported, but any components within these Blocks are not included.

Example Output

Model Statistics

Degrees of Freedom: 0

Total No. Variables: 52

No. Fixed Variables: 12

No. Unused Variables: 0 (Fixed: 0)

No. Variables only in Inequalities: 0
(Fixed: 0)

Total No. Constraints: 40

No. Equality Constraints: 40 (Deactivated: 0)

No. Inequality Constraints: 0 (Deactivated: 0)

No. Objectives: 0 (Deactivated: 0)

No. Blocks: 14 (Deactivated: 0)

No. Expressions: 2

`idaes.core.util.model_statistics.report_statistics(block, ostream=None)`

Method to print a report of the model statistics for a Pyomo Block

Parameters

- **block** – the Block object to report statistics from
- **ostream** – output stream for printing (defaults to `sys.stdout`)

Returns Printed output of the model statistics

Other Statistics Methods

In addition to the methods discussed above, the `model_statistics` module also contains a number of methods for quantifying model statistics which may be of use to the user in debugging models. These methods come in three types:

- Number methods (start with `number_`) return the number of components which meet a given criteria, and are useful for quickly quantifying different types of components within a model for determining where problems may exist.
- Set methods (end with `_set`) return a Pyomo `ComponentSet` containing all components which meet a given criteria. These methods are useful for determining where a problem may exist, as the `ComponentSet` indicates which components may be causing a problem.
- Generator methods (end with `_generator`) contain Python generators which return all components which meet a given criteria.

Available Methods

This module contains utility functions for reporting structural statistics of IDAES models.

```
idaes.core.util.model_statistics.activated_block_component_generator(block,  
                                                                    ctype)
```

Generator which returns all the components of a given `ctype` which exist in activated Blocks within a model.

Parameters

- **block** – model to be studied
- **ctype** – type of Pyomo component to be returned by generator.

Returns A generator which returns all components of `ctype` which appear in activated Blocks in `block`

```
idaes.core.util.model_statistics.activated_blocks_set(block)
```

Method to return a `ComponentSet` of all activated Block components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all activated Block components in block (including block itself)

```
idaes.core.util.model_statistics.activated_constraints_generator(block)
```

Generator which returns all activated Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all activated Constraint components block

```
idaes.core.util.model_statistics.activated_constraints_set(block)
```

Method to return a ComponentSet of all activated Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all activated Constraint components in block

```
idaes.core.util.model_statistics.activated_equalities_generator(block)
```

Generator which returns all activated equality Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all activated equality Constraint components block

```
idaes.core.util.model_statistics.activated_equalities_set(block)
```

Method to return a ComponentSet of all activated equality Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all activated equality Constraint components in block

```
idaes.core.util.model_statistics.activated_inequalities_generator(block)
```

Generator which returns all activated inequality Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all activated inequality Constraint components block

```
idaes.core.util.model_statistics.activated_inequalities_set(block)
```

Method to return a ComponentSet of all activated inequality Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all activated inequality Constraint components in block

`idaes.core.util.model_statistics.activated_objectives_generator(block)`
Generator which returns all activated Objective components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all activated Objective components block

`idaes.core.util.model_statistics.activated_objectives_set(block)`
Method to return a ComponentSet of all activated Objective components which appear in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all activated Objective components which appear in block

`idaes.core.util.model_statistics.active_variables_in_deactivated_blocks_set(block)`
Method to return a ComponentSet of any Var components which appear within an active Constraint but belong to a deactivated Block in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including any Var components which belong to a deactivated Block but appear in an active Constraint in block

`idaes.core.util.model_statistics.deactivated_blocks_set(block)`
Method to return a ComponentSet of all deactivated Block components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all deactivated Block components in block (including block itself)

`idaes.core.util.model_statistics.deactivated_constraints_generator(block)`
Generator which returns all deactivated Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all deactivated Constraint components block

`idaes.core.util.model_statistics.deactivated_constraints_set(block)`
Method to return a ComponentSet of

all deactivated Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all deactivated Constraint components in block

```
idaes.core.util.model_statistics.deactivated_equalities_generator(block)
```

Generator which returns all deactivated equality Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all deactivated equality Constraint components block

```
idaes.core.util.model_statistics.deactivated_equalities_set(block)
```

Method to return a ComponentSet of all deactivated equality Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all deactivated equality Constraint components in block

```
idaes.core.util.model_statistics.deactivated_inequalities_generator(block)
```

Generator which returns all deactivated inequality Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all deactivated equality Constraint components block

```
idaes.core.util.model_statistics.deactivated_inequalities_set(block)
```

Method to return a ComponentSet of all deactivated inequality Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all deactivated inequality Constraint components in block

```
idaes.core.util.model_statistics.deactivated_objectives_generator(block)
```

Generator which returns all deactivated Objective components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all deactivated Objective components block

`idaes.core.util.model_statistics.deactivated_objectives_set` (*block*)

Method to return a ComponentSet of all deactivated Objective components which appear in a model.

Parameters `block` – model to be studied

Returns A ComponentSet including all deactivated Objective components which appear in block

`idaes.core.util.model_statistics.derivative_variables_set` (*block*)

Method to return a ComponentSet of all DerivativeVar components which appear in a model. Users should note that DerivativeVars are converted to ordinary Vars when a DAE transformation is applied. Thus, this method is useful for detecting any DerivativeVars which were do transformed.

Parameters `block` – model to be studied

Returns A ComponentSet including all DerivativeVar components which appear in block

`idaes.core.util.model_statistics.expressions_set` (*block*)

Method to return a ComponentSet of all Expression components which appear in a model.

Parameters `block` – model to be studied

Returns A ComponentSet including all Expression components which appear in block

`idaes.core.util.model_statistics.fixed_unused_variables_set` (*block*)

Method to return a ComponentSet of all fixed Var components which do not appear within any activated Constraint in a model.

Parameters `block` – model to be studied

Returns A ComponentSet including all fixed Var components which do not appear within any Constraints in block

`idaes.core.util.model_statistics.fixed_variables_generator` (*block*)

Generator which returns all fixed Var components in a model.

Parameters `block` – model to be studied

Returns A generator which returns all fixed Var components block

`idaes.core.util.model_statistics.fixed_variables_in_activated_equalities_set` (*block*)

Method to return a ComponentSet of

all fixed Var components which appear within an equality Constraint in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all fixed Var components which appear within activated equality Constraints in block

`idaes.core.util.model_statistics.fixed_variables_only_in_inequalities(block)`

Method to return a ComponentSet of all fixed Var components which appear only within activated inequality Constraints in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all fixed Var components which appear only within activated inequality Constraints in block

`idaes.core.util.model_statistics.fixed_variables_set(block)`

Method to return a ComponentSet of all fixed Var components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all fixed Var components in block

`idaes.core.util.model_statistics.large_residuals_set(block, tol=1e-05, return_residual_values=False)`

Method to return a ComponentSet of all Constraint components with a residual greater than a given threshold which appear in a model.

Parameters

- **block** – model to be studied
- **tol** – residual threshold for inclusion in ComponentSet
- **return_residual_values** – boolean, if true return dictionary with residual values

Returns A ComponentSet including all Constraint components with a residual greater than tol which appear in block (if return_residual_values is false) residual_values: dictionary with constraint as key and residual (float) as value (if return_residual_values is true)

Return type large_residual_set

`idaes.core.util.model_statistics.number_activated_blocks` (*block*)

Method to return the number of activated Block components in a model.

Parameters `block` – model to be studied

Returns Number of activated Block components in block (including block itself)

`idaes.core.util.model_statistics.number_activated_constraints` (*block*)

Method to return the number of activated Constraint components in a model.

Parameters `block` – model to be studied

Returns Number of activated Constraint components in block

`idaes.core.util.model_statistics.number_activated_equalities` (*block*)

Method to return the number of activated equality Constraint components in a model.

Parameters `block` – model to be studied

Returns Number of activated equality Constraint components in block

`idaes.core.util.model_statistics.number_activated_inequalities` (*block*)

Method to return the number of activated inequality Constraint components in a model.

Parameters `block` – model to be studied

Returns Number of activated inequality Constraint components in block

`idaes.core.util.model_statistics.number_activated_objectives` (*block*)

Method to return the number of activated Objective components which appear in a model.

Parameters `block` – model to be studied

Returns Number of activated Objective components which appear in block

`idaes.core.util.model_statistics.number_active_variables_in_deactivated_blocks` (*block*)

Method to return the number of Var components which appear within an active Constraint but belong to a deactivated Block in a model.

Parameters `block` – model to be studied

Returns Number of Var components which belong to a deactivated Block but appear in an activate Constraint in block

`idaes.core.util.model_statistics.number_deactivated_blocks` (*block*)

Method to return the number of deactivated Block components in a model.

Parameters `block` – model to be studied

Returns Number of deactivated Block components in block (including block itself)

`idaes.core.util.model_statistics.number_deactivated_constraints` (*block*)

Method to return the number of deactivated Constraint components in a model.

Parameters `block` – model to be studied

Returns Number of deactivated Constraint components in block

`idaes.core.util.model_statistics.number_deactivated_equalities` (*block*)

Method to return the number of deactivated equality Constraint components in a model.

Parameters `block` – model to be studied

Returns Number of deactivated equality Constraint components in block

`idaes.core.util.model_statistics.number_deactivated_inequalities` (*block*)

Method to return the number of deactivated inequality Constraint components in a model.

Parameters `block` – model to be studied

Returns Number of deactivated inequality Constraint components in block

`idaes.core.util.model_statistics.number_deactivated_objectives` (*block*)

Method to return the number of deactivated Objective components which appear in a model.

Parameters `block` – model to be studied

Returns Number of deactivated Objective components which appear in block

`idaes.core.util.model_statistics.number_derivative_variables` (*block*)

Method to return the number of DerivativeVar components which appear in a model. Users should note that DerivativeVars are converted to ordinary Vars when a DAE transformation is applied. Thus, this method is useful for detecting any DerivativeVars which were do transformed.

Parameters `block` – model to be studied

Returns Number of DerivativeVar components which appear in block

`idaes.core.util.model_statistics.number_expressions(block)`

Method to return the number of Expression components which appear in a model.

Parameters **block** – model to be studied

Returns Number of Expression components which appear in block

`idaes.core.util.model_statistics.number_fixed_unused_variables(block)`

Method to return the number of fixed Var components which do not appear within any activated Constraint in a model.

Parameters **block** – model to be studied

Returns Number of fixed Var components which do not appear within any activated Constraints in block

`idaes.core.util.model_statistics.number_fixed_variables(block)`

Method to return the number of fixed Var components in a model.

Parameters **block** – model to be studied

Returns Number of fixed Var components in block

`idaes.core.util.model_statistics.number_fixed_variables_in_activated_equalities(block)`

Method to return the number of fixed Var components which appear within activated equality Constraints in a model.

Parameters **block** – model to be studied

Returns Number of fixed Var components which appear within activated equality Constraints in block

`idaes.core.util.model_statistics.number_fixed_variables_only_in_inequalities(block)`

Method to return the number of fixed Var components which only appear within activated inequality Constraints in a model.

Parameters **block** – model to be studied

Returns Number of fixed Var components which only appear within activated inequality Constraints in block

`idaes.core.util.model_statistics.number_large_residuals(block, tol=1e-05)`

Method to return the number Constraint components with a residual greater than

a given threshold which appear in a model.

Parameters

- **block** – model to be studied
- **tol** – residual threshold for inclusion in ComponentSet

Returns Number of Constraint components with a residual greater than tol which appear in block

`idaes.core.util.model_statistics.number_total_blocks(block)`

Method to return the number of Block components in a model.

Parameters **block** – model to be studied

Returns Number of Block components in block (including block itself)

`idaes.core.util.model_statistics.number_total_constraints(block)`

Method to return the total number of Constraint components in a model.

Parameters **block** – model to be studied

Returns Number of Constraint components in block

`idaes.core.util.model_statistics.number_total_equalities(block)`

Method to return the total number of equality Constraint components in a model.

Parameters **block** – model to be studied

Returns Number of equality Constraint components in block

`idaes.core.util.model_statistics.number_total_inequalities(block)`

Method to return the total number of inequality Constraint components in a model.

Parameters **block** – model to be studied

Returns Number of inequality Constraint components in block

`idaes.core.util.model_statistics.number_total_objectives(block)`

Method to return the number of Objective components which appear in a model

Parameters **block** – model to be studied

Returns Number of Objective components which appear in block

`idaes.core.util.model_statistics.number_unfixed_variables` (*block*)

Method to return the number of unfixed
Var components in a model.

Parameters `block` – model to be studied

Returns Number of unfixed Var components in block

`idaes.core.util.model_statistics.number_unfixed_variables_in_activated_equalities` (*block*)

Method to return the number of unfixed
Var components which appear within
activated equality Constraints in a
model.

Parameters `block` – model to be studied

Returns Number of unfixed Var components which appear within activated equality Constraints in block

`idaes.core.util.model_statistics.number_unused_variables` (*block*)

Method to return the number of Var
components which do not appear within
any activated Constraint in a model.

Parameters `block` – model to be studied

Returns Number of Var components which do not appear within any activated Constraints in block

`idaes.core.util.model_statistics.number_variables` (*block*)

Method to return the number of Var
components in a model.

Parameters `block` – model to be studied

Returns Number of Var components in block

`idaes.core.util.model_statistics.number_variables_in_activated_constraints` (*block*)

Method to return the number of Var
components that appear within active
Constraints in a model.

Parameters `block` – model to be studied

Returns Number of Var components which appear within active Constraints in block

`idaes.core.util.model_statistics.number_variables_in_activated_equalities` (*block*)

Method to return the number of Var
components which appear within
activated equality Constraints in a
model.

Parameters `block` – model to be studied

Returns Number of Var components which appear within activated equality Constraints in block

`idaes.core.util.model_statistics.number_variables_in_activated_inequalities` (*block*)

Method to return the number of Var components which appear within activated inequality Constraints in a model.

Parameters **block** – model to be studied

Returns Number of Var components which appear within activated inequality Constraints in block

`idaes.core.util.model_statistics.number_variables_near_bounds` (*block*,
tol=0.0001)

Method to return the number of all Var components in a model which have a value within tol (relative) of a bound.

Parameters

- **block** – model to be studied
- **tol** – relative tolerance for inclusion in generator (default = 1e-4)

Returns Number of components block that are close to a bound

`idaes.core.util.model_statistics.number_variables_only_in_inequalities` (*block*)

Method to return the number of Var components which appear only within activated inequality Constraints in a model.

Parameters **block** – model to be studied

Returns Number of Var components which appear only within activated inequality Constraints in block

`idaes.core.util.model_statistics.total_blocks_set` (*block*)

Method to return a ComponentSet of all Block components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Block components in block (including block itself)

`idaes.core.util.model_statistics.total_constraints_set` (*block*)

Method to return a ComponentSet of all Constraint components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Constraint components in block

`idaes.core.util.model_statistics.total_equalities_generator` (*block*)
Generator which returns all equality
Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all
equality Constraint components block

`idaes.core.util.model_statistics.total_equalities_set` (*block*)
Method to return a ComponentSet of
all equality Constraint components in a
model.

Parameters **block** – model to be studied

Returns A ComponentSet including all
equality Constraint components in
block

`idaes.core.util.model_statistics.total_inequalities_generator` (*block*)
Generator which returns all inequality
Constraint components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all in-
equality Constraint components block

`idaes.core.util.model_statistics.total_inequalities_set` (*block*)
Method to return a ComponentSet of
all inequality Constraint components in
a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all
inequality Constraint components in
block

`idaes.core.util.model_statistics.total_objectives_generator` (*block*)
Generator which returns all Objective
components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all Ob-
jective components block

`idaes.core.util.model_statistics.total_objectives_set` (*block*)
Method to return a ComponentSet of
all Objective components which appear
in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Ob-
jective components which appear in
block

`idaes.core.util.model_statistics.unfixed_variables_generator` (*block*)
Generator which returns all unfixed Var
components in a model.

Parameters **block** – model to be studied

Returns A generator which returns all unfixed Var components block

```
idaes.core.util.model_statistics.unfixed_variables_in_activated_equalities_set (block)
```

Method to return a ComponentSet of all unfixed Var components which appear within an activated equality Constraint in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all unfixed Var components which appear within activated equality Constraints in block

```
idaes.core.util.model_statistics.unfixed_variables_set (block)
```

Method to return a ComponentSet of all unfixed Var components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all unfixed Var components in block

```
idaes.core.util.model_statistics.unused_variables_set (block)
```

Method to return a ComponentSet of all Var components which do not appear within any activated Constraint in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Var components which do not appear within any Constraints in block

```
idaes.core.util.model_statistics.variables_in_activated_constraints_set (block)
```

Method to return a ComponentSet of all Var components which appear within a Constraint in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Var components which appear within activated Constraints in block

```
idaes.core.util.model_statistics.variables_in_activated_equalities_set (block)
```

Method to return a ComponentSet of all Var components which appear within an equality Constraint in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Var components which appear within activated equality Constraints in block

`idaes.core.util.model_statistics.variables_in_activated_inequalities_set` (*block*)

Method to return a ComponentSet of all Var components which appear within an inequality Constraint in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Var components which appear within activated inequality Constraints in block

`idaes.core.util.model_statistics.variables_near_bounds_generator` (*block*,
tol=0.0001,
relative=True,
skip_lb=False,
skip_ub=False)

Generator which returns all Var components in a model which have a value within tol (default: relative) of a bound.

Parameters

- **block** – model to be studied
- **tol** – (relative) tolerance for inclusion in generator (default = 1e-4)
- **relative** – Boolean, use relative tolerance (default = True)
- **skip_lb** – Boolean to skip lower bound (default = False)
- **skip_ub** – Boolean to skip upper bound (default = False)

Returns A generator which returns all Var components block that are close to a bound

`idaes.core.util.model_statistics.variables_near_bounds_set` (*block*, *tol=0.0001*,
relative=True,
skip_lb=False,
skip_ub=False)

Method to return a ComponentSet of all Var components in a model which have a value within tol (relative) of a bound.

Parameters

- **block** – model to be studied
- **tol** – relative tolerance for inclusion in generator (default = 1e-4)
- **relative** – Boolean, use relative tolerance (default = True)
- **skip_lb** – Boolean to skip lower bound (default = False)

- **skip_ub** – Boolean to skip upper bound (default = False)

Returns A ComponentSet including all Var components block that are close to a bound

`idaes.core.util.model_statistics.variables_only_in_inequalities(block)`
 Method to return a ComponentSet of all Var components which appear only within inequality Constraints in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Var components which appear only within inequality Constraints in block

`idaes.core.util.model_statistics.variables_set(block)`
 Method to return a ComponentSet of all Var components in a model.

Parameters **block** – model to be studied

Returns A ComponentSet including all Var components in block

Scaling Methods

This section describes scaling utility functions and methods.

Context

Creating well scaled models is important for increasing the efficiency and reliability of solvers. Depending on property package units of measure and process scale, variables and constraints are often badly scaled.

Scaling factors can be specified for any variable or constraint. Pyomo and many solvers support the `scaling_factor` suffix. To eliminate the possibility of defining conflicting scaling factors in various places in the model, the IDAES standard is to define the `scaling_factor` suffixes in the same block as the variable or constraint that they are scaling. This ensures that each scale factor is defined in only one place, and is organized based on the model block structure.

Scaling factors in IDAES (and Pyomo) are multiplied by the variable or constraint they scale. For example, a Pressure variable in Pa units may be expected to have a magnitude of around 10^6 for a specific process. To scale the variable to a more reasonable magnitude, the scale factor for the variable could be defined to be 1×10^{-5} .

While many scaling factors should be given good default values in the property packages, some (e.g. flow rates or material holdups) must be given scale factors by the user for a specific process model. Still other scale factors can be calculated from supplied scale factors, for example, mass balance scale factors could be determined from flow rate scale factors. To calculate scale factors, models may have a standard `calculate_scaling_factors()` method.

For much of the core IDAES framework, model constraints are automatically scaled via a simple transformation where both sides of the constraint are multiplied by a scale factor determined based on supplied variable and expression scaling factors. The goal of this is to ensure that solver tolerances are meaningful for each constraint. A constraint violation of 1×10^{-8} should be acceptable, but not too tight to achieve given machine pre-

For more specific scaling information, see the model docu-

cision limits. IDAES
 model constraints should
 conform approximately to
 this guideline after the
`calculate_scaling_factors()` method is executed. Users should follow this guideline for constraints they write. The scaling of constraints for reasonable residual tolerances is done as a constraint transformation independent of the scaling factor suffix. Scaling factors for constraints can still be set based on other methods such as reducing very large Jacobian matrix entries.

Specifying Scaling

Suffixes are used to specify scaling factors for IDAES models. These suffixes are created when needed by calling the `set_scaling_factor()` function. Using the `set_scaling_factor()`, `get_scaling_factor()`, and `unset_scaling_factor()` eliminates the need to deal directly with scaling suffixes, and ensures that scaling factors are stored in the IDAES standard location.

```
idaes.core.util.scaling.set_scaling_factor(c, v, data_objects=True)
```

Set a scaling factor for a model component. This function creates the `scaling_factor` suffix if needed.

Parameters

- **c** – component to supply scaling factor for
- **v** – scaling factor

Returns

None

```
idaes.core.util.scaling.get_scaling_factor(c, default=None, warning=False, exception=False)
```

Get a component scale factor.

Parameters

- **c** – component
- **default** – value to return if no scale factor exists (default=None)

```
idaes.core.util.scaling.unset_scaling_factor(c, data_objects=True)
```

Delete a component scaling factor.

Parameters

c – component

Returns

None

Constraint Transformation

As mentioned previously, constraints in the IDAES framework are transformed such that 1×10^{-8} is a reasonable criteria for convergence before any other scaling factors are applied. There are a few utility functions for scaling transformation of constraints. When transforming constraints with these functions, the scaling applies to the original constraint, not combined with any previous transformation.

```
idaes.core.util.scaling.constraint_scaling_transform(c, s)
```

This transforms a constraint by the argument *s*. The scaling factor applies to original constraint (e.g. if one were to call this twice in a row for a constraint with a scaling factor of 2, the original constraint would still, only be scaled by a factor of 2.)

Parameters

- **c** – Pyomo constraint
- **s** – scale factor applied to the constraint as originally written

Returns None

```
idaes.core.util.scaling.constraint_scaling_transform_undo(c)
```

The undoes the scaling transforms previously applied to a constraint.

Parameters **c** – Pyomo constraint

Returns None

```
idaes.core.util.scaling.get_constraint_transform_applied_scaling_factor(c,  
                                                                        de-  
                                                                        fault=None)
```

Get a the scale factor that was used to transform a constraint.

Parameters

- **c** – constraint data object
- **default** – value to return if no scaling factor exists (default=None)

Returns The scaling factor that has been used to transform the constraint or the default.

Calculation in Model

Some scaling factors may also be calculated by a call to a model's `calculate_scaling_factors()` method. For more information see specific model documentation.

Sometimes a scaling factor may be set on an indexed component and prorogated to it's data objects later can be useful for example in models that use the DAE transformation, not all data objects exist until after the transformation.

```
idaes.core.util.scaling.propagate_indexed_component_scaling_factors (blk,
                                                                    typ=(<class
                                                                    'py-
                                                                    omo.core.base.var.Var'>,
                                                                    <class
                                                                    'py-
                                                                    omo.core.base.constraint.Constraint'>,
                                                                    <class
                                                                    'py-
                                                                    omo.core.base.expression.Expression'>,
                                                                    over-
                                                                    write=False,
                                                                    de-
                                                                    scend_into=True)
```

Use the parent component scaling factor to set all component data object scaling factors.

Parameters

- **blk** – The block on which to search for components
- **typ** – Component type(s) (default=(Var, Constraint, Expression, Param))
- **overwrite** – if a data object already has a scaling factor should it be overwritten (default=False)
- **descend_into** – descend into child blocks (default=True)

Constraint Auto-Scaling

Constraints
can be
scaled to au-
tomati-
cally re-
duce very
large en-
tries in
the Jaco-
bian ma-
trix with
the `constraint_autoscale_large_jac()`
function.

```
idaes.core.util.scaling.constraint_autoscale_large_jac(m, ig-  
                                                         nore_constraint_scaling=False,  
                                                         ig-  
                                                         nore_variable_scaling=False,  
                                                         max_grad=100,  
                                                         min_scale=1e-06,  
                                                         no_scale=False)
```

Automatically scale constraints based on the Jacobian. This function immitates Ipopt's default constraint scaling. This scales constraints down to avoid extremely large values in the Jacobian

Parameters

- **m** – model to scale
- **ignore_constraint_scaling** – ignore existing constraint scaling
- **ignore_variable_scaling** – ignore existing variable scaling
- **max_grad** – maximum value in Jacobian after scaling, subject to minimum scaling factor restriction.
- **min_scale** – minimum scaling factor allowed, keeps constraints from being scaled too much.
- **no_scale** – just calculate the Jacobian and scaled Jacobian, don't scale anything

Inspect Scaling

Models

can be
large, so
it is of-
ten diffi-
cult to iden-
tify where
scaling
is needed
and where
the prob-
lem may be
poorly scaled.
The func-
tions be-
low may
be help-
ful in in-
specting
a mod-
els scal-
ing. Ad-
ditionally

`constraint_autoscale_large_jac()` described above can provide Jacobian information at the current variable values.

```
idaes.core.util.scaling.badly_scaled_var_generator(blk, large=10000.0, small=0.001,  
                                                    zero=1e-10, descend_into=True,  
                                                    include_fixed=False)
```

This provides a rough check for variables with poor scaling based on their current scale factors and values. For each potentially poorly scaled variable it returns the var and its current scaled value.

Parameters

- **blk** – pyomo block
- **large** – Magnitude that is considered to be too large
- **small** – Magnitude that is considered to be too small
- **zero** – Magnitude that is considered to be zero, variables with a value of zero are okay, and not reported.

Yields variable data object, current absolute value of scaled value

```
idaes.core.util.scaling.unscaled_variables_generator(blk, descend_into=True, include_fixed=False)
```

Generator for unscaled variables

Parameters block –

Yields variables with no scale factor

`idaes.core.util.scaling.unscaled_constraints_generator` (*blk, descend_into=True*)
Generator for unscaled constraints

Parameters block –

Yields constraints with no scale factor

`idaes.core.util.scaling.map_scaling_factor` (*iter, default=1, warning=False, func=<built-in function min>*)

Map `get_scaling_factor` to an iterable of Pyomo components, and call `func` on the result. This could be use, for example, to get the minimum or maximum scaling factor of a set of components.

Parameters

- **iter** – Iterable yeilding Pyomo componentes
- **default** – The default value used when a scaling factor is missing. The default is `default=1`.
- **warning** – Log a warning for missing scaling factors
- **func** – The function to call on the resulting iterable of scaling factors. The default is `min()`.

Returns The result of `func` on the set of scaling factors

`idaes.core.util.scaling.min_scaling_factor` (*iter, default=1, warning=True*)

Map `get_scaling_factor` to an iterable of Pyomo components, and get the minimum caling factor.

Parameters

- **iter** – Iterable yeilding Pyomo componentes
- **default** – The default value used when a scaling factor is missing. If `None`, this will raise an exception when scaling factors are missing. The default is `default=1`.
- **warning** – Log a warning for missing scaling factors

Returns Minimum scaling factor of the components in `iter`

Applying Scaling

Scale factor suffixes can be passed directly to a solver. How the scale factors are used may vary by solver. Pyomo also contains tools to transform a problem to a scaled version.

Ipopt is the standard solver in IDAES. To use scale factors with Ipopt, the `nlp_scaling_method` option should be set to `user-scaling`. Be aware that this deactivates any NLP automatic scaling.

Table Methods

The IDAES toolset contains a number of methods for generating and displaying summary tables of data in the form of pandas DataFrames.

Available Methods

```
idaes.core.util.tables.arcs_to_stream_dict(blk, additional=None, descend_into=True,  
                                           sort=False, prepend=None, s={})
```

Creates a stream dictionary from the Arcs in a model, using the Arc names as keys. This can be used to automate the creation of the streams dictionary needed for the `create_stream_table_dataframe()` and `stream_states_dict()` functions.

Parameters

- **blk** (*pyomo.environ._BlockData*) – Pyomo model to

search for Arcs

- **additional** (*dict*) – Additional states to add to the stream dictionary, which aren't represented by arcs in blk, for example feed or product streams without Arcs attached or states internal to a unit model.
- **descend_into** (*bool*) – If True, search subblocks for Arcs as well. The default is True.
- **sort** (*bool*) – If True sort keys and return an OrderedDict
- **prepend** (*str*) – Prepend a string to the arc name joined with a '.'. This can be useful to prevent conflicting names when sub blocks contain Arcs that have the same names when used in combination with descend_into=False.
- **s** (*dict*) – Add streams to an existing stream dict.

Returns Dictionary with Arc names as keys and the Arcs as values.

```
idaes.core.util.tables.create_stream_table_dataframe(streams, true_state=False,
                                                    time_point=0, orient='columns')
```

Method to create a stream table in the form of a pandas dataframe. Method takes a dict with name keys and stream values. Use an OrderedDict to list the streams in a specific order, otherwise the dataframe can be sorted later.

Parameters

- **streams** – dict with name keys and stream values. Names will be used as display names for stream table, and streams may be Arcs, Ports or State-Blocks.
- **true_state** – indicated whether the stream table should contain the display variables define in the StateBlock (False, default) or the state variables (True).
- **time_point** – point in the time domain at which to generate stream table (default = 0)
- **orient** – orientation of stream table. Accepted values are 'columns' (default) where streams are displayed as

columns, or 'index' where stream are displayed as rows.

Returns A pandas DataFrame containing the stream table data.

`idaes.core.util.tables.generate_table(blocks, attributes, heading=None, exception=True)`

Create a Pandas DataFrame that contains a list of user-defined attributes from a set of Blocks.

Parameters

- **blocks** (*dict*) – A dictionary with name keys and BlockData objects for values. Any name can be associated with a block. Use an OrderedDict to show the blocks in a specific order, otherwise the dataframe can be sorted later.
- **attributes** (*list or tuple of strings*) – Attributes to report from a Block, can be a Var, Param, or Expression. If an attribute doesn't exist or doesn't have a valid value, it will be treated as missing data.
- **heading** (*list or tuple of strings*) – A list of strings that will be used as column headings. If None the attribute names will be used.
- **exception** (*bool*) – If True, raise exceptions related to invalid or missing indexes. If false missing or bad indexes are ignored and None is used for the table value. Setting this to False allows tables where some state blocks have the same attributes with different indexing. (default is True)

Returns A Pandas dataframe containing a data table

Return type (DataFrame)

`idaes.core.util.tables.stream_states_dict(streams, time_point=0)`

Method to create a dictionary of state block representing stream states. This takes a dict with stream name keys and stream values.

Parameters

- **streams** – dict with name keys and stream values. Names will be used as display names for stream table, and streams may be Arcs, Ports or State-Blocks.

- **time_point** – point in the time domain at which to generate stream table (default = 0)

Returns A pandas DataFrame containing the stream table data.

`idaes.core.util.tables.stream_table_dataframe_to_string(stream_table, **kwargs)`

Method to print a stream table from a dataframe. Method takes any argument understood by DataFrame.to_string

`idaes.core.util.tables.tag_state_quantities(blocks, attributes, labels, exception=False)`

Take a stream states dictionary, and return a tag dictionary for stream quantities. This takes a dictionary (blk) that has state block labels as keys and state blocks as values. The attributes are a list of attributes to tag. If an element of the attribute list is list-like, the first element is the attribute and the remaining elements are indexes. Labels provides a list of attribute labels to be used to create the tag. Tags are blk_key + label for the attribute.

Parameters

- **blocks** (*dict*) – Dictionary of state blocks. The key is the block label to be used in the tag, and the value is a state block.
- **attributes** (*list-like*) – A list of attributes to tag. It is okay if a particular attribute does not exist in a state block. This allows you to mix state blocks with different sets of attributes. If an attribute is indexed, the attribute can be specified as a list or tuple where the first element is the attribute and the remaining elements are indexes.
- **labels** (*list-like*) – These are attribute labels. The order corresponds to the attribute list. They are used to create the tags. Tags are in the form blk.key + label.
- **exception** (*bool*) – If True, raise exceptions related to invalid or missing indexes. If false missing or bad indexes are ignored and None is used for the table value. Setting this to False allows tables where some state blocks have the same attributes with different indexing. (default is True)

Returns

Dictionary where the keys are tags and the values are model

attributes, usually Pyomo component
data objects.

Return type (`dict`)

Unit Model Costing

The IDAES Process Modeling Framework includes support for incorporating costing of unit operations into a flow-sheet to allow for calculation and optimization of process costs. Cost Correlations are implemented using unit costing sub-modules to allow users to easily develop and incorporate their own costing models.

Contents

- *Unit Model Costing*
- *Introduction*
- *Example*
- *Units*
- *IDAES Costing Module*
- * *Heat Exchanger Cost*
- * *Pressure Changer Cost*
- *Turbine Cost Model*
- *Pump Cost Model*
- *Mover (Compressor, Fan, Blower)*
- * *Fired Heater*
- * *Cost of Pressure Vessels and Towers for Distillation*
- *Vessel Cost*
- *Base Cost of Platforms and ladders*
- *Purchase Cost of Plates*

Introduction

All unit models within the core IDAES model library include a *get_costing* method which can be called to include cost correlations for an instance of that unit. The *get_costing* method for each unit takes a number of arguments used to specify the basis for costing each piece of equipment. Details are given for each unit model later in this documentation, however, all *get_costing* methods take the following two arguments:

- **module** - this argument specifies the costing module to use when constructing the constraints and associated variables. if not provided, this defaults to the standard IDAES costing module.
- **year** - this argument sets the year to which all costs should be normalized (CE index 2010 to 2019)

When *get_costing* is called on an instance of a unit model, a new sub-block is created on that unit named *costing* (i.e. *flowsheet.unit.costing*). All variables and constraints related to costing will be constructed within this new block (see detailed documentation for each unit for details on these variables and constraints).

In addition, the first time *get_costing* is called for a unit operation within a flowsheet, an additional *costing* block is created on the flowsheet object (i.e. *flowsheet.unit.costing*) in order to hold any global parameters relating to costing. The most common of these parameters is the cost normalization parameter based on the year selected by the user.

The unit costing module also contains an *initialize* method which can be used to estimate initial values for costing variables based on the current state of the associated unit model. This method can be called directly from the *unit_costing* module to initialize a specific costing block, or can be incorporated into a unit model initialization procedure. This method has been incorporated into the *initialize* method of

all the models in the core unit model library. Therefore, if `get_costing()` is called before `unit.initialize()`, the initialize method will deactivate the costing block, initialize the unit model as normal, and then activate the costing block and initialize costing block.

Note: The global paramters are created when the first instance of `get_costing` is called and use the values provided there for initialization. Subsequent `get_costing` calls use the existing paramters, and do not change the initialized values. i.e. any “year” argument provided to a `get_costing` call after the first will be ignored.

Table 1. Main Variables added to the unit block (“self.costing”).

Variable	Symbol	Units	Notes
Purchase cost	<i>purchase_cost</i>	dollars	Purchase cost
Base cost per unit	<i>base_cost_per_unit</i>	unitless	Base cost per unit
Base cost	<i>base_cost</i>	unitless	Base cost (base cost per unit * number of units)
Number of units	<i>number_of_units</i>	unitless	Number of units to be costed (to take advantage of the economics of scale)

Note: number of units by default is fixed to 1 and the user must unfix this variable to optimize the number of units. Also, *number of units* can be built as a continuous variable or an integer variable. If latest, the user must provide an mip solver. Use the global costing argument for this purpose (`integer_n_units=True` or `False`).

Example

Below is a simple example of how to add cost correlations to a flowsheet including a heat exchanger using the default IDAES costing module.

```
from pyomo.environ_
↳ import (ConcreteModel,
↳ SolverFactory)
from pyomo.util.calc_
↳ var_value import calculate_
↳ variable_from_constraint
from idaes.
↳ core import FlowsheetBlock
from idaes.
↳ generic_models.unit_models.
↳ heat_exchanger import \
    (HeatExchanger,
    ↳ HeatExchangerFlowPattern)
from idaes.generic_models.
↳ properties import iapws95
```

(continues on next page)

(continued from previous page)

```

from idaes.
↳core.util.model_statistics_
↳import degrees_of_freedom

m = ConcreteModel()
m.fs_
↳= FlowsheetBlock(default=
↳{"dynamic": False})

m.fs.properties = iapws95.
↳Iapws95ParameterBlock()

m.fs.unit_
↳= HeatExchanger(default={
↳
↳    "shell": {"property_
↳package": m.fs.properties},
↳
↳    "tube": {"property_
↳package": m.fs.properties,
↳    "flow_pattern":_
↳HeatExchangerFlowPattern.
↳countercurrent})
# set inputs
m.fs.unit.shell_inlet.flow_
↳mol[0].fix(100)      # mol/s
m.fs.unit.shell_inlet.enth_
↳mol[0].fix(3500)    # j/s
m.fs.unit.
↳shell_inlet.pressure[0].
↳fix(101325)        # Pa

m.fs.unit.tube_
↳inlet.flow_mol[0].fix(100)
m.fs.unit.tube_
↳inlet.enth_mol[0].fix(4000)
m.fs.unit.tube_inlet.
↳pressure[0].fix(101325.0)

m.fs.
↳unit.area.fix(1000)  # m2
m.fs.unit.overall_
↳heat_transfer_coefficient.
↳fix(100)            # W/m2K

m.fs.unit.
↳get_costing(module=costing,
↳length_factor='12ft')

m.fs.unit.initialize()

opt = SolverFactory('ipopt')
opt.options = {'tol
↳': 1e-6, 'max_iter': 50}
results_
↳= opt.solve(m, tee=True)

```

Units

It is important to highlight that the costing method interrogates the property package to determine the units of this model, if the user provided the correct units in the metadata dictionary (see property models for additional information), the model units will be converted to the right units. For example: in this example area is in m^2 , while the cost correlations for heat exchangers require units to be in ft^2 . Therefore, the costing method will convert the units to ft^2 . The use of Pyomo-unit conversion tools is under development.

IDAES Costing Module

A default costing module has been developed primarily based on base cost and purchase cost correlations from the following reference with some exceptions (noted in the documentation as appropriate).

Process and Product Design Principles: Synthesis, Analysis, and Evaluation. Seider, Seader, Lewin, Windagdo, 3rd Ed. John Wiley and Sons. Chapter 22. Cost Accounting and Capital Cost Estimation

Users should refer to the reference above for details of the costing correlations, however, a summary of this methods is provided below.

Table 2. Cost basis for each unit model.

Unit Model	Basis	Units
heat exchanger	<i>area</i>	ft^2
pump	<i>fluid_{work}</i>	ft^3/s
compressor	<i>mechanical_{work}</i>	hp
turbine	<i>mechanical_{work}</i>	hp
vessels	<i>DandL</i>	ft
fired heaters	<i>heat_{duty}</i>	BTU/hr

Heat Exchanger Cost

The purchase cost is computed based on the base unit cost and three correction factors (Eq. 22.43 in Seider et al.). The base cost is computed depending on the heat exchanger type selected by the user:

$$self.costing.purchase_cost = pressure_factor * material_factor * L_factor * self.costing.base_cost * (CE_{index}/500)$$

$$self.costing.base_cost_per = \exp(\alpha_1 - \alpha_2 * \log area * hx_os + \alpha_3 * (\log area * hx_os)^2)$$

$$self.costing.base_cost = self.costing.base_cost_per * self.costing.number_of_units$$

$$area = self.area / self.costing.number_of_units$$

where:

- `pressure_factor` - is the pressure design correction factor
- `material_factor` - is the construction material correction factor
- `length_factor` - is the tube length correction factor
- `CE_index` - is a global parameter for Chemical Engineering cost index for years 2010-2019
- `hx_os` - heat exchanger oversize factor (default = 1)
- `area` is a reference object and (`self.area` is the model variable)

The heat exchanger costing method has three arguments, `hx_type` = heat exchanger type, `FM_Mat` = construction material factor, and `FL` = tube length factor.

- `hx_type`: 'floating_head', 'fixed_head', 'U-tube', 'Kettle_vap'
- `material factor (Mat_factor)`: 'stain_steel', 'carb_steel'
- `tube length (length_factor)`: '8ft', '12ft', '16ft', '20ft'

where '*' corresponds to the default options, `FL` and `FM_MAT` are pyomutable parameters fixed based on user selection.

Table 3. Base cost factors for heat exchanger type.

Tube Length (ft)	α_1	α_2	α_3
floating_head	11.9052	0.8709	0.09005
fixed_head	11.2927	0.8228	0.09861
U-tube	11.3852	0.9186	0.09790
Kettle_vap	12.2052	0.8709	0.09005

Table 4. Tube-Length correction factor.

Tube Length (ft)	FL
8	1.25
12	1.12
16	1.05
20	1.00

Construction material correction factor (FM_Mat) can be computed with Eq. 22.44 (Seider et al.)

$$material_factor = a + \left(\frac{area}{100}\right)^b$$

Table 5. Materials of construction factors.

Materials of Construction		
Shell / Tube	a	b
carbon steel/carbon steel	0.00	0.00
carbon steel/brass	1.08	0.05
carbon steel/stainless steel	1.75	0.13
carbon steel/monel	2.1	0.13
carbon steel/titanium	5.2	0.16
carbon steel/Cr-Mo steel	1.55	0.05
Cr-Mo steel/Cr-Mo steel	1.7	0.07
stainless steel/stainless steel	2.7	0.07
monel/monel	3.3	0.08
titanium/titanium	9.6	0.06

Note that *Mat_factor* argument should be provided a string, for example: *Mat_factor*: 'carbon steel/carbon steel'.

Pressure Changer Cost

The costing of a pressure changer unit model is more complicated, because the pressure changer model can be imported into the flowsheet object representing a pump, turbine, compressor, or a simply pressure changer (fan, blower, etc.). The *get_costing* method currently supports costing of pumps, turbines,

and compressors. The method automatically interrogates the flowsheet object to determine if the unit is being used as a pump, turbine, or compressor.

The *get_costing* method automatically determines if the unit model is being used as a pump, turbine, or compressor based on the *compressor* and *thermodynamic_assumption* configuration arguments provided by the user where creating the unit model. A summary of the decision logic is shown below.

Unit Type	compressor	thermodynamic_assumption
Turbine	False	Any
Pump	True	pump
Mover	True	not pump

Additionally, some unit types have different sub-types which can be costed appropriately. In these cases, an additional argument is provided to *get_costing* to identify the sub-type to use which is detailed below.

Turbine Cost Model

The turbine cost is based on the mechanical work of unit (*work_mechanical*), this correlation has been obtained using the NETL Report (DOE/NETL 2015).

$$self.costing.purchase_cost = 580 * (mechanical_{work})^{0.81}$$

DOE/NETL, 2015, report. Cost and performance Baseline for Fossil Energy Plants. Volume 1a: Bituminous Coal (PC) and Natural Gas to Electricity. Revision 3

Pump Cost Model

Three subtypes are supported for costing of pumps, which can be set using the “*pump_type*” argument.

- 1) Centrifugal pumps
(*pump_type*=‘centrifugal’)
- 2) External gear pumps
(*pump_type*=‘external’)

- 3) Reciprocating Plunger pumps
(`pump_type='reciprocating'`)

Centrifugal Pump

The centrifugal pump cost has two main components, the cost of the pump and the cost of the motor. The pump cost is based on the fluid work (`work_fluid`), pump head, and size factor. Additional arguments are required:

- `pump_type_factor` = '1.4' (see Table 6)
- `pump_motor_type_factor` = 'open', 'enclosed', 'explosion_proof'

Based on user's inputs the `get_costing` method builds `base_cost` and `purchase_cost` for both the pump and the motor. The unit purchase cost is obtained by adding the motor and pump costs.

$$self.costing.purchase_cost = self.costing.pump_purchase_cost + self.costing.motor_purchase_cost$$

To compute the purchase cost of the centrifugal pump, first we obtain the pump size factor (S) with Eq. 22.13, then we obtain the base cost with Eq. 22.14. Finally, the purchase cost of the pump is obtained in Eq. 22.15. (Seider et al.)

$$S = QH^{0.5}$$

$$self.costing.pump_base_cost_per = \exp(9.7171 - 0.6019 * \log S + 0.0519 * (\log S)^2)$$

$$self.costing.pump_purchase_cost = F_T * material_factor * self.costing.pump_base_cost * (CE_{index}/500)$$

$$self.costing.base_cost = self.costing.pump_base_cost_per * self.costing.number_of_units$$

$$Q = self.Q/self.costing.number_of_units$$

Note: the same number of units have been considered for pumps and the pump motor

where:

- S is the pump size factor (`self.costing.size_factor`)
- Q is the volumetric flowrate in gpm (depending on the model this variable can be found as `self.unit.properties_in.flow_vol`)

- H is the head of the pump in ft (*self.pump_head*; which is defined as $H = \Delta P / \rho_{liq}$)
- FT is a parameter fixed based on the *pump_type_factor* argument (users must wisely select this factor based on the pump size factor, pump head range, and maximum motor hp)
- *material_factor* is the material factor for the pump

Table 6. Pump Type factor (Table 22.20 in Seider et al.).

Case	FT factor	# stages	Shaft rpm	Case-split	Pump Head range (ft)	Maximum Motor Hp
'1.1'	1.00	1	3600	VSC	50 - 900	75
'1.2'	1.50	1	1800	VSC	50 - 3500	200
'1.3'	1.70	1	3600	HSC	100 - 1500	150
'1.4'	2.00	1	1800	HSC	250 - 5000	250
'2.1'	2.70	2	3600	HSC	50 - 1100	250
'2.2'	8.90	2+	3600	HSC	100 - 1500	1450

For more details on how to select the FT factor, please see Seider et al.

Table 7. Materials of construction factors for centrifugal pumps and external gear pumps.

Material Factor	FM_MAT
cast iron	1.00
ductile iron	1.15
cast steel	1.35
bronze	1.90
stainless steel	2.00
hastelloy C	2.95
monel	3.30
nickel	3.50
titanium	9.70

Electric Motor:

A centrifugal pump is usually driven by an electric motor, the *self.costing.motor_purchase_cost* is calculated based on the power consumption.

$$self.motor_purchase_cost = FT * self.costing.motor_base_cost * (CE_{index}/500)(Eq.22.20)$$

$$self.costing.motor_base_cost = self.costing.motor_base_cost_per * self.costing.number_of_units$$

$$Q = self.Q/self.costing.number_of_units$$

$$self.costing.self.costing.motor_base_cost_per = \exp(5.8259 + 0.13141 \log PC + 0.053255(\log PC)^2 + 0.028628(\log PC)^3 - 0.000119(\log PC)^4) \quad (Eq.22.15)$$

$$PC = \frac{P_T}{\eta_P \eta_M} = \frac{P_B}{\eta_M} = \frac{QH\rho}{33000\eta_P \eta_M} \quad (Eq.22.16)$$

$$\eta_P = -0.316 + 0.24015 * \log Q - 0.01199(\log Q)^2 \quad (Eq.22.17)$$

$$\eta_M = 0.80 + 0.0319 \log PB - 0.00182(\log PB)^2 \quad (Eq.22.18)$$

Efficiencies are valid for PB in the range of 1 to 1500Hp and Q in the range of 50 to 5000 gpm

where:

- motor_FT is the motor type correction factor
- PC is the power consumption in hp (*self.power_consumption_hp*; coded as a pyomo expression)
- Q is the volumetric flowrate in gpm (*self.Q_gpm*)
- H is the pump head in ft (*self.pump_head*)
- PB is the pump brake hp (*self.work*)
- nP is the fractional efficiency of the pump
- nM is the fractional efficiency of the motor
- ρ is the liquid density in lb/gal

Table 8. FT Factors in Eq.(22.20) and Ranges for electric motors.

Type Motor Enclosure	3600rpm	1800rpm
Open, drip-proof enclosure, 1 to 700Hp	1.0	0.90
Totally enclosed, fan-cooled, 1 to 250Hp	1.4	1.3
Explosion-proof enclosure, 1 to 25Hp	1.8	1.7

External Gear Pumps

External gear pumps are not as common as the centrifugal pump, and various methods can be used to correlate base cost. Eq. 22.21 in Seider et al. Here the purchase cost is computed as a function of the volumetric flowrate (Q) in gpm Eq. 22.22 in Seider et al.

$$self.costing.pump_purchase_cost = material_factor * self.costing.pump_base_cost * (CE_{index}/500)$$

$$self.costing.pump_base_cost = self.costing.pump_base_cost_per * self.costing.number_of_units$$

· math::
self.costing.self.costing.pump_base_cost_perunit

$$= \exp\{(7.6964 + 0.1986\log\{Q\} + 0.0291(\log\{Q\})^2)\}$$

$$Q = \text{self.Q} / \text{self.costing.number_of_units}$$

Reciprocating Plunger Pumps

The cost correlation method used here is based on the brake horsepower (PB).

$$\text{self.costing.pump_purchase_cost} = \text{material_factor} * \text{self.costing.pump_base_cost} * (CE_{index}/500) \text{ (Eq.22.22)}$$

$$\text{self.costing.pump_base_cost} = \text{self.costing.pump_base_cost_per} * \text{self.costing.number_of_units}$$

$$\text{self.costing.pump_base_cost_per} = \exp(7.8103 + 0.26986 \log PB + 0.06718(\log PB)^2) \text{ (Eq.22.23)}$$

$$PB = f(Q)$$

$$Q = \text{self.Q} / \text{self.costing.number_of_units}$$

Table 9. Materials of construction factors for reciprocating plunger pumps.

Material	Mat_factor
ductile iron	1.00
Ni-Al-Bronze	1.15
carbon steel	1.50
stainless steel	2.20

Mover (Compressor, Fan, Blower)

If the unit represents a “Mover”, the user can select to cost it as a compressor, fan, or blower. Therefore, the user must set the “mover_type” argument.

- mover_type= ‘compressor’ or ‘fan’ or ‘blower’ (upper/lower case sensitive)

Compressor Cost

The compressor cost is based on the mechanical work of the unit. Additional arguments are required to estimate the cost such as compressor type, driver mover type, and material factor (Mat_factor).

- compressor_type = ‘centrifugal’, ‘reciprocating’, ‘screw’
- driver_mover_type = ‘electrical_motor’, ‘steam_turbine’, ‘gas_turbine’

- `Mat_factor` = 'carbon_steel',
'stain_steel', 'nickel_alloy'

$$self.costing.purchase_cost = (CE_{index}/500) * F_D * material_factor * self.costing.base_cost$$

$$self.costing.base_cost = self.costing.base_cost_per_unit * self.costing.number_of_units$$

$$self.costing.base_cost_per_unit = \exp(\alpha_1 + \alpha_2 * \log mechanical_{work})$$

$$mechanical_{work} = self.mechanical_{work} / self.costing.number_of_units$$

where:

- `FD` is the driver mover type factor and
`FM` is the construction material factor.

Table 10. Compressor type factors.

Compressor type	α_1	α_2
Centrifugal	7.5800	0.80
Reciprocating	7.9661	0.80
Screw Compressor	8.1238	0.7243

Table 11. Driver mover type (for compressors only).

Mover type	FD (mover_type)
Electric Mover	1.00
Steam Turbine	1.15
Gas Turbine	1.25

Table 12. Material of construction factor (for compressors only).

Material	Mat_factor
Cast iron	1.00
Stainless steel	1.15
Nickel alloy	1.25

Fan Cost

The fan cost is a function of the actual cubic feet per minute (Q) entering the fan. Additional arguments are required to estimate the fan cost such as `mover_type='fan'`, `fan_head_factor`, `fan_type`, and material factor (`Mat_factor`).

- `fan_type` = 'centrifugal_backward',
'centrifugal_straight', 'vane_axial',
'tube_axial'
- `fan_head_factor` = see table 14

- `Mat_factor` = 'carbon_steel', 'fiber-glass', 'stain_steel', 'nickel_alloy'

To select the correct fan type users must calculate the total head in inH2O and select the proper fan type from table 13. Additionally, the user must select the head factor (`head_factor`) from table 14.

Table 13. Typical Operating Ranges of Fans

Fan type	Flow rate (ACFM)	Total head inH2O
ACFM ^a inH2O		
Centrifugal backward curved	1000-100000	1-40
Centrifugal straight radial	1000-20000	1-30
Vane axial	1000-800000	0.02-16
Tube axial	2000-800000	0.00-10

Finally, the purchase cost of the fan is given by base cost, material factor, and fan head factor. While, the base cost is given as a function of the ACFM (Q).

$$self.costing.purchase_cost = (CE_{index}/500) * head_factor * material_factor * self.costing.base_cost$$

$$self.costing.base_cost = self.costing.base_cost_per_unit * self.costing.number_of_units$$

$$self.costing.base_cost_per_unit = \exp(\alpha_1 - \alpha_2 * \log Q + \alpha_3 * (\log Q)^2)$$

$$Q = self.Q/self.costing.number_of_units$$

Table 14. Head Factor, FH, for fans

Head (in H2O)	Centrifugal backward curved	Centrifugal straight radial	Vane axial	Tube Axial
5-8	1.15	1.15	1.15	1.15
9-15	1.30	1.30	1.30	
16-30	1.45	1.45		
31-40	1.55			

Table 15. Materials of construction factor

Material Factor	FM
carbon_steel	1
fiberglass	1.8
stain_steel	2.5
nickel_alloy	5.0

Blower Cost

The blower cost is based on the brake horsepower, which can be calculated with the inlet volumetric flow rate and pressure (cfm and lbf/in², respectively). Additional arguments are required to estimate the blower cost such as `mover_type='blower'`, `blower_type`, and material of construction factor (`Mat_factor`).

- `blower_type = 'centrifugal', 'rotary'`
- `Mat_factor = 'carbon_steel', 'aluminum', 'fiberglass', 'stain_steel', 'nickel_alloy'`

where the material factors given in table 15 for the fans can be used. In addition, centrifugal blowers are available with cast aluminum blades with `Mat_factor = 0.60`.

The purchase cost is given by the material factor and base cost. While, the base cost is given by the power consumption in horsepower (P_c).

$$self.costing.purchase_cost = material_factor * self.costing.base_cost$$

$$self.costing.base_cost = self.costing.base_cost_per_unit * self.costing.number_of_units$$

Centrifugal turbo blower (valid from $PC = 5$ to 1000 Hp):

$$self.costing.base_cost_per_unit = \exp(6.8929 + 0.7900 * \log P_c)$$

Rotary straight-lobe blower (valid from $PC = 1$ to 1000 Hp):

$$self.costing.base_cost_per_unit = \exp(7.59176 + 0.79320 * \log P_c - 0.012900 * (\log P_c)^2)$$

$$P_c = f(Q)$$

$$Q = self.Q / self.costing.number_of_units$$

Fired Heater

Indirect fired heaters, also called fired heaters, process heaters, and furnaces, are used to heat or vaporize process streams at elevated temperatures (beyond where steam is usually employed). This method computes the purchase cost of the fired heater based on the

heat duty, fuel used (`fired_type`), pressure design, and materials of construction (`Mat_factor`).

- `fuel_type` = 'fuel', 'reformer', 'pyrolysis', 'hot_water', 'salts', 'dowtherm_a', 'steam_boiler'
- `Mat_factor` = see table 16

Table 16. Materials of construction factor

Material Factor	(FM)
carbon_steel	1
Cr-Mo_alloy	1.4
stain_steel	1.7

The pressure design factor is given by (where P is pressure in psig and it is valid between 500 to 3000 psig):

$$self.pressure_factor == 0.986 - 0.0035 * (P/500.00) + 0.0175 * (P/500.00)^2$$

The base cost changes depending on the fuel type: fuel:

$$self.costing.base_cost_per_unit = \exp(0.32325 + 0.766 * \log heat_duty)$$

reformer:

$$self.costing.base_cost_per_unit = 0.859 * heat_duty^{0.81}$$

pyrolysis:

$$self.costing.base_cost_per_unit = 0.650 * heat_duty^{0.81}$$

hot_water:

$$self.costing.base_cost_per_unit = \exp(9.593 - 0.3769 * \log heat_duty + 0.03434 * (\log heat_duty)^2)$$

salts:

$$self.costing.base_cost_per_unit = 12.32 * heat_duty^{0.64}$$

dowtherm_a:

$$self.costing.base_cost_per_unit = 12.74 * heat_duty^{0.65}$$

steam_boiler:

$$self.costing.base_cost_per_unit = 0.367 * heat_duty^{0.77}$$

$$self.costing.base_cost = self.costing.base_cost_per_unit * self.costing.number_of_units$$

Finally, the purchase cost is given by:

$$self.purchase_cost = (CE_{index}/500) * pressure_design * material_factor * base_cost$$

Cost of Pressure Vessels and Towers for Distillation

Pressure vessels cost is based on the weight of the vessel, the cost of platforms and ladders can be included, and the cost of internal packing or trays can be calculated as well. This method constructs by default the cost of pressure vessels with platforms and ladders, and trays cost can be calculated if `trays=True`. This method requires a few arguments to build the cost of vessel. We recommend using this method to cost reactors (CSTR or PFR), flash tanks, vessels, and distillation columns.

- `alignment` = 'horizontal', 'vertical'
- `Mat_factor` = 'carbon_steel'
- `weight_limit` = 'option1', 'option2' (option 1: 1000 to 920,000 lb, option 2: 9000 to 2.5M lb only for vertical vessels)
- `L_D_range` = 'option1', 'option2' (option 1: $3 < D < 21$, $12 < L < 40$; option 2: $3 < D < 24$, $27 < L < 170$; all in ft D: diameter, L: length) only for vertical vessels
- `PL='True'`, 'False': to build platforms and ladders cost
- `plates` = 'True', 'False': to build tray cost for distillation columns
- `tray_mat_factor` = 'carbon_steel' see table 18
- `tray_type` = 'sieve'
- `number_tray` = 10
- `ref_parameter_diameter`=None
- `ref_parameter_length`=None

By adding reference parameter, the method can be constructed in any pyomo costing block. Since the generic models do not include the variables required to cost these type of units, the user must create the blocks and variables. For example: `m.fs.unit = Block()`, `m.fs.unit.diameter = Var()`, `m.fs.unit.length = Var()`. Then `m.fs.unit.costing = pyo.Block()` and call `vessel_costing` method = `vessel_costing(m.fs.unit.costing, args)`.

Table 17. Materials of construction factor and material density

Material Factor	(FM)	methal density (lb/in^3)
carbon_steel	1	0.284
low_alloy_steel	1.2	0.271
stain_steel_304	1.7	0.270
stain_steel_316	2.1	0.276
carpenter_20CB-3	3.2	0.292
nickel_200	5.4	0.3216
monel_400	3.6	0.319
inconel_600	3.9	0.3071
incoloy_825	3.7	0.2903
titanium	7.7	0.1628

Vessel Cost

The weight of the unit is calculated based on the methal density, length, Diameter, and shell thickness. *shel_thickness* is a parameter initialized to 1.25, however, the user must calculate the shell wall minimum thickness computed from the ASME pressure vessel code (tp) add the average vessel thickness, the necessary wall thickness (tE), and select the appropriate *shel_thickness*.

$$self.weight == \pi * ((D * 12) + self.shel_thickness) * ((L * 12) + (0.8 * D * 12)) * self.shel_thickness * self.material_den$$

The base cost of the vessel is given by:

Horizontal vessels (option1: 1000 < W < 920,000 lb):

$$self.costing.base_cost_per_unit = \exp(8.9552 - 0.2330 * \log weight + 0.04333 * (\log weight)^2)$$

Vertical vessels (option1: 4200 < W < 1M lb):

$$self.costing.base_cost_per_unit = \exp(8.9552 - 0.2330 * \log weight + 0.04333 * (\log weight)^2)$$

Vertical vessels (option2: 9,000 < W < 2.5M lb):

$$self.costing.base_cost_per_unit = \exp(7.2756 - 0.18255 * \log weight + 0.02297 * (\log weight)^2)$$

$$self.costing.base_cost = self.costing.base_cost_per_unit * self.costing.number_of_units$$

$$weight = self.weight / self.costing.number_of_units$$

The vessel purchase cost is given by:

$$self.vessel_purchase_cost = (CE_{index}/500) * material_factor * self.base_cost + (self.base_cost_platf_ladders * self.costi$$

note that if PL = 'False', the cost of platforms and ladders is not included.

The final purchase cost is given by:

$$\text{self.purchase_cost} = \text{self.vessel_purchase_cost} + (\text{self.purchase_cost_trays} * \text{self.costing.number_of_units})$$

note that if plates='False', the cost of trays is not included.

Base Cost of Platforms and ladders

The cost of platforms and ladders is based on the diameter and length in ft. Horizontal vessels (option1: $3 < D < 12$ ft):

$$\text{self.base_cost_platf_ladders} = 20059 * D^{0.20294}$$

Vertical vessels (option1: $3 < D < 12$ ft and $12 < L < 40$ ft):

$$\text{self.base_cost_platf_ladders} = 361.8 * D^{0.73960} * L^{0.70684}$$

Vertical vessels (option2: $3 < D < 24$ ft and $27 < L < 170$ ft):

$$\text{self.base_cost_platf_ladders} = 300.9 * D^{0.63316} * L^{0.80161}$$

Purchase Cost of Plates

The cost of plates is based on the number of trays, the type of trays used, and materials of construction. Tray type factor (tray_factor) is 1.0 for sieve trays, 1.18 for valve trays (valve), and 1.87 for bubble cap trays (bubble_cap). The number of trays factor (number_tray_factor) is equal to 1 if the number of trays is greater than 20. However, if the number of trays is less than 20, the number_tray_factor is given by:

$$\text{self.number_tray_factor} = \frac{2.25}{1.0414^{NT}}$$

The materials of construction factor is calculated using the following equation:

$$\alpha_1 + \alpha_2 * D$$

where alphas for different materials of construction are given in table 18.

Table 18. Materials of construction factor

Material	alpha1	alpha2
carbon_steel	1	0
stain_steel_303	1.189	0.0577
stain_steel_316	1.401	0.0724
carpenter_20CB-3	1.525	0.0788
monel_400	2.306	0.1120

The tray base cost is then calculated as:

$$self.base_cost_trays = 468.00 * \exp(0.1739 * D)$$

The purchase cost of the trays is given by:

$$self.purchase_cost_trays = (CE_{index}/500) * self.number_trays * self.number_tray_factor * self.type_tray_factor * self.$$

Variable-Like Expressions

There are a number of cases within IDAES where a modeler may wish to use an *Expression* in place of a *Var* to reduce the complexity of their model. A common example of this is in the ideal *Separator* unit where the outlet *Ports* use *Expressions* for the state variable in order to reduce the number of variables (and thus constraints) in the model.

In these cases, it is possible that a user might mistake the *Expression* for a *Var* and attempt to use methods such as *fix()* on it. In order to provide the user with a useful error message informing them that this will not work, IDAES has created a derived *VarLikeExpression* component for these situations. This component derives directly from Pyomo's *Expression* component and implements common methods associated with *Vars* which will return an error message informing the user that the component is an *Expression*, and a suggestion on how to proceed.

```
class idaes.core.util.misc.VarLikeExpression(*args, **kwds)
```

A shared var-like expression container, which may be defined over an index.

Constructor Arguments: initialize: A Pyomo expression or dictionary of expressions used to initialize this object.

expr: A synonym for initialize.

rule: A rule function used to initialize this object.

```
class idaes.core.util.misc.SimpleVarLikeExpression (*args, **kws)
```

```
add (index, expr)
```

Add an expression with a given index.

```
class idaes.core.util.misc.IndexedVarLikeExpression (*args, **kws)
```

```
add (index, expr)
```

Add an expression with a given index.

```
class idaes.core.util.misc._GeneralVarLikeExpressionData (expr=None,      component=None)
```

An object derived from `_GeneralExpressionData` which implements methods for common APIs on Vars.

Constructor Arguments: expr: The Pyomo expression stored in this expression.

component: The Expression object that owns this data.

Public Class Attributes: expr: The expression owned by this data.

Private class attributes: `_component`: The expression component.

property value

The `.value` property getter on `_GeneralExpressionDataImpl` is deprecated. Use the `.expr` property getter instead (deprecated in 4.3.11323)

Type DEPRECATION WARNING

4.5.2 Model Libraries

Generic IDAES Model Library

This library contains a suite of generic models that are applicable across most process applications. This library also forms the foundation for many of the specialized application libraries which build off these models.

Property Models

Cubic Equations of State

This property package implements a general form of a cubic equation of state which can be used for most cubic-type equations of state. This package supports phase equilibrium calculations with a smooth phase transition formulation that makes it amenable for equation oriented optimization. The following equations of state are currently supported:

- Peng-Robinson
- Soave-Redlich-Kwong

Flow basis: Molar

Units: SI units

State Variables:

The state block uses the following state variables:

Inputs

When instantiating the parameter block that uses this particular state block, 1 optional argument can be passed:

The `valid_phase` argument denotes the valid phases for a given set of inlet conditions. For example, if the user knows a priori that it will only be a single phase (for example liquid only), then it is best not to include the complex flash equilibrium constraints in the model. If the user does not specify any option, then the package defaults to a 2 phase assumption meaning that the constraints to compute the phase equilibrium will be computed.

Degrees of Freedom

In general, the general cubic equation of state has a number of degrees of freedom equal to 2 + the number of components in the system (total flow rate, temperature, pressure and N-1 mole fractions). In some cases (primarily inlets to units), this is increased by 1 due to the removal of a constraint on the sum of mole fractions.

General Cubic Equation of State

All equations come from “The Properties of Gases and Liquids, 4th Edition” by Reid, Prausnitz and Poling. The general cubic equation of state is represented by the following equations:

$$0 = Z^3 - (1 + B - uB)Z^2 + (A - uB - (u - w)B^2)Z - AB - wB^2 - wB^3$$

$$A = \frac{a_m P}{R^2 T^2}$$

$$B = \frac{b_m P}{RT}$$

where Z is the compressibility factor of the mixture, a_m and b_m are properties of the mixture and u and w are parameters which depend on the specific equation of state being used as shown in the table below.

Equation	u	w	Ω_A	Ω_B	κ_j
Peng-Robinson	2	-1	0.45724	0.07780	$(1 + (1 - T_r^2)(0.37464 + 1.54226\omega_j - 0.26992\omega_j^2))^2$
Soave-Redlich-Kwong	1	0	0.42748	0.08664	$(1 + (1 - T_r^2)(0.48 + 1.574\omega_j - 0.176\omega_j^2))^2$

The properties a_m and b_m are calculated from component specific properties a_j and b_j as shown below:

$$a_j = \frac{\Omega_A R^2 T_{c,j}^2}{P_{c,j}} \kappa_j$$

$$b_j = \frac{\Omega_B R T_{c,j}}{P_{c,j}}$$

$$a_m = \sum_i \sum_j y_i y_j (a_i a_j)^{1/2} (1 - k_{ij})$$

$$b_m = \sum_i y_i b_i$$

where $P_{c,j}$ and $T_{c,j}$ are the component critical pressures and temperatures, y_j is the mole fraction of component j , k_{ij} are a set of binary interaction parameters which are specific to the equation of state and Ω_A , Ω_B and κ_j are taken from the table above. ω_j is the Pitzer acentric factor of each component.

The cubic equation of state is solved for each phase via a call to an external function which automatically identifies the correct root of the cubic and returns the value of Z as a function of A and B along with the first and second partial derivatives.

VLE Model with Smooth Phase Transition

The flash equations consists of the following equations:

$$F^{in} = F^{liq} + F^{vap}$$

$$z_i^{in} F^{in} = x_i^{liq} F^{liq} + y_i^{vap} F^{vap}$$

At the equilibrium condition, the fugacity of the vapor and liquid phase are defined as follows:

$$\ln f_i^{vap} = \ln f_i^{liq}$$

$$f_i^{phase} = y_i^{phase} \phi_i^{phase} P$$

$$\ln \phi_i = \frac{b_i}{b_m} (Z - 1) - \ln (Z - B) + \frac{A}{B\sqrt{u^2 - 4w}} \left(\frac{b_i}{b_m} - \delta_i \right) \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right)$$

$$\delta_i = \frac{2a_i^{1/2}}{a_m} \sum_j x_j a_j^{1/2} (1 - k_{ij})$$

The cubic equation of state is solved to find Z for each phase subject to the composition of that phase. Typically, the flash calculations are computed at a given temperature, T . However, the flash calculations become trivial if the given conditions do not fall in the two phase region. For simulation only studies, the user may know a priori the condition of the stream but when the same set of equations are used for optimization, there is a high probability that the specifications can transcend the phase envelope and hence the flash equations included may be trivial in the single phase region (i.e. liquid or vapor only). To circumvent this problem, property packages in IDAES that support VLE will compute the flash calculations at an “equilibrium” temperature T_{eq} . The equilibrium temperature is computed as follows:

$$T_1 = \max(T_{bubble}, T)$$

$$T_{eq} = \min(T_1, T_{dew})$$

where T_{eq} is the equilibrium temperature at which flash calculations are computed, T is the stream temperature, T_1 is the intermediate temperature variable, T_{bubble} is the bubble point temperature of mixture, and T_{dew} is the dew point temperature of the mixture. Note that, in the above equations, approximations are used for the max and min functions as follows:

$$T_1 = 0.5[T + T_{bubble} + \sqrt{(T - T_{bubble})^2 + \epsilon_1^2}]$$

$$T_{eq} = 0.5[T_1 + T_{dew} - \sqrt{(T - T_{dew})^2 + \epsilon_2^2}]$$

where ϵ_1 and ϵ_2 are smoothing parameters (mutable). The default values are 0.01 and 0.0005 respectively. It is recommended that $\epsilon_1 > \epsilon_2$. Please refer to reference 4 for more details. Therefore, it can be seen that if the stream temperature is less than that of the bubble point temperature, the VLE calculations will be computed at the bubble point. Similarly, if the stream temperature is greater than the dew point temperature, then the VLE calculations are computed at the dew point temperature. For all other conditions, the equilibrium calculations will be computed at the actual temperature.

Other Constraints

Additional constraints are included in the model to compute the thermodynamic properties based on the cubic equation of state, such as enthalpies and entropies. Please note that, these constraints are added only if the variable is called for when building the model. This eliminates adding unnecessary constraints to compute properties that are not needed in the model.

All thermophysical properties are calculated using an ideal and residual term, such that:

$$p = p^0 + p^r$$

The residual term is derived from the partial derivatives of the cubic equation of state, whilst the ideal term is determined using empirical correlations.

Enthalpy

The ideal enthalpy term is given by:

$$h_i^0 = \int_{298.15}^T (A + BT + CT^2 + DT^3) dT + \Delta h_{form}^{298.15K}$$

The residual enthalpy term is given by:

$$h_i^r b_m \sqrt{u^2 - 4w} = \left(T \frac{da}{dT} - a_m \right) \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right) + RT(Z - 1) b_m \sqrt{u^2 - 4w}$$

$$\frac{da}{dT} \sqrt{T} = -\frac{R}{2} \sqrt{\Omega_A} \sum_i \sum_j y_i y_j (1 - k_{ij}) \left(f_{w,j} \sqrt{a_i \frac{T_{c,j}}{P_{c,j}}} + f_{w,i} \sqrt{a_j \frac{T_{c,i}}{P_{c,i}}} \right)$$

Entropy

The ideal entropy term is given by:

$$s_i^0 = \int_{298.15}^T \frac{(A + BT + CT^2 + DT^3)}{T} dT + \Delta s_{form}^{298.15K}$$

The residual entropy term is given by:

$$s_i^r b_m \sqrt{u^2 - 4w} = R \ln \frac{Z - B}{Z} b_m \sqrt{u^2 - 4w} + R \ln \frac{Z^{Pref}}{P} b_m \sqrt{u^2 - 4w} + \frac{da}{dT} \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right)$$

Fugacity

Fugacity is calculated from the system pressure, mole fractions and fugacity coefficients as follows:

$$f_{i,p} = x_{i,p} \phi_{i,p} P$$

Fugacity Coefficient

The fugacity coefficient is calculated from the departure function of the cubic equation of state as shown below:

$$\ln \phi_i = \frac{b_i}{b_m} (Z - 1) - \ln (Z - B) + \frac{A}{B \sqrt{u^2 - 4w}} \left(\frac{b_i}{b_m} - \delta_i \right) \ln \left(\frac{2Z + B(u + \sqrt{u^2 - 4w})}{2Z + B(u - \sqrt{u^2 - 4w})} \right)$$
$$\delta_i = \frac{2a_i^{1/2}}{a_m} \sum_j x_j a_j^{1/2} (1 - k_{ij})$$

Gibbs Energy

The Gibbs energy of the system is calculated using the definition of Gibbs energy:

$$g_i = h_i - T \Delta s_i$$

List of Variables

Variable Name	Description	Units
flow_mol	Total molar flow rate	mol/s
mole_frac_comp	Mixture mole fraction indexed by component	None
temperature	Temperature	K
pressure	Pressure	Pa
flow_mol_phase	Molar flow rate indexed by phase	mol/s
mole_frac_phase_comp	Mole fraction indexed by phase and component	None
pressure_sat	Saturation or vapor pressure indexed by component	Pa
dens_mol_phase	Molar density indexed by phase	mol/m ³
dens_mass_phase	Mass density indexed by phase	kg/m ³
enth_mol_phase	Molar enthalpy indexed by phase	J/mol
enth_mol	Molar enthalpy of mixture	J/mol
entr_mol_phase	Molar entropy indexed by phase	J/mol.K
entr_mol	Molar entropy of mixture	J/mol.K
fug_phase_comp	Fugacity indexed by phase and component	Pa
fug_coeff_phase_comp	Fugacity coefficient indexed by phase and component	None
gibbs_mol_phase	Molar Gibbs energy indexed by phase	J/mol
mw	Molecular weight of mixture	kg/mol
mw_phase	Molecular weight by phase	kg/mol
temperature_bubble	Bubble point temperature	K
temperature_dew	Dew point temperature	K
pressure_bubble	Bubble point pressure	Pa
pressure_dew	Dew point pressure	Pa
_teq	Temperature at which the VLE is calculated	K

List of Parameters

Parameter Name	Description	Units
cubic_type	Type of cubic equation of state to use, from CubicEoS Enum	None
pressure_ref	Reference pressure	Pa
temperature_ref	Reference temperature	K
omega	Pitzer acentricity factor	None
kappa	Binary interaction parameters for EoS (note that parameters are specific for a given EoS)	None
mw_comp	Component molecular weights	kg/mol
cp_ig	Parameters for calculating component heat capacities	varies
dh_form	Component standard heats of formation (used for enthalpy at reference state)	J/mol
ds_form	Component standard entropies of formation (used for entropy at reference state)	J/mol.K
antoine	Component Antoine coefficients (used to initialize bubble and dew point calculations)	bar, K

Config Block Documentation

```
class idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicParameterData (component)
    General Property Parameter Block
    Class

build()
    Callable method for Block construction.

classmethod define_metadata (obj)
    Define properties supported and units.

class idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicStateBlock (*args,
                                                                                   **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

parameters A reference to an instance of the Property Parameter Block associated with this property package.

defined_state Flag indicating whether the state should be considered fully defined, and thus whether constraints such as sum of mass/mole fractions should be included, **default** - False. **Valid values:** { **True** - state variables will be fully defined, **False** - state variables will not be fully defined. }

has_phase_equilibrium Flag indicating whether phase equilibrium constraints should be constructed in this state block, **default** - True. **Valid values:** { **True** - StateBlock should calculate phase equilibrium, **False** - StateBlock should not calculate phase equilibrium. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.

- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (CubicStateBlock) New instance

class `idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicStateBlockData` (**args*, ***kwargs*)

An general property package for cubic equations of state with VLE.

build ()

Callable method for Block construction.

define_display_vars ()

Method used to specify components to use to generate stream tables and other outputs. Defaults to define_state_vars, and developers should overload as required.

define_state_vars ()

Define state vars.

get_energy_density_terms (*p*)

Create energy density terms.

get_enthalpy_flow_terms (*p*)

Create enthalpy flow terms.

get_material_density_terms (*p, j*)

Create material density terms.

get_material_flow_basis ()

Method which returns an Enum indicating the basis of the material flow term.

get_material_flow_terms (*p, j*)

Create material flow terms for control volume.

model_check ()

Model checks for property block.

Vapor-Liquid Equilibrium Property Models (Ideal Gas - Non-ideal Liquids)

This property package supports phase equilibrium calculations with a smooth phase transition formulation that makes it amenable for equation oriented optimization. The gas phase is assumed to be ideal and for the liquid phase, the package supports an ideal liquid or a non-ideal liquid using an activity coefficient model. To compute the activity coefficient, the package currently supports the Non Random Two Liquid Model (NRTL) or the Wilson model. Therefore, this property package supports the following combinations for gas-liquid mixtures for VLE calculations:

1. Ideal (vapor) - Ideal (liquid)
2. Ideal (vapor) - NRTL (liquid)
3. Ideal (vapor) - Wilson (liquid)

Flow basis: Molar

Units: SI units

State Variables:

The state block supports the following two sets of state variables:

Option 1 - “FTPz”:

Option 2 - “FcTP”:

The user can specify the choice of state variables while instantiating the parameter block. See the Inputs section for more details.

Support for other combinations of state variables will be made available in the future.

Inputs

When instantiating the parameter block that uses this particular state block, 2 arguments can be passed:

The `valid_phase` argument denotes the valid phases for a given set of inlet conditions. For example, if the user knows a priori that it will only be a single phase (for example liquid only),

then it is best not to include the complex flash equilibrium constraints in the model. If the user does not specify any option, then the package defaults to a 2 phase assumption meaning that the constraints to compute the phase equilibrium will be computed.

The `activity_coeff_model` denotes the liquid phase assumption to be used. If the user does not specify any option, then the package defaults to assuming an ideal liquid assumption.

The `state_vars` denotes the preferred set of state variables to be used. If the user does not specify any option, then the package defaults to using the total flow, mixture mole fraction, temperature and pressure as the state variables.

Degrees of Freedom

The number of degrees of freedom that need to be fixed to yield a square problem (i.e. degrees of freedom = 0) depends on the options selected. The following table provides a summary of the variables to be fixed and also the corresponding variable names in the model.

Property Model Type	State variables	Additional Variables	Total number of variables
Ideal (vapor) - Ideal (liquid)	flow_mol, temperature, pressure, mole_frac_comp	None	$3 + N_c$
Ideal (vapor) - NRTL (liquid)	flow_mol, temperature, pressure, mole_frac_comp	alpha, tau	$3 + N_c + 2N_c^2$
Ideal (vapor) - Wilson (liquid)	flow_mol, temperature, pressure, mole_frac_comp	vol_mol_comp, tau	$3 + N_c + 2N_c^2$

Please refer to reference 3 for recommended values for `tau`.

VLE Model with Smooth Phase Transition

The flash equations consists of the following equations depending on the choice of state variables selected by the user.

If the state variables are total flow, mole fraction, temperature, and pressure, then the following constraints are implemented:

$$F^{in} = F^{liq} + F^{vap}$$

$$z_i^{in} F^{in} = x_i^{liq} F^{liq} + y_i^{vap} F^{vap}$$

If the state variables are component flow rates, temperature, and pressure, then the following constraints are implemented:

$$F_i^{in} = F_i^{liq} + F_i^{vap}$$

The equilibrium condition, the fugacity of the vapor and liquid phase are defined as follows:

$$f_i^{vap} = f_i^{liq}$$

$$f_i^{vap} = y_i \phi_i P$$

$$f_i^{liq} = x_i p_i^{sat} \nu_i$$

The equilibrium constraint is written as a generic constraint such that it can be extended easily for non-ideal gases and liquids. As this property package only supports an ideal gas, the fugacity coefficient (ϕ_i) for the vapor phase is 1 and hence the expression reduces to $y_i P$. For the liquid phase, if the ideal option is selected then the activity coefficient (ν_i) is 1. If an activity coefficient model is selected then corresponding constraints are added to compute the activity coefficient.

Typically, the flash calculations are computed at a given temperature, T . However, the flash calculations become trivial if the given conditions do not fall in the two phase region. For simulation only studies, the user may know a priori the condition of the stream but when the same set of equations are used for optimization, there is a high probability that the specifications can transcend the phase envelope and hence the flash equations included may be trivial in the single phase region (i.e. liquid or vapor only). To circumvent this problem, property packages in IDAES that

support VLE will compute the flash calculations at an “equilibrium” temperature T_{eq} . The equilibrium temperature is computed as follows:

$$T_1 = \max(T_{bubble}, T)$$

$$T_{eq} = \min(T_1, T_{dew})$$

where T_{eq} is the equilibrium temperature at which flash calculations are computed, T is the stream temperature, T_1 is the intermediate temperature variable, T_{bubble} is the bubble point temperature of mixture, and T_{dew} is the dew point temperature of the mixture. Note that, in the above equations, approximations are used for the max and min functions as follows:

$$T_1 = 0.5[T + T_{bubble} + \sqrt{(T - T_{bubble})^2 + \epsilon_1^2}]$$

$$T_{eq} = 0.5[T_1 + T_{dew} - \sqrt{(T_1 - T_{dew})^2 + \epsilon_2^2}]$$

where ϵ_1 and ϵ_2 are smoothing parameters(mutable). The default values are 0.01 and 0.0005 respectively. It is recommended that $\epsilon_1 > \epsilon_2$. Please refer to reference 4 for more details. Therefore, it can be seen that if the stream temperature is less than that of the bubble point temperature, the VLE calculations will be computed at the bubble point. Similarly, if the stream temperature is greater than the dew point temperature, then the VLE calculations are computed at the dew point temperature. For all other conditions, the equilibrium calculations will be computed at the actual temperature.

Additional constraints are included in the model to compute the thermodynamic properties such as component saturation pressure, enthalpy, specific heat capacity. Please note that, these constraints are added only if the variable is called for when building the model. This eliminates adding unnecessary constraints to compute properties that are not needed in the model.

The saturation or vapor pressure (`pressure_sat`) for component i is computed using the following correlation[1]:

$$\log \frac{P^{sat}}{P_c} = \frac{Ax + Bx^{1.5} + Cx^3 + Dx^6}{1 - x}$$

$$x = 1 - \frac{T_{eq}}{T_c}$$

where P_c is the critical pressure, T_c is the critical temperature of the component and T_{eq} is the equilibrium temperature at which the saturation pressure is computed. Please note that when using this expression, $T_{eq} < T_c$ is required and when violated it results in a negative number raised to the power of a fraction.

The specific enthalpy (`enthalpy_comp_liq`) for component i is computed using the following expression for the liquid phase:

$$h_i^{liq} = \Delta h_{form,Liq,i} + \int_{298.15}^T (A + BT + CT^2 + DT^3 + ET^4) dT$$

The specific enthalpy (`enthalpy_comp_vap`) for component i is computed using the following expression for the vapor phase:

$$h_i^{vap} = \Delta h_{form,Vap,i} + \int_{298.15}^T (A + BT + CT^2 + DT^3 + ET^4) dT$$

The mixture specific enthalpies (`enthalpy_liq` & `enthalpy_vap`) are computed using the following expressions for the liquid and vapor phase respectively:

$$H^{liq} = \sum_i h_i^{liq} x_i$$

$$H^{vap} = \sum_i h_i^{vap} y_i$$

Similarly, specific entropies are calculated as follows. The specific entropy (`entropy_comp_liq`) for component i is computed using the following expression for the liquid phase:

$$s_i^{liq} = \Delta s_{form,Liq,i} + \int_{298.15}^T (A/T + B + CT + DT^2 + ET^3) dT$$

The specific entropy (`entropy_comp_vap`) for component i is computed using the following expression for the vapor phase:

$$s_i^{vap} = \Delta s_{form,Vap,i} + \int_{298.15}^T (A/T + B + CT + DT^2 + ET^3) dT$$

Please refer to references 1 and 2 to get parameters for different components.

Activity Coefficient Model - NRTL

The activity coefficient for component i is computed using the following equations when using the Non-Random Two Liquid model [3]:

$$\log \gamma_i = \frac{\sum_j x_j \tau_{ji} G_{ji}}{\sum_k x_k G_{ki}} + \sum_j \frac{x_j G_{ij}}{\sum_k x_k G_{kj}} \left[\tau_{ij} - \frac{\sum_m x_m \tau_{mj} G_{mj}}{\sum_k x_k G_{kj}} \right]$$

$$G_{ij} = \exp(-\alpha_{ij} \tau_{ij})$$

where α_{ij} is the non-randomness parameter and τ_{ij} is the binary interaction parameter for the NRTL model.

Note that in the IDAES implementation, these are declared as variables that allows for more flexibility and the ability to use these in a parameter estimation problem. These NRTL model specific variables need to be either fixed for a given component set or need to be estimated from VLE data.

The bubble point is computed by enforcing the following condition:

$$\sum_i [z_i p_i^{sat}(T_{bubble}) \nu_i] - P = 0$$

Activity Coefficient Model - Wilson

The activity coefficient for component i is computed using the following equations when using the Wilson model [3]:

$$\log \gamma_i = 1 - \log \sum_j x_j G_{ji} - \sum_j \frac{x_j G_{ij}}{\sum_k x_k G_{kj}}$$

$$G_{ij} = (v_i/v_j) \exp(-\tau_{ij})$$

where v_i is the molar volume of component i and τ_{ij} is the binary interaction parameter. These are Wilson model specific variables that either need to be fixed for a given component set or need to be estimated from VLE data.

The bubble point is computed by enforcing the following condition:

$$\sum_i [z_i p_i^{sat}(T_{bubble}) \nu_i] - P = 0$$

List of Variables

Variable Name	Description	Units
flow_mol	Total molar flow rate	mol/s
mole_frac_comp	Mixture mole fraction indexed by component	None
temperature	Temperature	K
pressure	Pressure	Pa
flow_mol_phase	Molar flow rate indexed by phase	mol/s
mole_frac_phase_comp	Mole fraction indexed by phase and component	None
pressure_sat	Saturation or vapor pressure indexed by component	Pa
density_mol_phase	Molar density indexed by phase	mol/m3
ds_vap	Molar entropy of vaporization	J/mol.K
enthalpy_comp_liq	Liquid molar enthalpy indexed by component	J/mol
enthalpy_comp_vap	Vapor molar enthalpy indexed by component	J/mol
enthalpy_liq	Liquid phase enthalpy	J/mol
enthalpy_vap	Vapor phase enthalpy	J/mol
entropy_comp_liq	Liquid molar entropy indexed by component	J/mol
entropy_comp_vap	Vapor molar entropy indexed by component	J/mol
entolpy_liq	Liquid phase entropy	J/mol
entropy_vap	Vapor phase entropy	J/mol
temperature_bubble	Bubble point temperature	K
temperature_dew	Dew point temperature	K
_temperature_equilibrium	Temperature at which the VLE is calculated	K

Table 5: NRTL model specific variables

Variable Name	Description	Units
alpha	Non-randomness parameter indexed by component and component	None
tau	Binary interaction parameter indexed by component and component	None
activity_coeff_comp	Activity coefficient indexed by component	None

Table 6: Wilson model specific variables

Variable Name	Description	Units
vol_mol_comp	Molar volume of component indexed by component	None
tau	Binary interaction parameter indexed by component and component	None
activity_coeff_comp	Activity coefficient indexed by component	None

Initialization

Config Block Documentation

```
class idaes.generic_models.properties.activity_coeff_models.activity_coeff_prop_pack.ActivityCo
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.

- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

default_arguments Default arguments to use with Property Package

activity_coeff_model Flag indicating the activity coefficient model to be used for the non-ideal liquid, and thus corresponding constraints should be included, **default** - Ideal liquid. **Valid values:** { “NRTL” - Non Random Two Liquid Model, “Wilson” - Wilson Liquid Model, }

state_vars Flag indicating the choice for state variables to be used for the state block, and thus corresponding constraints should be included, **default** - FTPz **Valid values:** { “FTPx” - Total flow, Temperature, Pressure and Mole fraction, “FcTP” - Component flow, Temperature and Pressure }

valid_phase Flag indicating the valid phase for a given set of conditions, and thus corresponding constraints should be included, **default** - (“Vap”, “Liq”). **Valid values:** { “Liq” - Liquid only, “Vap” - Vapor only, (“Vap”, “Liq”) - Vapor-liquid equilibrium, (“Liq”, “Vap”) - Vapor-liquid equilibrium, }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (ActivityCoeffParameterBlock)
New instance

class idaes.generic_models.properties.activity_coeff_models.activity_coeff_prop_pack.**ActivityCo**

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

parameters A reference to an instance of the Property Parameter Block associated with this property package.

defined_state Flag indicating whether the state should be considered fully defined, and thus whether constraints such as sum of mass/mole fractions should be included, **default** - False. **Valid values:** { **True** - state variables will be fully defined, **False** - state variables will not be fully defined. }

has_phase_equilibrium Flag indicating whether phase equilibrium constraints should be constructed in this state block, **default** - True. **Valid values:** { **True** - StateBlock should calculate phase equilibrium, **False** - StateBlock should not calculate phase equilibrium. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (ActivityCoeffStateBlock) New instance

class `idaes.generic_models.properties.activity_coeff_models.activity_coeff_prop_pack.ActivityCo`

An example property package for ideal VLE.

build()

Callable method for Block construction.

define_state_vars ()
Define state vars.

get_energy_density_terms (p)
Create enthalpy density terms.

get_enthalpy_flow_terms (p)
Create enthalpy flow terms.

get_material_density_terms (p, j)
Create material density terms.

get_material_flow_basis ()
Declare material flow basis.

get_material_flow_terms (p, j)
Create material flow terms for control volume.

model_check ()
Model checks for property block.

References

1. “The properties of gases and liquids by Robert C. Reid”
2. “Perry’s Chemical Engineers Handbook by Robert H. Perry”.
3. H. Renon and J.M. Prausnitz, “Local compositions in thermodynamic excess functions for liquid mixtures.”, AIChE Journal Vol. 14, No.1, 1968.
4. AP Burgard, JP Eason, JC Eslick, JH Ghose, A Lee, LT Biegler, DC Miller. “A Smooth, Square Flash Formulation for Equation Oriented Flowsheet Optimization”, Computer Aided Chemical Engineering 44, 871-876, 2018

Pure Component Helmholtz EoS

The Helmholtz Equation of State (EoS) classes serve as a common core for pure component property packages where very accurate and thermodynamically consistent pure component properties are required. This contains general information. Thermodynamic properties for all Helmholtz EoS packages are calculated by the core class only the parameters differ between specific component implementation. Specific implementations may also contain additional

properties such as viscosity and thermal conductivity. For specific property packages details see the pages below.

International Association of the Properties of Water and Steam IAPWS-95

Accurate and thermodynamically consistent steam properties are provided for the IDAES framework by implementing the International Association for the Properties of Water and Steam's *"Revised Release on the IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use."* Non-analytic terms designed to improve accuracy very near the critical point were omitted, because they cause a singularity at the critical point, a feature which is undesirable in optimization problems. The IDAES implementation provides features which make the water and steam property calculations amenable to rigorous mathematical optimization.

Please see the [general Helmholtz documentation](#) for more information.

Example

The Heater unit model [example](#), provides a simple example for using water properties.

```
from idaes.generic_models.  
↳ properties import iapws95  
import pyomo.environ  
↳ as pe # Pyomo environment  
from idaes.  
↳ core import FlowsheetBlock,  
↳ MaterialBalanceType  
from idaes.generic_models.  
↳ unit_models import Heater
```

```
# Create an  
↳ empty flowsheet and steam  
↳ property parameter block.  
model = pe.ConcreteModel()  
model.fs  
↳ = FlowsheetBlock(default=  
↳ {"dynamic": False})  
model.  
↳ fs.  
↳ properties  
↳ =  
↳ iapws95.  
↳ iapws95ParameterBlock(default=  
↳ {
```

(continues on next page)

```
458  
↳ iapws95.  
↳ iapws95ParameterBlock(default=  
↳ {
```

(continued from previous page)

```

    "phase_presentation
↳ ":iapws95.PhaseType.LG,
    "state_vars
↳ ":iapws95.StateVars.PH))

# Add a Heater_
↳ model to the flowsheet.
model.fs.
↳ heater = Heater(default={
    "property_package
↳ ": model.fs.properties,
    "material_balance_
↳ type": MaterialBalanceType.
↳ componentTotal}))

# Setup_
↳ the heater model by fixing_
↳ the inputs and heat duty
model.fs.heater.
↳ inlet[:].enth_mol.fix(4000)
model.fs.heater.
↳ inlet[:].flow_mol.fix(100)
model.fs.heater.inlet[:].
↳ pressure.fix(101325)
model.fs.heater.
↳ heat_duty[:].fix(100*20000)

# Initialize the model.
model.fs.heater.initialize()

```

Since all properties except the state variables are Pyomo Expressions in the water properties module, after solving the problem any property can be calculated in any state block. Continuing from the heater example, to get the viscosity of both phases, the lines below could be added.

```

mu_l_
↳ = pe.value(model.fs.heater.
↳ control_volume.properties_
↳ out[0].visc_d_phase["Liq"])
mu_v_
↳ = pe.value(model.fs.heater.
↳ control_volume.properties_
↳ out[0].visc_d_phase["Vap"])

```

For more information about how State-Blocks and PropertyParameterBlocks work see the [StateBlock documentation](#).

Expressions

The IAPWS-95 property package contains the standard expressions described in the *general Helmholtz documentation*, but it also defines expressions for transport properties.

Expression	Description
<code>therm_cond_phase[phase]</code>	Thermal conductivity of phase (W/K/m)
<code>visc_d_phase[phase]</code>	Viscosity of phase (Pa/s)
<code>visc_k_phase[phase]</code>	Kinimatic viscosity of phase (m ² /s)

Convenience Functions

`idaes.generic_models.properties.iapws95.htpx` ($T=None$, $P=None$, $x=None$)

Convenience function to calculate steam enthalpy from temperature and either pressure or vapor fraction. This function can be used for inlet streams and initialization where temperature is known instead of enthalpy. User must provided values for two of T, P, or x.

Parameters

- **T** – Temperature with units (between 200 and 3000 K)
- **P** – Pressure with units (between 1 and 1e9 Pa), None if saturated vapor
- **x** – Vapor fraction [mol vapor/mol total] (between 0 and 1), None if
- **or subcooled** (*superheated*) –

Returns Total molar enthalpy [J/mol].

iapws95StateBlock Class

```
class idaes.generic_models.properties.iapws95.Iapws95StateBlock(*args,
                                                                **kws)
```

This is some placeholder doc.

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”

- **default** (*dict*) – Default Process-BlockData config

Keys

parameters A reference to an instance of the Property Parameter Block associated with this property package.

defined_state Flag indicating whether the state should be considered fully defined, and thus whether constraints such as sum of mass/mole fractions should be included, **default** - False. **Valid values:** { **True** - state variables will be fully defined, **False** - state variables will not be fully defined. }

has_phase_equilibrium Flag indicating whether phase equilibrium constraints should be constructed in this state block, **default** - True. **Valid values:** { **True** - StateBlock should calculate phase equilibrium, **False** - StateBlock should not calculate phase equilibrium. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Iapws95StateBlock) New instance

Iapws95StateBlockData Class

```
class idaes.generic_models.properties.iapws95.Iapws95StateBlockData (*args,
                                                                    **kwargs)
```

This is a property package for calculating thermophysical properties of water.

build (*args)
Callable method for Block construction

lapws95ParameterBlock Class

```
class idaes.generic_models.properties.iapws95.Iapws95ParameterBlock(*args,  
                                                                    **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

default_arguments Default arguments to use with Property Package

phase_presentation Set the way phases are presented to models. The MIX option appears to the framework to be a mixed phase containing liquid and/or vapor. The mixed option can simplify calculations at the unit model level since it can be treated as a single phase, but unit models such as flash vessels will not be able to treat the phases independently. The LG option presents as two separate phases to the framework. The L or G options can be used if it is known for sure that only one phase is present. **default** - PhaseType.MIX **Valid values:** { **PhaseType.MIX** - Present a mixed phase with liquid and/or vapor, **PhaseType.LG** - Present a liquid and vapor phase, **PhaseType.L** - Assume only liquid can be present, **PhaseType.G** - Assume only vapor can be present }

state_vars The set of state variables to use. Depending on the use, one state variable set or another may be better computationally. Usually pressure and enthalpy are the best choice because they are well behaved during a phase change. **default** - StateVars.PH **Valid values:** { **StateVars.PH** - Pressure-Enthalpy, **StateVars.TPX** - Temperature-Pressure-Quality }

temperature_bounds This is an optional

tuple providing default temperature bounds. The elements of the tuple should include units of temperature, for example, if `pyomo` units is imported as `pyunits`, `(270*pyunits.K, 1000*pyunits.K)`.

pressure_bounds This is an optional tuple providing default pressure bounds. The elements of the tuple should include units of pressure, for example, if `pyomo` units is imported as `pyunits`, `(1*pyunits.kPa, 1e6*pyunits.kPa)`.

enthalpy_mol_bounds This is an optional tuple providing default enthalpy per mole bounds. The elements of the tuple should include units of energy per mole, for example, if `pyomo` units is imported as `pyunits`, `(1*pyunits.J/pyunits.mol, 1e5*pyunits.J/pyunits.mol)`.

enthalpy_mass_bounds This is an optional tuple providing default enthalpy per mass bounds. The elements of the tuple should include units of energy per mass, for example, if `pyomo` units is imported as `pyunits`, `(1*pyunits.J/pyunits.kg, 1e5*pyunits.J/pyunits.kg)`.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (`Iapws95ParameterBlock`) New instance

Iapws95ParameterBlockData Class

class `idaes.generic_models.properties.iapws95.Iapws95ParameterBlockData` (*component*)

build()

General build method for Property-ParameterBlocks. Inheriting models should call `super().build`.

Parameters None –

Returns None

References

International Association for the Properties of Water and Steam (2016). IAPWS R6-95 (2016), “Revised Release on the IAPWS Formulation 1995 for the Properties of Ordinary Water Substance for General Scientific Use,” URL: <http://iapws.org/relguide/IAPWS95-2016.pdf>

Wagner, W., A. Pruss (2002). “The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use.” J. Phys. Chem. Ref. Data, 31, 387-535.

Wagner, W. et al. (2000). “The IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam,” ASME J. Eng. Gas Turbines and Power, 122, 150-182.

Akasaka, R. (2008). “A Reliable and Useful Method to Determine the Saturation State from Helmholtz Energy Equations of State.” Journal of Thermal Science and Technology, 3(3), 442-451.

International Association for the Properties of Water and Steam (2011). IAPWS R15-11, “Release on the IAPWS Formulation 2011 for the Thermal Conductivity of Ordinary Water Substance,” URL: <http://iapws.org/relguide/ThCond.pdf>.

International Association for the Properties of Water and Steam (2008). IAPWS R12-08, “Release on the IAPWS Formulation

2008 for the Viscosity of Ordinary Water Substance,” URL: <http://iapws.org/relguide/visc.pdf>.

Span-Wager CO2

This implements the Span-Wagner equation of state for CO2 “*Span-Wagner equation of state for CO2*” Please see the *general Helmholtz documentation* for more information.

Example

The Heater unit model *example*, provides a simple example for using water properties.

```
from idaes.generic_models.  
↳ properties import swco2  
from pyomo.environ import   
↳ ConcreteModel, units   
↳ as pyunits, SolverFactory   
↳ # Pyomo environment  
from   
↳ idaes.generic_models.unit_  
↳ models import Compressor  
from idaes.  
↳ core import FlowsheetBlock  
  
model = ConcreteModel()  
model.fs   
↳ = FlowsheetBlock(default=  
↳ {"dynamic": False})  
model.fs.properties =   
↳ swco2.SWCO2ParameterBlock()  
model.fs.  
↳ unit = Compressor(default=  
↳ {"property_package  
↳ ": model.fs.properties})  
F = 1000  
Tin = 500  
Pin = 10000  
Pout = 20000  
hin   
↳ = swco2.htpx(T=Tin*pyunits.  
↳ K, P=Pin*pyunits.Pa)  
  
model.fs.unit.  
↳ inlet.flow_mol[0].fix(F)  
model.fs.unit.  
↳ inlet.enth_mol[0].fix(hin)  
model.fs.unit.  
↳ inlet.pressure[0].fix(Pin)
```

(continues on next page)

(continued from previous page)

```

model.fs.
    ↪ unit.deltaP.fix(Pout - Pin)
model.fs.unit. efficiency_
    ↪ isentropic.fix(0.9)
model.
    ↪ fs.unit.initialize(optarg=
    ↪ {'tol': 1e-6})

solver_
    ↪ = SolverFactory("ipopt")
solver.solve(model)

```

For more information about how State-Blocks and PropertyParameterBlocks work see the [StateBlock documentation](#).

Expressions

The Span-Wager property package contains the standard expressions described in the [general Helmholtz documentation](#), but it also defines expressions for transport properties.

Expression	Description
<code>therm_cond_phase[phase]</code>	Thermal conductivity of phase (W/K/m)
<code>visc_d_phase[phase]</code>	Viscosity of phase (Pa/s)
<code>visc_k_phase[phase]</code>	Kinematic viscosity of phase (m ² /s)

Convenience Functions

`idaes.generic_models.properties.swco2.htpx` ($T=None$, $P=None$, $x=None$)

Convenience function to calculate enthalpy from temperature and either pressure or vapor fraction. This function can be used for inlet streams and initialization where temperature is known instead of enthalpy. User must provided values for two of T, P, or x.

Parameters

- **T** – Temperature with units (between 200 and 3000 K)
- **P** – Pressure with units (between 1 and 1e9 Pa), None if saturated vapor
- **x** – Vapor fraction [mol vapor/mol total] (between 0 and 1), None if superheated or subcooled

Returns Total molar enthalpy [J/mol].

SWCO2StateBlock Class

```
class idaes.generic_models.properties.swco2.SWCO2StateBlock (*args, **kws)
    This is some placeholder doc.
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

parameters A reference to an instance of the Property Parameter Block associated with this property package.

defined_state Flag indicating whether the state should be considered fully defined, and thus whether constraints such as sum of mass/mole fractions should be included, **default** - False. **Valid values:** { **True** - state variables will be fully defined, **False** - state variables will not be fully defined. }

has_phase_equilibrium Flag indicating whether phase equilibrium constraints should be constructed in this state block, **default** - True. **Valid values:** { **True** - StateBlock should calculate phase equilibrium, **False** - StateBlock should not calculate phase equilibrium. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (SWCO2StateBlock) New instance

SWCO2StateBlockData Class

```
class idaes.generic_models.properties.swco2.SWCO2StateBlockData(*args,
                                                                **kwargs)
```

This is a property package for calculating thermophysical properties of water.

```
build(*args)
    Callable method for Block construction
```

SWCO2ParameterBlock Class

```
class idaes.generic_models.properties.swco2.SWCO2ParameterBlock(*args,
                                                                **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

default_arguments Default arguments to use with Property Package

phase_presentation Set the way phases are presented to models. The MIX option appears to the framework to be a mixed phase containing liquid and/or vapor. The mixed option can simplify calculations at the unit model level since it can be treated as a single phase, but unit models such as flash vessels will not be able to treat the phases independently. The LG option presents as two separate phases to the framework. The L or G options can be used if it is known for sure that only one phase is present. **default** - PhaseType.MIX **Valid values:** { **PhaseType.MIX** - Present a mixed phase with liquid and/or vapor, **PhaseType.LG** - Present a liquid and vapor phase, **PhaseType.L** - Assume only liquid can be present, **PhaseType.G** - Assume only vapor can be present }

state_vars The set of state variables to use. Depending on the use, one state variable set or another may be better computationally. Usually pressure and enthalpy are the best choice because they are well behaved during a phase change. **default** - StateVars.PH **Valid values:** { **StateVars.PH** - Pressure-Enthalpy, **StateVars.TPX** - Temperature-Pressure-Quality }

temperature_bounds This is an optional tuple providing default temperature bounds. The elements of the tuple should include units of temperature, for example, if pyomo units is imported as pyunits, (270*pyunits.K, 1000*pyunits.K).

pressure_bounds This is an optional tuple providing default pressure bounds. The elements of the tuple should include units of pressure, for example, if pyomo units is imported as pyunits, (1*pyunits.kPa, 1e6*pyunits.kPa).

enthalpy_mol_bounds This is an optional tuple providing default enthalpy per mole bounds. The elements of the tuple should include units of energy per mole, for example, if pyomo units is imported as pyunits, (1*pyunits.J/pyunits.mol, 1e5*pyunits.J/pyunits.mol).

enthalpy_mass_bounds This is an optional tuple providing default enthalpy per mass bounds. The elements of the tuple should include units of energy per mass, for example, if pyomo units is imported as pyunits, (1*pyunits.J/pyunits.kg, 1e5*pyunits.J/pyunits.kg).

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (SWCO2ParameterBlock) New instance

SWCO2ParameterBlockData Class

class `idaes.generic_models.properties.swco2.SWCO2ParameterBlockData` (*component*)

build()

General build method for Property-ParameterBlocks. Inheriting models should call `super().build`.

Parameters None –

Returns None

References

Span, R., W. Wagner, W. (1996). “A New Equation of State for Carbon Dioxide Covering the Fluid Region from the Triple-Point Temperature to 1100 K at Pressures up to 800 MPa.” J. Phys. Chem. Ref. Data, 25(6), 1509-1596.

Akasaka, R. (2008). “A Reliable and Useful Method to Determine the Saturation State from Helmholtz Energy Equations of State.” Journal of Thermal Science and Technology, 3(3), 442-451.

Vesovic, V., W.A. Wakeham, G.A. Olchow, J.V. Sengers, J.T.R. Watson, J. Millat, (1990). “The transport properties of carbon dioxide.” J. Phys. Chem. Ref. Data, 19, 763-808.

Fenghour, A., W.A. Wakeham, V. Vesovic, (1998). “The Viscosity of Carbon Dioxide.” J. Phys. Chem. Ref. Data, 27, 31-44.

The basic Helmholtz EoS is described “*Revised Release on the IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use.*”. The Helmholtz EoS as used in the IAPWS-95 contains non-analytic terms to improve accuracy near the critical point. These terms, however cause a singularity at the critical point and can causes

computational difficulty, so the non-analytic where omitted in the IDAES implementation.

The IDAES implementation is of the Helmholtz EoS makes use of external function for many of the properties. Solving the VLE and changing state variables require solution of nonlinear equations with multiple solutions, so solving then externally provides a method of decomposition where it can be guaranteed that the nonlinear equations associated with the Helmholtz EoS are solved correctly. The external functions provide first and second derivatives, and are compatible with advanced optimization solvers. Phase change does cause problems to be non-smooth, but, as a practical matter, problem using the IDAES implementation of the Helmholtz EoS, still seem to solve well even with phase change.

Units

The iapws95 property module uses SI units (m, kg, s, J, mol) for all public variables and expressions. Temperature is in K. Note that this means molecular weight is in the unusual unit of kg/mol.

A few expressions intended used internally and all external function calls use units of kg, kJ, kPa, and K. These generally are not needed by the end user.

Phase Presentation

The property package wrapper can present fluid phase information to the IDAES framework in different ways. For specifics on how to set the options see a specific implementation page.

The `PhaseType.MIX` option causes to modeling framework to view water and steam as a single mixed liquid and vapor phase. This generally reduces model complexity. Phase equilibrium is still calculated and `vapor_frac` and individual phase properties are available, just are they would be with the

two-phase presentation. The mixed-phase presentation can be used with most standard unit models that do not provide phase separation. If phase separation is required, either use the two-phase presentation or create a custom model.

The `PhaseType.LG` option appears to the IDAES framework to be two phases “Vap” and “Liq”. This option requires one of two unit model options to be set. You can use the total material balance option for unit models, to specify that only one material balance equation should be written not one per phase. The other possible option is to specify `has_phase_equilibrium=True`. This will write a material balance per phase, but will add a phase generation term to the model. For Helmholtz EoS packages, it is generally recommended that specifying total material balances is best because it results in a problem with fewer variables, and phase equilibrium is always calculated by the property package.

There are two single phase options `PhaseType.L` and `PhaseType.G`; these present a single phase “Liq” or “Vap” to the framework. The vapor fraction will also always return 0 or 1 as appropriate. These options can be used when the phase of a fluid is known for certain to only be liquid or only be vapor. For the temperature-pressure-vapor fraction formulation, this eliminates the complementarity constraint, but for the enthalpy-pressure formulation, where the vapor fraction is always calculated, the single phase options probably do not provide any real benefit.

State Variables

There is a choice of state variables, pressure-enthalpy and temperature-pressure-vapor fraction. In general the enthalpy-pressure form is preferable. Both the pressure and enthalpy variables are smooth and sufficient to define the fluid state. For systems where two-phases may be present, it is expected that pressure-enthalpy is the best choice of state variables.

The temperature-pressure-vapor fraction form is more convenient, since temperature is directly measurable and more familiar than enthalpy. Complementarity constraints are used to deal with the vapor fraction variable, but the additional complimentary constraints may make the problem less robust. Temperature-pressure is often a good choice of state variables where there is only a single known phase.

Pressure-Enthalpy Formulation

The advantage of this choice of state variables is that it is more robust when phase changes occur, and is especially useful when it is not known if a phase change will occur. The disadvantage of this choice of state variables is that for equations like heat transfer equations that are highly dependent on temperature, a model could be harder to solve near regions with phase change. Temperature is a non-smooth function with non-smoothness when transitioning from the single-phase to the two-phase region. Temperature also has a zero derivative with respect to enthalpy in the two-phase region, so near the two-phase region solving a constraint that specifies a specific temperature may be difficult.

The variables for this form are `flow_mol` (mol/s), `pressure` (Pa), and `enth_mol` (J/mol).

Since temperature and vapor fraction are not state variables in this formulation, they are provided by expressions,

and cannot be fixed. For example, to set a temperature to a specific value, a constraint could be added which says the temperature expression equals a fixed value.

These expressions are specific to the P-H formulation:

temperature Expression that calculates temperature by calling an ExternalFunction of enthalpy and pressure. This expression is non-smooth in the transition from single-phase to two-phase and has a zero derivative with respect to enthalpy in the two-phase region.

vapor_frac Expression that calculates vapor fraction by calling an ExternalFunction of enthalpy and pressure. This expression is non-smooth in the transition from single-phase to two-phase and has a zero derivative with respect to enthalpy in the single-phase region, where the value is 0 (liquid) or 1 (vapor).

Temperature-Pressure-Vapor Fraction

This formulation uses temperature (K), pressure (Pa), and vapor fraction as state variables. When a single phase option is given, the vapor fraction is fixed to the appropriate value and the complementarity constraint is deactivated.

A complementarity constraint is required for the T-P-x formulation when two-phases may be present. First, two expressions are defined below where P^- is pressure under saturation pressure and P^+ is pressure over saturation pressure. The `max()` function is provided as an IDAES utility which provides a smooth max expression.

$$P^- = \max(0, P_{\text{sat}} - P)$$

$$P^+ = \max(0, P - P_{\text{sat}})$$

With the “pressure over” and “pressure under” expressions a complementarity constraint can be written. If the pressure under saturation is more than zero, only vapor exists. If the pressure over saturation is greater than zero only a liquid exists. If both are about zero two phases

can exist. The saturation pressure function maxes out at the critical pressure and any temperature above the critical temperature will yield a saturation pressure that is the critical pressure, so supercritical fluids will be classified as liquids as is the convention for this property package.

$$0 = xP^+ - (1 - x)P^-$$

Assuming the vapor fraction (x) is positive and noting that only one of P^+ and P^- can be nonzero (approximately), the complementarity equation above requires x to be 0 when P^+ is not zero (liquid) or x to be 1 when P^- is not zero (vapor). When both P^+ and P^- are about 0, the complementarity constraint says nothing about x , but it basically reduces another constraint, that $P = P_{\text{sat}}$. When two phases are present x is found by the unit model energy balance, where the temperature will be T_{sat} (because $P = P_{\text{sat}}$).

An alternative approach is sometimes useful to simplify the problem when it is certain that there are two phases. The complementarity constraint can be deactivated and a $P = P_{\text{sat}}$ or $T = T_{\text{sat}}$ constraint can be added.

Using the T-P-x formulation requires better initial guesses than the P-H form. It is not strictly necessary but it is best to try to get an initial guess that is in the correct phase region for the expected result model.

Expressions

Unless otherwise noted, the property expressions are common to both the T-P-x and P-H formulations. For phase specific properties, valid phase indexes are "Liq" and "Vap". Even when using the mixed phase version of the property package, both liquid and vapor properties are available.

Expression	Description
mw	Molecular weight (kg/mol)
tau	Critical temperature divided by temperature (unitless)
temperature	Temperature (K) if PH form
temperature_red	Reduced temperature, temperature divided by critical temperature (unitless)
temperature_sat	Saturation temperature (K)
tau_sat	Critical temperature divided by saturation temperature (unitless)
pressure_sat	Saturation pressure (Pa)
dens_mass_phase[phase]	Density phase (kg/m ³)
dens_phase_red[phase]	Phase reduced density (δ), mass density divided by critical density (unitless)
dens_mass	Total mixed phase mass density (kg/m ³)
dens_mol	Total mixed phase mole density (kg/m ³)
flow_vol	Total volumetric flow rate (m ³ /s)
enth_mass	Mass enthalpy (J/kg)
enth_mol_sat_phase[phase]	Saturation enthalpy of phase, enthalpy at P and T_{sat} (J/mol)

continues on next page

Table 7 – continued from previous page

Expression	Description
enth_mol	Molar enthalpy (J/mol) if TPx form
enth_mol_phase[phase]	Molar enthalpy of phase (J/mol)
energy_internal_mol	molar internal energy (J/mol)
energy_internal_mol_phase[phase]	Molar internal energy of phase (J/mol)
entr_mol_phase	Molar entropy of phase (J/mol/K)
entr_mol	Total mixed phase entropy (J/mol/K)
cp_mol_phase[phase]	Constant pressure molar heat capacity of phase (J/mol/K)
cv_mol_phase[phase]	Constant pressure volume heat capacity of phase (J/mol/K)
cp_mol	Total mixed phase constant pressure heat capacity (J/mol/K)
cv_mol	Total mixed phase constant volume heat capacity (J/mol/K)
heat_capacity_ratio	cp_mol/cv_mol
speed_sound_phase[phase]	Speed of sound in phase (m/s)
dens_mol_phase[phase]	Mole density of phase (mol/m ³)
vapor_frac	Vapor fraction, if PH form
phase_frac[phase]	Phase fraction
flow_mol_comp["H2O"]	Same as total flow since only water (mol/s)
P_under_sat	Pressure under saturation pressure (kPa)
P_over_sat	Pressure over saturation pressure (kPa)

ExternalFunctions

This provides a list of ExternalFunctions available in the wrapper. These functions do not use SI units and are not usually called directly. If these functions are needed, they should be used with caution. Some of these are used in the property expressions, some are just provided to allow easier testing with a Python framework.

All of these functions provide first and second derivative and are generally suited to optimization (including the ones that return derivatives of Helmholtz free energy). Some functions may have non-smoothness at phase transitions. The `delta_vap` and `delta_liq` functions return the same values in the critical region. They will also return real values when a phase doesn't exist, but those values do not necessarily have physical meaning.

There are a few variables that are common to a lot of these functions, so they are summarized here τ is the critical temperature divided by the temperature $\frac{T}{T_c}$, δ is density divided by the critical density $\frac{\rho}{\rho_c}$, and ϕ is Helmholtz free energy divided by the ideal gas constant

and temperature $\frac{f}{RT}$.

Object	C Function	Returns	Arguments
func_p	p	pressure (kPa)	δ, τ
func_p_stau	p_stau	pressure (kPa)	s (kJ/kg/K), τ
func_u	u	internal energy (kJ/kg)	δ, τ
func_s	s	entropy (kJ/K/kg)	δ, τ
func_h	h	enthalpy (kJ/kg)	δ, τ
func_hvpt	hvpt	vapor enthalpy (kJ/kg)	P (kPa), τ
func_hlpt	hlpt	liquid enthalpy (kJ/kg)	P (kPa), τ
func_svpt	svpt	vapor entropy (kJ/kg/K)	P (kPa), τ
func_slpt	slpt	liquid entropy (kJ/kg/K)	P (kPa), τ
func_uvpt	uvpt	vapor internal energy (kJ/kg)	P (kPa), τ
func_ulpt	ulpt	liquid internal energy (kJ/kg)	P (kPa), τ
func_tau	tau	τ (unitless)	h (kJ/kg), P (kPa)
func_tau_sp	tau_sp	τ (unitless)	s (kJ/kg/K), P (kPa)
func_tau_up	tau_up	τ (unitless)	u (kJ/kg), P (kPa)
func_vf	vf	vapor fraction (unitless)	h (kJ/kg), P (kPa)
func_vfs	vfs	vapor fraction (unitless)	s (kJ/kg/K), P (kPa)
func_vfu	vfu	vapor fraction (unitless)	u (kJ/kg), P (kPa)
func_g	g	Gibbs free energy (kJ/kg)	δ, τ
func_f	f	Helmholtz free energy (kJ/kg)	δ, τ
func_cv	cv	const. volume heat capacity (kJ/K/kg)	δ, τ
func_cp	cp	const. pressure heat capacity (kJ/K/kg)	δ, τ
func_w	w	speed of sound (m/s)	δ, τ
func_delta_liq	delta_liq	liquid δ (unitless)	P (kPa), τ
func_delta_vap	delta_vap	vapor δ (unitless)	P (kPa), τ
func_delta_sat_l	delta_sat_l	sat. liquid δ (unitless)	τ
func_delta_sat_v	delta_sat_v	sat. vapor δ (unitless)	τ
func_p_sat	p_sat	sat. pressure (kPa)	τ
func_tau_sat	tau_sat	sat. τ (unitless)	P (kPa)
func_phi0	phi0	ϕ idaes gas part (unitless)	δ, τ
func_phi0_delta	phi0_delta	$\frac{\partial \phi_0}{\partial \delta}$	δ
func_phi0_delta2	phi0_delta2	$\frac{\partial^2 \phi_0}{\partial \delta^2}$	δ
func_phi0_tau	phi0_tau	$\frac{\partial \phi_0}{\partial \tau}$	τ
func_phi0_tau2	phi0_tau2	$\frac{\partial^2 \phi_0}{\partial \tau^2}$	τ
func_phir	phir	ϕ real gas part (unitless)	δ, τ
func_phir_delta	phir_delta	$\frac{\partial \phi_r}{\partial \delta}$	δ, τ
func_phir_delta2	phir_delta2	$\frac{\partial^2 \phi_r}{\partial \delta^2}$	δ, τ
func_phir_tau	phir_tau	$\frac{\partial \phi_r}{\partial \tau}$	δ, τ
func_phir_tau2	phir_tau2	$\frac{\partial^2 \phi_r}{\partial \tau^2}$	δ, τ
func_phir_delta_tau	phir_delta_tau	$\frac{\partial^2 \phi_r}{\partial \delta \partial \tau}$	δ, τ

Initialization

The IAPWS-95 property functions do provide initialization functions for general compatibility with the IDAES framework, but as long as the state variables are specified to some reasonable value, initialization is not required. All required solves are handled by external functions.

References

Although a general Helmholtz EoS was developed, the equations were taken from the IAPWS-95 standard. For specific parameter sources see specific implementation documentation.

International Association for the Properties of Water and Steam (2016). IAPWS R6-95 (2016), “Revised Release on the IAPWS Formulation 1995 for the Properties of Ordinary Water Substance for General Scientific Use,” URL: <http://iapws.org/relguide/IAPWS95-2016.pdf>

Wagner, W., A. Pruss (2002). “The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use.” J. Phys. Chem. Ref. Data, 31, 387-535.

Akasaka, R. (2008). “A Reliable and Useful Method to Determine the Saturation State from Helmholtz Energy Equations of State.” Journal of Thermal Science and Technology, 3(3), 442-451.

Property Interrogator Tool

When preparing to model a process flowsheet, it is necessary to specify models for all the thermophysical and kinetic properties that will be required by the different unit operations to simulate the process. However, it is often difficult to know what properties will be required *a priori*. The IDAES Property Interrogator tool allows a user to define a general flowsheet structure and

interrogate it for the full list of properties that will be required, thus informing them of what methods they will need to define in their property package(s).

Tool Usage

The IDAES Properties Interrogator tool consists of two classes; a *PropertiesInterrogatorBlock* and a *ReactionInterrogatorBlock*. These blocks are used in place of the normal *PhysicalParameterBlock* and *ReactionParameterBlock* whilst declaring a flowsheet, however rather than constructing a solvable flowsheet they record all calls for properties made whilst constructing the flowsheet. These Blocks then contain a number of methods for reporting the logged property calls for the user.

An example of how Property Interrogator tool is used is shown below:

```
import pyomo.environ
↳ as pyo # Pyomo environment
from idaes.
↳ core import FlowsheetBlock
from idaes.generic_models.
↳ unit_models import CSTR
from idaes.
↳ generic_models.properties.
↳ interrogator import
↳ PropertyInterrogatorBlock,
↳ ReactionInterrogatorBlock

m = pyo.ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": True})

m.fs.params =_
↳ PropertyInterrogatorBlock()
m.fs.rxn_params =_
↳ ReactionInterrogatorBlock(
    default={"property_
↳ package": m.fs.params})

m.fs.R01_
↳ = CSTR(default={"property_
↳ package": m.fs.params,
    "reaction_
↳ package": m.fs.rxn_params,
    "has_
↳ heat_of_reaction": True})
```

(continues on next page)

Note: Flowsheets constructed using the Property Interrogator tools are not solvable flowsheets, and will result in errors if sent to a solver.

Output and Display Methods

Both the *PropertiesInterrogatorBlock* and *ReactionInterrogatorBlock* support the following methods for reporting the results of the flowsheet interrogation. The *PropertiesInterrogatorBlock* will contain a summary of all thermo-physical properties expected of a *StateBlock* in the flowsheet, whilst the *ReactionInterrogatorBlock* will contain a summary of all reaction related properties required of a *ReactionBlock*.

- `list_required_properties()` - returns a list containing all properties called for by the flowsheet.
- `print_required_properties()` - prints a summary of the required properties
- `list_models_requiring_property(property)` - returns a list of unit models within the flowsheet that require the given property
- `print_models_requiring_property(property)` - prints the name of all unit models within the flowsheet that require the given property
- `list_properties_required_by_model(model)` - returns a list of all properties required by a given unit model in the flowsheet
- `print_properties_required_by_model(model)` - prints a summary of all properties required by a given unit model in the flowsheet

For more details on these methods, see the detailed class documentation below.

Additionally, the *PropertiesInterrogatorBlock* and *ReactionInterrogatorBlock* contain a *dict* named *required_properties* which stores the data regarding the properties required by the model. The keys of this *dict* are

the names of all the properties required (as strings) and the values are a list of names for the unit models requiring the given property.

Class Documentation

class `idaes.generic_models.properties.interrogator.properties_interrogator.PropertyInterrogator`

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

default_arguments Default arguments to use with Property Package

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (PropertyInterrogatorBlock) New instance

class `idaes.generic_models.properties.interrogator.properties_interrogator.PropertyInterrogator`
Interrogator Parameter Block Class

This class contains the methods and attributes for recording and displaying the properties required by the flowsheet.

build()

Callable method for Block construction.

classmethod define_metadata (*obj*)
Set all the metadata for properties and units.

This method should be implemented by subclasses. In the implementation, they should set information into the object provided as an argument.

Parameters `pcm`

(*PropertyClassMetadata*) – Add metadata to this object.

Returns None

`list_models_requiring_property(prop)`

Method to list all models in the flowsheet requiring the given property.

Parameters `prop` – the property of interest

Returns A list of unit model names which require `prop`

`list_properties_required_by_model(model)`

Method to list all thermophysical properties required by a given unit model.

Parameters `model` – the unit model of interest. Can be given as either a model component or the unit name as a string

Returns A list of thermophysical properties required by `model`

`list_required_properties()`

Method to list all thermophysical properties required by the flowsheet.

Parameters None –

Returns A list of properties required

`print_models_requiring_property(prop, ostream=None)`

Method to print a summary of the models in the flowsheet requiring a given property.

Parameters

- `prop` – the property of interest.
- `ostream` – output stream to print to. If not provided will print to `sys.stdout`

Returns None

`print_properties_required_by_model(model, ostream=None)`

Method to print a summary of the thermophysical properties required by a given unit model.

Parameters

- `model` – the unit model of interest.

- **ostream** – output stream to print to. If not provided will print to `sys.stdout`

Returns None

print_required_properties (*ostream=None*)

Method to print a summary of the thermophysical properties required by the flowsheet.

Parameters **ostream** – output stream to print to. If not provided will print to `sys.stdout`

Returns None

class `idaes.generic_models.properties.interrogator.reactions_interrogator.ReactionInterrogatorBlock`

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

property_package Reference to associated PropertyPackageParameter object

default_arguments Default arguments to use with Property Package

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (`ReactionInterrogatorBlock`) New instance

class `idaes.generic_models.properties.interrogator.reactions_interrogator.ReactionInterrogatorBlock`

Interrogator Parameter Block Class

This class contains the methods and attributes for recording and displaying the reaction properties required by the flow-sheet.

build()

Callable method for Block construction.

classmethod define_metadata (*obj*)

Set all the metadata for properties and units.

This method should be implemented by subclasses. In the implementation, they should set information into the object provided as an argument.

Parameters **pcm**

(*PropertyClassMetadata*) – Add metadata to this object.

Returns None

list_models_requiring_property (*prop*)

Method to list all models in the flow-sheet requiring the given property.

Parameters **prop** – the property of interest

Returns A list of unit model names which require prop

list_properties_required_by_model (*model*)

Method to list all reaction properties required by a given unit model.

Parameters **model** – the unit model of interest. Can be given as either a model component or the unit name as a string

Returns A list of reaction properties required by model

list_required_properties ()

Method to list all reaction properties required by the flowsheet.

Parameters None –

Returns A list of properties required

print_models_requiring_property (*prop, ostream=None*)

Method to print a summary of the models in the flowsheet requiring a given property.

Parameters

- **prop** – the property of interest.
- **ostream** – output stream to print to. If not provided will print to sys.stdout

Returns None

print_properties_required_by_model (*model*, *ostream=None*)

Method to print a summary of the reaction properties required by a given unit model.

Parameters

- **model** – the unit model of interest.
- **ostream** – output stream to print to. If not provided will print to sys.stdout

Returns None

print_required_properties (*ostream=None*)

Method to print a summary of the reaction properties required by the flowsheet.

Parameters **ostream** – output stream to print to. If not provided will print to sys.stdout

Returns None

Unit Models

Compressor

The Compressor model is a *PressureChanger*, where the configuration is set so that the “compressor” option can only be True, and the default “thermodynamic_assumption” is “isentropic.” See the *PressureChanger documentation* for details.

Example

The example below demonstrates the basic Compressor model usage:

```
import pyomo.environ as pyo
from idaes.
↳core import FlowsheetBlock
from_
↳idaes.generic_models.unit_
↳models import Compressor
from idaes.generic_models.
↳properties import iapws95

m = pyo.ConcreteModel()
m.fs_
↳= FlowsheetBlock(default=
↳{"dynamic": False})
```

(continues on next page)

(continued from previous page)

```

m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.fs.
↳ unit = Compressor(default=
↳ {"property_
↳ package": m.fs.properties})

m.fs.unit.
↳ inlet.flow_mol[0].fix(100)
m.fs.unit.
↳ inlet.enth_mol[0].fix(4000)
m.fs.unit.inlet.
↳ pressure[0].fix(101325)

m.fs.unit.deltaP.fix(50000)
m.fs.unit.entropy_
↳ isentropic.fix(0.9)

```

Continuous Stirred Tank Reactor

The IDAES CSTR model represents a unit operation where a material stream undergoes some chemical reaction(s) in a well-mixed vessel.

Degrees of Freedom

CSTRs generally have one degree of freedom. Typically, the fixed variable is reactor volume.

Model Structure

The core CSTR unit model consists of a single `ControlVolume0D` (named `control_volume`) with one `Inlet Port` (named `inlet`) and one `Outlet Port` (named `outlet`).

Additional Constraints

CSTR units write the following additional Constraints beyond those written by the `ControlVolume` Block.

$$X_{t,r} = V_t \times r_{t,r}$$

where $X_{t,r}$ is the extent of reaction of reaction r at time t , V_t is the volume of the reacting material at time t (allows

for varying reactor volume with time) and $r_{t,r}$ is the volumetric rate of reaction of reaction r at time t (from the outlet property package).

Variables

CSTR units add the following additional Variables beyond those created by the ControlVolume Block.

Variable	Name	Notes
V_t	volume	If <code>has_holdup = True</code> this is a reference to <code>control_volume.volume</code> , otherwise a Var attached to the Unit Model
Q_t	heat	Only if <code>has_heat_transfer = True</code> , reference to <code>control_volume.heat</code>

CSTR Class

```
class idaes.generic_models.unit_models.cstr.CSTR(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be

constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - False. **Valid values:** { **True** - include heat transfer terms, **False** - exclude heat transfer terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms.}

has_heat_of_reaction Indicates whether terms for heat of reaction terms should be constructed, **default** - False. **Valid values:** { **True** - include heat of reaction terms, **False** - exclude heat of reaction terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PhysicalParameterObject** - a **PhysicalParameterBlock** object.}

property_package_args A **ConfigBlock** with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a **ReactionParameterBlock** object.}

reaction_package_args A **ConfigBlock** with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (CSTR) New instance

CSTRData Class

class `idaes.generic_models.unit_models.cstr.CSTRData` (*component*)
Standard CSTR Unit Model Class

build ()
Begin building model (pre-DAE transformation). :param None:

Returns None

Equilibrium Reactor

The IDAES Equilibrium reactor model represents a unit operation where a material stream undergoes some chemical reaction(s) to reach an equilibrium state. This model is for systems with reaction with equilibrium coefficients - for Gibbs energy minimization see Gibbs reactor documentation.

Degrees of Freedom

Equilibrium reactors generally have 1 degree of freedom.

Typical fixed variables are:

- reactor heat duty (`has_heat_transfer = True` only).

Model Structure

The core Equilibrium reactor unit model consists of a single `ControlVolume0D` (named `control_volume`) with one Inlet Port (named `inlet`) and one Outlet Port (named `outlet`).

Additional Constraints

Equilibrium reactors units write the following additional Constraints beyond those written by the Control Volume if rate controlled reactions are present.

$$r_{t,r} = 0$$

where $r_{t,r}$ is the rate of reaction for reaction r at time t . This enforces equilibrium in any reversible rate controlled reactions which are present. Any non-reversible reaction that may be present will proceed to completion.

Variables

Equilibrium reactor units add the following additional Variables beyond those created by the Control Volume.

Variable	Name	Notes
V_t	volume	If <code>has_holdup = True</code> this is a reference to <code>control_volume.volume</code> , otherwise a <code>Var</code> attached to the Unit Model
Q_t	heat	Only if <code>has_heat_transfer = True</code> , reference to <code>control_volume.heat</code>

EquilibriumReactor Class

```
class idaes.generic_models.unit_models.equilibrium_reactor.EquilibriumReactor(*args,
                                                                              **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”

- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = False. Equilibrium Reactors do not support dynamic behavior.

has_holdup Indicates whether holdup terms should be constructed or not. **default** - False. Equilibrium reactors do not have defined volume, thus this must be False.

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.useDefault`. **Valid values:** { `MaterialBalanceType.useDefault` - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, `MaterialBalanceType.componentPhase` - use phase component balances, `MaterialBalanceType.componentTotal` - use total component balances, `MaterialBalanceType.elementTotal` - use total element balances, `MaterialBalanceType.total` - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { `EnergyBalanceType.useDefault` - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, `EnergyBalanceType.enthalpyTotal` - single enthalpy balance for material, `EnergyBalanceType.enthalpyPhase` - enthalpy balances for each phase, `EnergyBalanceType.energyTotal` - single energy balance for material, `EnergyBalanceType.energyPhase` - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { `MomentumBalanceType.none` - exclude momen-

tum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_rate_reactions Indicates whether terms for rate controlled reactions should be constructed, along with constraints equating these to zero, **default** - True. **Valid values:** { **True** - include rate reaction terms, **False** - exclude rate reaction terms.}

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** - True. **Valid values:** { **True** - include phase equilibrium term, **False** - exclude phase equilibrium terms.}

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - False. **Valid values:** { **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_heat_of_reaction Indicates whether terms for heat of reaction terms should be constructed, **default** - False. **Valid values:** { **True** - include heat of reaction terms, **False** - exclude heat of reaction terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid**

values: { **useDefault** - use default package from parent model or flow-sheet, **PhysicalParameterObject** - a **PhysicalParameterBlock** object. }

property_package_args A **ConfigBlock** with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a **ReactionParameterBlock** object. }

reaction_package_args A **ConfigBlock** with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (**EquilibriumReactor**) New instance

EquilibriumReactorData Class

class `idaes.generic_models.unit_models.equilibrium_reactor.EquilibriumReactorData` (*component*)
Standard Equilibrium Reactor Unit
Model Class

build()
Begin building model.

Parameters **None** –

Returns **None**

Feed Block

Feed Blocks are used to represent sources of material in Flowsheets. Feed blocks do not calculate phase equilibrium of the feed stream, and the composition of the material in the outlet stream will be exactly as specified in the input. For applications where the users wishes the outlet stream to be in phase equilibrium, see the Feed_Flash unit model.

Degrees of Freedom

The degrees of freedom of Feed blocks depends on the property package being used and the number of state variables necessary to fully define the system. Users should refer to documentation on the property package they are using.

Model Structure

Feed Blocks consists of a single State-Block (named properties), each with one Outlet Port (named outlet). Feed Blocks also contain References to the state variables defined within the State-Block

Additional Constraints

Feed Blocks write no additional constraints to the model.

Variables

Feed blocks add no additional Variables.

Feed Class

```
class idaes.generic_models.unit_models.feed.Feed(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = False. Feed blocks are always steady-state.

has_holdup Feed blocks do not contain holdup, thus this must be False.

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Feed) New instance

FeedData Class

class `idaes.generic_models.unit_models.feed.FeedData` (*component*)
Standard Feed Block Class

build()
Begin building model.

Parameters None –

Returns None

initialize (*state_args=None, outlvl=0, solver='ipopt', optarg={'tol': 1e-06}*)
This method calls the initialization method of the state block.

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = None).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

Feed Block with Flash

Feed Blocks are used to represent sources of material in Flowsheets. In some cases, users may have a situation where a feed stream may be in a multi-phase state, but may not know the full details of the equilibrium state. The IDAES Feed Block with Flash (Feed-Flash) allows users to define a feed block where the outlet is in phase equilibrium based on calculations from the chosen property package and a sufficient set of state variables prior to being passed to the first unit operation. The phase equilibrium is performed assuming an isobaric and isothermal flash operation.

A Feed Block with Flash is only required in cases where the feed may be in phase equilibrium AND the chosen

property package uses a state definition that includes phase separations. Some property packages support phase equilibrium, but use a state definition that involves only total flows - in these cases a flash calculation is performed at the inlet of every unit and thus it is not necessary to perform a flash calculation at the feed block.

Degrees of Freedom

The degrees of freedom of FeedFlash blocks depends on the property package being used and the number of state variables necessary to fully define the system. Users should refer to documentation on the property package they are using.

Model Structure

FeedFlash Blocks contain a single ControlVolume0D (named `control_volume`) with one Outlet Port (named `outlet`). FeedFlash Blocks also contain References to the state variables defined within the inlet StateBlock of the ControlVolume (representing the unflashed state of the feed).

FeedFlash Blocks do not write a set of energy balances within the Control Volume - instead a constraint is written which enforces an isothermal flash.

Additional Constraints

The FeedFlash Block writes one additional constraint to enforce isothermal behavior.

$$T_{in,t} = T_{out,t}$$

where $T_{in,t}$ and $T_{out,t}$ are the temperatures of the material before and after the flash operation.

Variables

FeedFlash blocks add no additional Variables.

FeedFlash Class

```
class idaes.generic_models.unit_models.feed_flash.FeedFlash(*args,**kwds)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Feed units do not support dynamic behavior.

has_holdup Feed units do not have defined volume, thus this must be False.

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

flash_type Indicates what type of flash operation should be used. **default** - FlashType.isothermal. **Valid values:** { **FlashType.isothermal** - specify temperature, **FlashType.isenthalpic** - specify enthalpy. }

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (FeedFlash) New instance

FeedFlashData Class

class `idaes.generic_models.unit_models.feed_flash.FeedFlashData` (*component*)
Standard Feed block with phase equilibrium

build()
Begin building model.

Parameters None –

Returns None

Flash Unit

The IDAES Flash model represents a unit operation where a single stream undergoes a flash separation into two phases. The Flash model supports multiple types of flash operations, including pressure changes and addition or removal of heat.

Degrees of Freedom

Flash units generally have 2 degrees of freedom.

Typical fixed variables are:

- heat duty or outlet temperature (see note),
- pressure change or outlet pressure.

Note: When setting the outlet temperature of a Flash unit, it is best to set `control_volume.properties_out[t].temperature`.

Setting the temperature in one of the outlet streams directly results in a much harder problem to solve, and may be degenerate or unbounded in some cases.

Model Structure

The core Flash unit model consists of a single `ControlVolume0DBlock` (named `control_volume`) with one Inlet Port (named `inlet`) connected to a Separator unit model with two outlet Ports named ‘`vap_outlet`’ and ‘`liq_outlet`’. The Flash model utilizes the separator unit model in IDAES to split the outlets by phase flows to the liquid and vapor outlets respectively.

The Separator unit model supports both direct splitting of state variables and writing of full splitting constraints via the *ideal_separation* construction argument. Full details on the Separator unit model can be found in the documentation for that unit. To support direct splitting, the property package must use one of a specified set of state variables and support a certain set of property calculations, as outlined in the table below.

State Variables	Required Properties
Material flow and composition	
flow_mol & mole_frac	flow_mol_phase & mole_frac_phase
flow_mol_phase & mole_frac_phase	flow_mol_phase & mole_frac_phase
flow_mol_comp	flow_mol_phase_comp
flow_mol_phase_comp	flow_mol_phase_comp
flow_mass & mass_frac	flow_mass_phase & mass_frac_phase
flow_mass_phase & mass_frac_phase	flow_mass_phase & mass_frac_phase
flow_mass_comp	flow_mass_phase_comp
flow_mass_phase_comp	flow_mass_phase_comp
Energy state	
temperature	temperature
enth_mol	enth_mol_phase
enth_mol_phase	enth_mol_phase
enth_mass	enth_mass_phase
enth_mass_phase	enth_mass_phase
Pressure state	
pressure	pressure

Construction Arguments

Flash units have the following construction arguments:

- `property_package` - property package to use when constructing Property Blocks (default = 'use_parent_value'). This is provided as a Property Parameter Block by the Flowsheet when creating the model. If a value is not provided, the Holdup Block will try to use the default property package if one is defined.
- `property_package_args` - set of arguments to be passed to the Property Blocks when they are created.

Additionally, Flash units have the following construction arguments which are passed to the Holdup Block for determining which terms to construct in the balance equations.

Argument	Default Value
<code>dynamic</code>	False
<code>include_holdup</code>	False
<code>material_balance_type</code>	MaterialBalanceType.componentPhase
<code>energy_balance_type</code>	EnergyBalanceType.enthalpyTotal
<code>momentum_balance_type</code>	MomentumBalanceType.pressureTotal
<code>has_phase_equilibrium</code>	True
<code>has_heat_transfer</code>	True
<code>has_pressure_change</code>	True

Finally, Flash units also have the following arguments which are passed to the Separator block for determining how to split to two-phase mixture.

Argument	Default Value
ideal_separation	True
energy_split_basis	EnergySplittingType.equal_temperature

Additional Constraints

Flash units write no additional Constraints beyond those written by the ControlVolume0DBlock and the Separator block.

Variables

Name	Notes
heat_duty	Reference to control_volume.heat
deltaP	Reference to control_volume.deltaP

Flash Class

```
class idaes.generic_models.unit_models.flash.Flash(*args, **kwds)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = False. Flash units do not support dynamic behavior.

has_holdup Indicates whether holdup terms should be constructed or not. **default** - False. Flash units do not have defined volume, thus this must be False.

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.useDefault`. **Valid values:** { `MaterialBalanceType.useDefault` - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { `EnergyBalanceType.useDefault` - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { `MomentumBalanceType.none` - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

energy_split_basis Argument indicating basis to use for splitting energy this is not used for when

`ideal_separation == True`. **default** - `EnergySplittingType.equal_temperature`.

Valid values: { **EnergySplittingType.equal_temperature** - outlet temperatures equal inlet **EnergySplittingType.equal_molar_enthalpy** - outlet molar enthalpies equal inlet, **EnergySplittingType.enthalpy_split** - apply split fractions to enthalpy flows. }

ideal_separation Argument indicating whether ideal splitting should be used. Ideal splitting assumes perfect separation of material, and attempts to avoid duplication of StateBlocks by directly partitioning outlet flows to ports, **default** - `True`. **Valid values:** { **True** - use ideal splitting methods. Cannot be combined with `has_phase_equilibrium = True`, **False** - use explicit splitting equations with split fractions. }

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - `False`. **Valid values:** { **True** - include heat transfer terms, **False** - exclude heat transfer terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - `True`. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations, **default** - `useDefault`. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a `PropertyParameterBlock` object. }

property_package_args A `ConfigBlock` with arguments to be passed to a property block(s) and used when constructing these, **default** - `None`. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) - `ProcessBlockData` config for individual elements. Keys are `BlockData` indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) - Function to

take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Flash) New instance

FlashData Class

class `idaes.generic_models.unit_models.flash.FlashData` (*component*)
Standard Flash Unit Model Class

build()
Begin building model (pre-DAE transformation).

Parameters None –

Returns None

Gibbs Reactor

The IDAES Gibbs reactor model represents a unit operation where a material stream undergoes some set of reactions such that the Gibbs energy of the resulting mixture is minimized. Gibbs reactors rely on conservation of individual elements within the system, and thus require element balances, and make use of Lagrange multipliers to find the minimum Gibbs energy state of the system.

Configuration Arguments

The Gibbs Reactor unit model allows users to specify a list of components which should be considered to be inerts within the reactor. This is done using the “inert_species” configuration argument, which should be a list of valid component names. These components will be considered inert, such that flows in and out of the unit for those components in each phase are equal.

Degrees of Freedom

Gibbs reactors generally have between 0 and 2 degrees of freedom, depending on construction arguments.

Typical fixed variables are:

- reactor heat duty (has_heat_transfer = True only).
- reactor pressure change (has_pressure_change = True only).

Model Structure

The core Gibbs reactor unit model consists of a single ControlVolume0DBlock (named control_volume) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Variables

Gibbs reactor units add the following additional Variables beyond those created by the Control Volume Block.

Variable Name	Symbol	Notes
lagrange_mult	$L_{t,e}$	Lagrange multipliers
heat_duty	Q_t	Only if has_heat_transfer = True, reference
deltaP	ΔP_t	Only if has_pressure_change = True, reference

Parameters

The Gibbs reactor unit model includes a scaling parameter for the Gibbs energy minimization constraint, which is named “gibbs_scaling”. The default value is 1 and users may adjust the value of this parameter is required.

Constraints

Gibbs reactor models write the following additional constraints to calculate the state that corresponds to the minimum Gibbs energy of the system.

gibbs_minimization(time, phase, component):

$$0 = \times g_{\text{partial},t,j} + \times \sum_e (L_{t,e} \times \alpha_{j,e})$$

where $g_{\text{partial},t,j}$ is the partial molar Gibbs energy of component j at time t , $L_{t,e}$ is the Lagrange multiplier for element e at time t and $\alpha_{j,e}$ is the number of moles of element e in one mole of component j . $g_{\text{partial},t,j}$ and $\alpha_{j,e}$ come from the outlet StateBlock. t , eps is the `gibbs_scaling` parameter. In cases where inert species are present, these are excluded from the \sum_e term.

In cases where inerts are present, the following additional constraint is written for each inert component and phase:

$$0 = F_{\text{in},p,j} - F_{\text{out},p,j}$$

GibbsReactor Class

```
class idaes.generic_models.unit_models.gibbs_reactor.GibbsReactor(*args,
                                                                **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Gibbs reactors do not support dynamic models, thus this must be False.

has_holdup Gibbs reactors do not have defined volume, thus this must be False.

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { **`EnergyBalanceType.useDefault`** - refer to property package for default balance type, **`**EnergyBalanceType.none`** - exclude energy balances, **`EnergyBalanceType.enthalpyTotal`** - single enthalpy balance for material, **`EnergyBalanceType.enthalpyPhase`** - enthalpy balances for each phase, **`EnergyBalanceType.energyTotal`** - single energy balance for material, **`EnergyBalanceType.energyPhase`** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **`MomentumBalanceType.none`** - exclude momentum balances, **`MomentumBalanceType.pressureTotal`** - single pressure balance for material, **`MomentumBalanceType.pressurePhase`** - pressure balances for each phase, **`MomentumBalanceType.momentumTotal`** - single momentum balance for material, **`MomentumBalanceType.momentumPhase`** - momentum balances for each phase. }

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - `False`. **Valid values:** { **`True`** - include heat transfer terms, **`False`** - exclude heat transfer terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - `False`. **Valid values:** { **`True`** - include pressure change terms, **`False`** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations, **default** - `useDefault`. **Valid values:** { **`useDefault`** - use default package from parent model or flow-sheet, **`PropertyParameterObject`** - a `PropertyParameterBlock` object. }

property_package_args A `ConfigBlock`

with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

inert_species List of species which do not take part in reactions.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (GibbsReactor) New instance

GibbsReactorData Class

class `idaes.generic_models.unit_models.gibbs_reactor.GibbsReactorData` (*component*)
Standard Gibbs Reactor Unit Model
Class

This model assume all possible reactions reach equilibrium such that the system partial molar Gibbs free energy is minimized. Since some species mole flow rate might be very small, the natural log of the species molar flow rate is used. Instead of specifying the system Gibbs free energy as an objective function, the equations for zero partial derivatives of the grand function with Lagrangian multiple terms with respect to product species mole flow rates and the multiples are specified as constraints.

build()
Begin building model (pre-DAE transformation).

Parameters None –

Returns None

Heater

The Heater model is a simple 0D model that adds or removes heat from a material stream.

Example

```
import pyomo.environ
↳as pe # Pyomo environment
from idaes.core import
↳FlowsheetBlock, StateBlock
from idaes.generic_models.
↳unit_models import Heater
from idaes.generic_models.
↳properties import iapws95

# Create an
↳empty flowsheet and steam
↳property parameter block.
model = pe.ConcreteModel()
model.fs
↳= FlowsheetBlock(default=
↳{"dynamic": False})
model.
↳fs.properties = iapws95.
↳Iapws95ParameterBlock()

# Add a Heater
↳model to the flowsheet.
model.
↳fs.heater = Heater(default=
↳{"property_package
↳": model.fs.properties})

# Setup
↳the heater model by fixing
↳the inputs and heat duty
model.fs.heater.
↳inlet[:].enth_mol.fix(4000)
model.fs.heater.
↳inlet[:].flow_mol.fix(100)
model.fs.heater.inlet[:].
↳pressure.fix(101325)
model.fs.heater.
↳heat_duty[:].fix(100*20000)

# Initialize the model.
model.fs.heater.initialize()
```

Degrees of Freedom

Aside from the inlet conditions, a heater model usually has one degree of freedom, which is the heat duty.

Model Structure

A heater model contains one `ControlVolume0DBlock` block.

Variables

The `heat_duty` variable is a reference to `control_volume.heat`.

Constraints

A heater model contains no additional constraints beyond what are contained in a `ControlVolume0DBlock` model.

Heater Class

```
class idaes.generic_models.unit_models.heater.Heater(*args, **kwargs)
    Simple 0D heater/cooler model.
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = `useDefault`. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if `dynamic = True`,

default - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.useDefault`. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum

balances for each phase.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object.}

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Heater) New instance

HeaterData Class

class `idaes.generic_models.unit_models.heater.HeaterData` (*component*)

Simple 0D heater unit. Unit model to add or remove heat from a material.

build()

Building model

Parameters None –

Returns None

HeatExchanger (0D)

The HeatExchanger model can be imported from `idaes.generic_models.unit_models`, while additional rules and utility functions can be imported from `idaes.generic_models.unit_models.heat_exchanger`.

Example

The example below demonstrates how to initialize the HeatExchanger model, and override the default temperature difference calculation.

```
import pyomo.environ
↳ as pe # Pyomo environment
from idaes.core import
↳ FlowsheetBlock, StateBlock
from
↳ idaes.generic_models.unit_
↳ models import HeatExchanger
from idaes.generic_
↳ models.unit_models.heat_
↳ exchanger import delta_
↳ temperature_amtd_callback
from idaes.generic_models.
↳ properties import iapws95

# Create an
↳ empty flowsheet and steam_
↳ property parameter block.
model = pe.ConcreteModel()
model.fs
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
model.
↳ fs.properties = iapws95.
↳ Iapws95ParameterBlock()
```

(continues on next page)

(continued from previous page)

```

# Add a Heater_
↳model to the flowsheet.
model.fs.heat_exchanger_
↳= HeatExchanger(default={
    "delta_temperature_"
↳callback":delta_
↳temperature_amtd_callback,

↳
↳ "shell":{"property_package
↳": model.fs.properties},

↳
↳ "tube":{"property_package
↳": model.fs.properties}))

model.fs.heat_
↳exchanger.area.fix(1000)
model.fs.heat_exchanger.
↳overall_heat_transfer_
↳coefficient[0].fix(100)
model.
↳fs.heat_exchanger.shell_
↳inlet.flow_mol.fix(100)
model.
↳fs.heat_exchanger.shell_
↳inlet.pressure.fix(101325)
model.
↳fs.heat_exchanger.shell_
↳inlet.enth_mol.fix(4000)
model.fs.heat_exchanger.tube_
↳inlet.flow_mol.fix(100)
model.fs.heat_exchanger.tube_
↳inlet.pressure.fix(101325)
model.fs.heat_exchanger.tube_
↳inlet.enth_mol.fix(3000)

# Initialize the model
model.fs.
↳heat_exchanger.initialize()

```

Degrees of Freedom

Aside from the inlet conditions, a heat exchanger model usually has two degrees of freedom, which can be fixed for it to be fully specified. Things that are frequently fixed are two of:

- heat transfer area,
- heat transfer coefficient, or
- temperature approach.

The user may also provide constants to calculate the heat transfer coefficient.

Model Structure

The `HeatExchanger` model contains two `ControlVolume0DBlock` blocks. By default the hot side is named `shell` and the cold side is named `tube`. These names are configurable. The sign convention is that duty is positive for heat flowing from the hot side to the cold side. Aside from the sign convention there is no requirement that the hot side be hotter than the cold side.

The control volumes are configured the same as the `ControlVolume0DBlock` in the *Heater model*. The `HeatExchanger` model contains additional constraints that calculate the amount of heat transferred from the hot side to the cold side.

The `HeatExchanger` has two inlet ports and two outlet ports. By default these are `shell_inlet`, `tube_inlet`, `shell_outlet`, and `tube_outlet`. If the user supplies different hot and cold side names the inlet and outlets are named accordingly.

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	t	Heat transferred from hot side to the cold side
area	A	None	Heat transfer area
heat_transfer_coefficient	U	t	Heat transfer coefficient
delta_temperature	ΔT	t	Temperature difference, defaults to LMTD

Note: `delta_temperature` may be either a variable or expression depending on the callback used. If the specified cold side is hotter than the specified hot side this value will be negative.

Constraints

The default constants can be overridden by providing *alternative rules* for the heat transfer equation, temperature difference, and heat transfer coefficient. The section describes the default constraints.

Heat transfer from shell to tube:

$$Q = UA\Delta T$$

Temperature difference is an expression:

$$\Delta T = \frac{\Delta T_1 - \Delta T_2}{\log_e \left(\frac{\Delta T_1}{\Delta T_2} \right)}$$

The heat transfer coefficient is a variable with no associated constraints by default.

Class Documentation

Note: The `hot_side_config` and `cold_side_config` can also be supplied using the name of the hot and cold sides (`shell` and `tube` by default) as in *the example*.

```
class idaes.generic_models.unit_models.heat_exchanger.HeatExchanger(*args,  
                                                                    **kws)
```

Simple 0D heat exchanger model.

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = `useDefault`. **Valid values:** { `useDefault` - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

hot_side_name Hot side name, sets control volume and inlet and outlet names

cold_side_name Cold side name, sets control volume and inlet and outlet names

hot_side_config A config block used to construct the hot side control volume. This config can be given by the hot side name instead of hot_side_config.

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance

should be constructed, **default** - MomentumBalanceType.pressureTotal.

Valid values: { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

cold_side_config A config block used to construct the cold side control volume. This config can be given by the cold side name instead of cold_side_config.

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude

material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = `False`. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - `False`. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change

terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

delta_temperature_callback Callback for for temperature difference calculations

flow_pattern Heat exchanger flow pattern, **default** - HeatExchangerFlowPattern.countercurrent. **Valid values:** { **HeatExchangerFlowPattern.countercurrent** - countercurrent flow, **HeatExchangerFlowPattern.cocurrent** - cocurrent flow, **HeatExchangerFlowPattern.crossflow** - cross flow, factor times countercurrent temperature difference. }

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HeatExchanger) New instance

class `idaes.generic_models.unit_models.heat_exchanger.HeatExchangerData` (*component*)
Simple 0D heat exchange unit. Unit model to transfer heat from one material to another.

build()
Building model

Parameters None –

Returns None

initialize (*state_args_1=None, state_args_2=None, outlvl=0, solver='ipopt', optarg={'tol': 1e-06}, duty=None*)
Heat exchanger initialization method.

Parameters

- **state_args_1** – a dict of arguments to be passed to the property initialization for the hot side (see documentation of the specific property package) (default = {}).
- **state_args_2** – a dict of arguments to be passed to the property initialization for the cold side (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')
- **duty** – an initial guess for the amount of heat transferred. This should be a tuple in the form (value, units), (default = (1000 J/s))

Returns None

Callbacks

A selection of functions for constructing the `delta_temperature` variable or expression are provided in the `idaes.generic_models.unit_models.heat_exchanger` module. The user may also provide their own function. These callbacks should all take one argument (the HeatExchanger block). With the block argument, the function can add any additional variables, constraints, and expressions needed. The only requirement is that either a variable or expression called `delta_temperature` must be added to the block.

Defined Callbacks for the `delta_temperature_callback` Option

These callbacks provide expressions for the temperature difference used in the heat transfer equations.

`idaes.generic_models.unit_models.heat_exchanger.delta_temperature_lmt_d_callback(b)`

This is a callback for a temperature difference expression to calculate ΔT in the heat exchanger model using log-mean temperature difference (LMTD). It can be supplied to “delta_temperature_callback” HeatExchanger configuration option.

`idaes.generic_models.unit_models.heat_exchanger.delta_temperature_amtd_callback(b)`

This is a callback for a temperature difference expression to calculate ΔT in the heat exchanger model using arithmetic-mean temperature difference (AMTD). It can be supplied to “delta_temperature_callback” HeatExchanger configuration option.

`idaes.generic_models.unit_models.heat_exchanger.delta_temperature_underwood_callback(b)`

This is a callback for a temperature difference expression to calculate ΔT in the heat exchanger model using log-mean temperature difference (LMTD) approximation given by Underwood (1970). It can be supplied to “delta_temperature_callback” HeatExchanger configuration option. This uses a cube root function that works with negative numbers returning the real negative root. This should always evaluate successfully.

Heat Exchangers (1D)

Heat Exchanger models represents a unit operation with two material streams which exchange heat. The IDAES 1-D Heat Exchanger model is used for detailed modeling of heat exchanger units with variations in one spatial dimension. For a simpler representation of a heat exchanger unit see Heat Exchanger (0-D).

Degrees of Freedom

1-D Heat Exchangers generally have 7 degrees of freedom.

Typical fixed variables are:

- shell length and diameter,
- tube length and diameter,
- number of tubes,
- heat transfer coefficients (at all spatial points) for both shell and tube sides.

Model Structure

The core 1-D Heat Exchanger Model unit model consists of two ControlVolume1DBlock Blocks (named shell and tube), each with one Inlet Port (named shell_inlet and tube_inlet) and one Outlet Port (named shell_outlet and tube_outlet).

Construction Arguments

1-D Heat Exchanger units have construction arguments specific to the shell side, tube side and for the unit as a whole.

Arguments that are applicable to the heat exchanger unit are as follows:

- flow_type - indicates the flow arrangement within the unit to be modeled. Options are:
 - ‘co-current’ - (default) shell and tube both flow in the same direction (from $x=0$ to $x=1$)
 - ‘counter-current’ - shell and tube flow in opposite directions (shell from $x=0$ to $x=1$ and tube from $x=1$ to $x=0$).
- finite_elements - sets the number of finite elements to use when discretizing the spatial domains (default = 20). This is used for both shell and tube side domains.
- collocation_points - sets the number of collocation points to use when discretizing the spatial domains (default = 5, col-

location methods only). This is used for both shell and tube side domains.

- **has_wall_conduction** - option to enable a model for heat conduction across the tube wall:

- ‘none’ - 0D wall model
- ‘1D’ - 1D heat conduction equation along the thickness of the tube wall
- ‘2D’ - 2D heat conduction equation along the length and thickness of the tube wall

Arguments that are applicable to the shell side:

- **property_package** - property package to use when constructing shell side Property Blocks (default = ‘use_parent_value’). This is provided as a Physical Parameter Block by the Flowsheet when creating the model. If a value is not provided, the ControlVolume Block will try to use the default property package if one is defined.
- **property_package_args** - set of arguments to be passed to the shell side Property Blocks when they are created.
- **transformation_method** - argument to specify the DAE transformation method for the shell side; should be compatible with the Pyomo DAE TransformationFactory
- **transformation_scheme** - argument to specify the scheme to use for the selected DAE transformation method; should be compatible with the Pyomo DAE TransformationFactory

Arguments that are applicable to the tube side:

- **property_package** - property package to use when constructing tube side Property Blocks (default = ‘use_parent_value’). This is provided as a Property Parameter Block by the Flowsheet when creating the model. If a value is not provided, the ControlVolume Block will try to use the default property package if one is defined.
- **property_package_args** - set of arguments to be passed to the tube side Property Blocks when they are created.

- `transformation_method` - argument to specify the DAE transformation method for the tube side; should be compatible with the Pyomo DAE Transformation-Factory
- `transformation_scheme` - argument to specify the scheme to use for the selected DAE transformation method; should be compatible with the Pyomo DAE TransformationFactory

Additionally, 1-D Heat Exchanger units have the following construction arguments which are passed to the `ControlVolume1DBlock` Block for determining which terms to construct in the balance equations for the shell and tube side.

Argument	Default Value
<code>dynamic</code>	<code>useDefault</code>
<code>has_holdup</code>	<code>False</code>
<code>material_balance_type</code>	<code>'componentTotal'</code>
<code>energy_balance_type</code>	<code>'enthalpyTotal'</code>
<code>momentum_balance_type</code>	<code>'pressureTotal'</code>
<code>has_phase_equilibrium</code>	<code>False</code>
<code>has_heat_transfer</code>	<code>True</code>
<code>has_pressure_change</code>	<code>False</code>

Additional Constraints

1-D Heat Exchanger models write the following additional Constraints to describe the heat transfer between the two sides of the heat exchanger. Firstly, the shell- and tube-side heat transfer is calculated as:

$$Q_{shell,t,x} = -N_{tubes} \times (\pi \times U_{shell,t,x} \times D_{tube,outer} \times (T_{shell,t,x} - T_{wall,t,x}))$$

where $Q_{shell,t,x}$ is the shell-side heat duty at point x and time t , N_{tubes} D_{tube} are the number of and diameter of the tubes in the heat exchanger, $U_{shell,t,x}$ is the shell-side heat transfer coefficient, and $T_{shell,t,x}$ and $T_{wall,t,x}$ are the shell-side and tube wall temperatures respectively.

$$Q_{tube,t,x} = N_{tubes} \times (\pi \times U_{tube,t,x} \times D_{tube,inner} \times (T_{wall,t,x} - T_{tube,t,x}))$$

where $Q_{tube,t,x}$ is the tube-side heat duty at point x and time t , $U_{tube,t,x}$

is the tube-side heat transfer coefficient and $T_{tube,t,x}$ is the tube-side temperature.

If a OD wall model is used for the tube wall conduction, the following constraint is implemented to connect the heat terms on the shell and tube side:

$$N_{tubes} \times Q_{tube,t,x} = -Q_{shell,t,x}$$

Finally, the following Constraints are written to describe the unit geometry:

$$4 \times A_{tube} = \pi \times D_{tube}^2$$

$$4 \times A_{shell} = \pi \times (D_{shell}^2 - N_{tubes} \times D_{tube}^2)$$

where A_{shell} and A_{tube} are the shell and tube areas respectively and D_{shell} and D_{tube} are the shell and tube diameters.

Variables

1-D Heat Exchanger units add the following additional Variables beyond those created by the ControlVolume1DBlock Block.

Variable	Name	Notes
L_{shell}	shell_length	Reference to shell.length
A_{shell}	shell_area	Reference to shell.area
D_{shell}	d_shell	
L_{tube}	tube_length	Reference to tube.length
A_{tube}	tube_area	Reference to tube.area
D_{tube}	d_tube	
N_{tubes}	N_tubes	
$T_{wall,t,x}$	temperature_wall	
$U_{shell,t,x}$	shell_heat_transfer_coefficient	
$U_{tube,t,x}$	tube_heat_transfer_coefficient	

HeatExchanger1dClass

```
class idaes.generic_models.unit_models.heat_exchanger_1D.HeatExchanger1D(*args,  
                                                                           **kwds)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”

- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

shell_side shell side config arguments

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBal-

anceType.useDefault. **Valid values:**
{ **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

has_phase_equilibrium Argument to enable phase equilibrium on the shell side. - True - include phase equilibrium term - False - do not include phase equilibrium term

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from

parent (default = None) - a dict (see property package for documentation)

transformation_method Discretization method to use for DAE transformation. See Pyomo documentation for supported transformations.

transformation_scheme Discretization scheme to use when transforming domain. See Pyomo documentation for supported schemes.

tube_side tube side config arguments

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** -

single enthalpy balance for material,
EnergyBalanceType.enthalpyPhase
- enthalpy balances for each phase,
EnergyBalanceType.energyTotal -
single energy balance for material,
EnergyBalanceType.energyPhase -
energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal.
Valid values: { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

has_phase_equilibrium Argument to enable phase equilibrium on the shell side. - True - include phase equilibrium term - False - do not include phase equilibrium term

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

transformation_method Discretization method to use for DAE transformation. See Pyomo documentation for supported transformations.

transformation_scheme Discretization scheme to use when transforming domain. See Pyomo documentation for supported schemes.

finite_elements Number of finite elements to use when discretizing length domain (default=20)

collocation_points Number of collocation points to use per finite element when discretizing length domain (default=3)

flow_type Flow configuration of heat exchanger - HeatExchangerFlowPattern.cocurrent: shell and tube flows from 0 to 1 (default) - HeatExchangerFlowPattern.countercurrent: shell side flows from 0 to 1 tube side flows from 1 to 0

has_wall_conduction Argument to enable type of wall heat conduction model. - WallConductionType.zero_dimensional - 0D wall model (default), - WallConductionType.one_dimensional - 1D wall model along the thickness of the tube, - WallConductionType.two_dimensional - 2D wall model along the length and thickness of the tube

- **initialize** (*dict*) - ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HeatExchanger1D) New instance

HeatExchanger1dDataClass

class `idaes.generic_models.unit_models.heat_exchanger_1D.HeatExchanger1dData` (*component*)
Standard Heat Exchanger 1D Unit
Model Class.

build()
Begin building model (pre-DAE transformation).

Parameters None –

Returns None

initialize (*shell_state_args=None, tube_state_args=None, outlvl=0, solver='ipopt', optarg={'tol': 1e-06}*)
Initialization routine for the unit (default solver ipopt).

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={ 'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

Mixer

The IDAES Mixer unit model represents operations where multiple streams of material are combined into a single flow. The Mixer class can be used to create either a stand-alone mixer unit, or as part of a unit model where multiple streams need to be mixed.

Degrees of Freedom

Mixer units have zero degrees of freedom.

Model Structure

The IDAES Mixer unit model does not use ControlVolumes, and instead writes a set of material, energy and momentum balances to combine the inlet streams into a single mixed stream. Mixer models have a user-defined number of inlet Ports (by default named inlet_1, inlet_2, etc.) and one outlet Port (named outlet).

Mixed State Block

If a mixed state block is provided in the construction arguments, the Mixer model will use this as the StateBlock for the mixed stream in the resulting balance equations. This allows a Mixer unit to be used as part of a larger unit operation by linking multiple inlet streams to a single existing StateBlock.

Variables

Mixer units have the following variables (*i* indicates index by inlet):

Variable Name	Sym- bol	Notes
phase_equilibrium_generation	$X_{eq,t,r}$	Only if has_phase_equilibrium = True, Generation term for phase equilibrium
minimum_pressure	$P_{min,t,i}$	Only if momentum_mixing_type = MomentumMixingType.minimize

Parameters

Mixer units have the following parameters:

Variable Name	Sym- bol	Notes
eps_pressure	ϵ	Only if momentum_mixing_type = MomentumMixingType.minimize, smooth minimum parameter

Constraints

The constraints written by the Mixer model depend upon the construction arguments chosen.

If *material_mixing_type* is *extensive*:

- If *material_balance_type* is *component-Phase*:

material_mixing_equations(*t*, *p*, *j*):

$$0 = \sum_i F_{in,i,p,j} - F_{out,p,j} + \sum_r n_{r,p,j} \times X_{eq,t,r}$$

- If *material_balance_type* is *component-Total*:

material_mixing_equations(*t*, *j*):

$$0 = \sum_p \left(\sum_i F_{in,i,p,j} - F_{out,p,j} + \sum_r n_{r,p,j} \times X_{eq,t,r} \right)$$

- If *material_balance_type* is *total*:

material_mixing_equations(*t*):

$$0 = \sum_p \sum_j \left(\sum_i F_{in,i,p,j} - F_{out,p,j} + \sum_r n_{r,p,j} \times X_{eq,t,r} \right)$$

where $n_{r,p,j}$ is the stoichiometric coefficient of component *j* in phase *p* in reaction *r*.

If 'energy_mixing_type' is *extensive*:

enthalpy_mixing_equations(*t*):

$$0 = \sum_i \sum_p H_{in,i,p} - \sum_p H_{out,p}$$

If 'momentum_mixing_type' is *minimize*, a series of smooth minimum operations are performed:

minimum_pressure_constraint(*t*, *i*):

For the first inlet:

$$P_{min,t,i} = P_{t,i}$$

Otherwise:

$$P_{min,t,i} = \text{smmin}(P_{min,t,i-1}, P_{t,i}, \text{eps})$$

Here, $P_{t,i}$ is the pressure in inlet i at time t , $P_{min,t,i}$ is the minimum pressure in all inlets up to inlet i , and $smmin$ is the smooth minimum operator (see IDAES Utility Function documentation).

The minimum pressure in all inlets is then:

mixture_pressure(t):

$$P_{mix,t} = P_{min,t,i=last}$$

If *momentum_mixing_type* is *equality*, the pressure in all inlets and the outlet are equated.

Note: This may result in an over-specified problem if the user is not careful.

pressure_equality_constraints(t, i):

$$P_{mix,t} = P_{t,i}$$

Often the minimum inlet pressure constraint is useful for sequential modular type initialization, but the equal pressure constants are required for pressure-driven flow models. In these cases it may be convenient to use the minimum pressure constraint for some initialization steps, then deactivate it and use the equal pressure constraints. The *momentum_mixing_type* is *minimum_and_equality* this will create the constraints for both with the minimum pressure constraint being active.

The *mixture_pressure(t)* and *pressure_equality_constraints(t, i)* can be directly activated and deactivated, but only one set of constraints should be active at a time.

The `use_minimum_inlet_pressure_constraint()`

and `use_equal_pressure_constraint()`
methods are also provided to switch
between constant sets.

Mixer Class

```
class idaes.generic_models.unit_models.mixer.Mixer(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = False. Mixer blocks are always steady-state.

has_holdup Mixer blocks do not contain holdup, thus this must be False.

property_package Property parameter object used to define property calculations, **default** - `useDefault`. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a `PropertyParameterBlock` object. }

property_package_args A `ConfigBlock` with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

inlet_list A list containing names of inlets, **default** - None. **Valid values:** { **None** - use `num_inlets` argument, **list** - a list of names to use for inlets. }

num_inlets Argument indicating number (int) of inlets to construct, not used if `inlet_list` arg is provided, **default** - None. **Valid values:** { **None** - use `inlet_list` arg instead, or default to 2 if neither argument provided, **int** - number of inlets }

to create (will be named with sequential integers from 1 to num_inlets).}

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.useDefault`. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance.}

has_phase_equilibrium Argument indicating whether phase equilibrium should be calculated for the resulting mixed stream, **default** - `False`. **Valid values:** { **True** - calculate phase equilibrium in mixed stream, **False** - do not calculate equilibrium in mixed stream.}

energy_mixing_type Argument indicating what method to use when mixing energy flows of incoming streams, **default** - `MixingType.extensive`. **Valid values:** { **MixingType.none** - do not include energy mixing equations, **MixingType.extensive** - mix total enthalpy flows of each phase.}

momentum_mixing_type Argument indicating what method to use when mixing momentum/ pressure of incoming streams, **default** - `MomentumMixingType.minimize`. **Valid values:** { **MomentumMixingType.none** - do not include momentum mixing equations, **MomentumMixingType.minimize** - mixed stream has pressure equal to the minimum pressure of the incoming streams (uses `smoothMin` operator), **MomentumMixingType.equality** - enforces equality of pressure in mixed and all incoming streams., **MomentumMixingType.minimize_and_equality** - add constraints for pressure equal to the minimum pressure of the inlets and constraints for equality of pressure in

mixed and all incoming streams. When the model is initially built, the equality constraints are deactivated. This option is useful for switching between flow and pressure driven simulations.}

mixed_state_block An existing state block to use as the outlet stream from the Mixer block, **default** - None. **Valid values:** { **None** - create a new StateBlock for the mixed stream, **StateBlock** - a StateBlock to use as the destination for the mixed stream.}

construct_ports Argument indicating whether model should construct Port objects linked to all inlet states and the mixed state, **default** - True. **Valid values:** { **True** - construct Ports for all states, **False** - do not construct Ports.

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Mixer) New instance

MixerData Class

class `idaes.generic_models.unit_models.mixer.MixerData` (*component*)

This is a general purpose model for a Mixer block with the IDAES modeling framework. This block can be used either as a stand-alone Mixer unit operation, or as a sub-model within another unit operation.

This model creates a number of StateBlocks to represent the incoming streams, then writes a set of phase-component material balances, an overall enthalpy balance and a momentum balance (2 options) linked to a mixed-state StateBlock. The mixed-state StateBlock can either be speci-

fied by the user (allowing use as a sub-model), or created by the Mixer.

When being used as a sub-model, Mixer should only be used when a set of new StateBlocks are required for the streams to be mixed. It should not be used to mix streams from multiple ControlVolumes in a single unit model - in these cases the unit model developer should write their own mixing equations.

add_energy_mixing_equations (*inlet_blocks, mixed_block*)

Add energy mixing equations (total enthalpy balance).

add_inlet_state_blocks (*inlet_list*)

Construct StateBlocks for all inlet streams.

Parameters of strings to use as StateBlock names (*list*) –

Returns list of StateBlocks

add_material_mixing_equations (*inlet_blocks, mixed_block, mb_type*)

Add material mixing equations.

add_mixed_state_block ()

Constructs StateBlock to represent mixed stream.

Returns New StateBlock object

add_port_objects (*inlet_list, inlet_blocks, mixed_block*)

Adds Port objects if required.

Parameters

- **list of inlet StateBlock objects** (*a*) –
- **mixed state StateBlock object** (*a*) –

Returns None

add_pressure_equality_equations (*inlet_blocks, mixed_block*)

Add pressure equality equations. Note that this writes a number of constraints equal to the number of inlets, enforcing equality between all inlets and the mixed stream.

add_pressure_minimization_equations (*inlet_blocks, mixed_block*)

Add pressure minimization equations. This is done by sequential comparisons of each inlet to the minimum pressure so far, using the IDAES smooth minimum function.

build()

General build method for MixerData. This method calls a number of sub-methods which automate the construction of expected attributes of unit models.

Inheriting models should call *super().build*.

Parameters None –

Returns None

create_inlet_list()

Create list of inlet stream names based on config arguments.

Returns list of strings

get_mixed_state_block()

Validates StateBlock provided in user arguments for mixed stream.

Returns The user-provided StateBlock or an Exception

initialize(outlvl=6, optarg={}, solver='ipopt', hold_state=False)

Initialization routine for mixer (default solver ipopt)

Keyword Arguments

- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={ })
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')
- **hold_state** – flag indicating whether the initialization routine should unfix any state variables fixed during initialization, **default** - False. **Valid values:** **True** - states variables are not unfixed, and a dict of returned containing flags for which states were fixed during initialization, **False** - state variables are unfixed after initialization by calling the release_state method.

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()

This method executes the model_check methods on the associated state blocks

(if they exist). This method is generally called by a unit model as part of the unit's `model_check` method.

Parameters None –

Returns None

release_state (*flags, outlvl=0*)

Method to release state variables fixed during initialization.

Keyword Arguments

- **flags** – dict containing information of which state variables were fixed during initialization, and should now be unfixed. This dict is returned by `initialize` if `hold_state = True`.
- **outlvl** – sets output level of logging

Returns None

use_equal_pressure_constraint ()

Deactivate the mixer pressure = minimum inlet pressure constraint and activate the mixer pressure and all inlet pressures are equal constraints. This should only be used when `momentum_mixing_type == MomentumMixingType.minimize_and_equality`.

use_minimum_inlet_pressure_constraint ()

Activate the mixer pressure = minimum inlet pressure constraint and deactivate the mixer pressure and all inlet pressures are equal constraints. This should only be used when `momentum_mixing_type == MomentumMixingType.minimize_and_equality`.

Plug Flow Reactor

The IDAES Plug Flow Reactor (PFR) model represents a unit operation where a material stream passes through a linear reactor vessel whilst undergoing some chemical reaction(s). This model requires modeling the system in one spatial dimension.

Degrees of Freedom

PFRs generally have at least 2 degrees of freedom.

Typical fixed variables are:

- 2 of reactor length, area and volume.

Model Structure

The core PFR unit model consists of a single `ControlVolume1DBlock` (named `control_volume`) with one `Inlet Port` (named `inlet`) and one `Outlet Port` (named `outlet`).

Variables

PFR units add the following additional Variables:

Variable	Name	Notes
L	length	Reference to <code>control_volume.length</code>
A	area	Reference to <code>control_volume.area</code>
V	volume	Reference to <code>control_volume.volume</code>
$Q_{t,x}$	heat	Only if <code>has_heat_transfer = True</code> , reference to <code>holdup.heat</code>
$\Delta P_{t,x}$	deltaP	Only if <code>has_pressure_change = True</code> , reference to <code>holdup.deltaP</code>

Constraints

PFR units write the following additional Constraints at all points in the spatial domain:

$$X_{t,x,r} = A \times r_{t,x,r}$$

where $X_{t,x,r}$ is the extent of reaction of reaction r at point x and time t , A is the cross-sectional area of the reactor and $r_{t,r}$ is the volumetric rate of reaction of reaction r at point x and time t (from the outlet `StateBlock`).

PFR Class

```
class idaes.generic_models.unit_models.plug_flow_reactor.PFR(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **EnergyBalanceType.useDefault** -

refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms.}

has_heat_of_reaction Indicates whether terms for heat of reaction terms should be constructed, **default** - False. **Valid values:** { **True** - include heat of reaction terms, **False** - exclude heat of reaction terms.}

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - False. **Valid values:** { **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object.}

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object.}

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

length_domain_set A list of values to be used when constructing the length domain of the reactor. Point must lie between 0.0 and 1.0, **default** - [0.0, 1.0]. **Valid values:** { a list of floats}

transformation_method Method to use to transform domain. Must be a method recognised by the Pyomo TransformationFactory, **default** - "dae.finite_difference".

transformation_scheme Scheme to use when transforming domain. See Pyomo documentation for supported schemes, **default** - "BACKWARD".

finite_elements Number of finite elements to use when transforming length domain, **default** - 20.

collocation_points Number of collocation points

tion points to use when transforming length domain, **default** - 3.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (PFR) New instance

PFRData Class

```
class idaes.generic_models.unit_models.plug_flow_reactor.PFRData (component)  
    Standard Plug Flow Reactor Unit  
    Model Class
```

```
build()  
    Begin building model (pre-DAE trans-  
    formation).
```

Parameters None –

Returns None

Pressure Changer

The IDAES Pressure Changer model represents a unit operation with a single stream of material which undergoes a change in pressure due to the application of a work. The Pressure Changer model contains support for a number of different thermodynamic assumptions regarding the working fluid.

Degrees of Freedom

Pressure Changer units generally have one or more degrees of freedom, depending on the thermodynamic assumption used.

Typical fixed variables are:

- outlet pressure, P_{ratio} or ΔP ,

- unit efficiency (isentropic or pump assumption).

Model Structure

The core Pressure Changer unit model consists of a single `ControlVolume0D` (named `control_volume`) with one Inlet Port (named `inlet`) and one Outlet Port (named `outlet`). Additionally, if an isentropic pressure changer is used, the unit model contains an additional `StateBlock` named `properties_isentropic` at the unit model level.

Variables

Pressure Changers contain the following Variables (not including those contained within the control volume Block):

Variable	Name	Notes
P_{ratio}	ratioP	
V_t	volume	Only if <code>has_rate_reactions = True</code> , reference to <code>control_volume.rate_reaction_extent</code>
$W_{mechanical,t}$	work_mechanical	Reference to <code>control_volume.work</code>
$W_{fluid,t}$	work_fluid	Pump assumption only
$\eta_{pump,t}$	efficiency_pump	Pump assumption only
$W_{isentropic,t}$	work_isentropic	Isentropic assumption only
$\eta_{isentropic,t}$	efficiency_isentropic	Isentropic assumption only

Isentropic Pressure Changers also have an additional Property Block named `properties_isentropic` (attached to the Unit Model).

Constraints

In addition to the Constraints written by the Control Volume block, Pressure Changer writes additional Constraints which depend on the thermodynamic assumption chosen. All Pressure Changers add the following Constraint to calculate the pressure ratio:

$$P_{ratio,t} \times P_{in,t} = P_{out,t}$$

Isothermal Assumption

The isothermal assumption writes one additional Constraint:

$$T_{out} = T_{in}$$

Adiabatic Assumption

The isothermal assumption writes one additional Constraint:

$$H_{out} = H_{in}$$

Isentropic Assumption

The isentropic assumption creates an additional set of Property Blocks (indexed by time) for the isentropic fluid calculations (named `properties_isentropic`). This requires a set of balance equations relating the inlet state to the isentropic conditions, which are shown below:

$$F_{in,t,p,j} = F_{isentropic,t,p,j}$$

$$s_{in,t} = s_{isentropic,t}$$

$$P_{in,t} \times P_{ratio,t} = P_{isentropic,t}$$

where $F_{t,p,j}$ is the flow of component j in phase p at time t and s is the specific entropy of the fluid at time t .

Next, the isentropic work is calculated as follows:

$$W_{isentropic,t} = \sum_p H_{isentropic,t,p} - \sum_p H_{in,t,p}$$

where $H_{t,p}$ is the total energy flow of phase p at time t . Finally, a constraint which relates the fluid work to the actual mechanical work via an efficiency term η .

If compressor is True, $W_{isentropic,t} = W_{mechanical,t} \times \eta_t$

If compressor is False, $W_{isentropic,t} \times \eta_t = W_{mechanical,t}$

Pump (Incompressible Fluid) Assumption

The incompressible fluid assumption writes two additional constraints. Firstly, a Constraint is written which relates fluid work to the pressure change of the fluid.

$$W_{fluid,t} = (P_{out,t} - P_{in,t}) \times F_{vol,t}$$

where $F_{vol,t}$ is the total volumetric flowrate of material at time t (from the outlet Property Block). Secondly, a constraint which relates the fluid work to the actual mechanical work via an efficiency term η .

If compressor is True, $W_{fluid,t} = W_{mechanical,t} \times \eta_t$

If compressor is False, $W_{fluid,t} \times \eta_t = W_{mechanical,t}$

PressureChanger Class

```
class idaes.generic_models.unit_models.pressure_changer.PressureChanger(*args,
**kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.useDefault`. **Valid values:** { `MaterialBalanceType.useDefault` - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { `EnergyBalanceType.useDefault` - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { `MomentumBalanceType.none` - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = `False`.

Valid values: { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

compressor Indicates whether this unit should be considered a compressor (True (default), pressure increase) or an expander (False, pressure decrease).

thermodynamic_assumption Flag to set the thermodynamic assumption to use for the unit. - ThermodynamicAssumption.isothermal (default) - ThermodynamicAssumption.isentropic - ThermodynamicAssumption.pump - ThermodynamicAssumption.adiabatic

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (PressureChanger) New instance

PressureChangerData Class

class `idaes.generic_models.unit_models.pressure_changer.PressureChangerData` (*component*)
Standard Compressor/Expander Unit
Model Class

add_adiabatic()
Add constraints for adiabatic assumption.

Parameters None –

Returns None

add_isentropic()
Add constraints for isentropic assumption.

Parameters None –

Returns None

add_isothermal()
Add constraints for isothermal assumption.

Parameters None –

Returns None

add_pump()
Add constraints for the incompressible fluid assumption

Parameters None –

Returns None

build()

Parameters None –

Returns None

init_isentropic (*state_args, outlvl, solver, optarg*)
Initialization routine for unit (default solver ipopt)

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={ 'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

initialize (*state_args=None, routine=None, outlvl=0, solver='ipopt', optarg={'tol': 1e-06}*)

General wrapper for pressure changer initialization routines

Keyword Arguments

- **routine** – str stating which initialization routine to execute * None - use routine matching thermodynamic_assumption * 'isentropic' - use isentropic initialization routine * 'isothermal' - use isothermal initialization routine
- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

model_check ()

Check that pressure change matches with compressor argument (i.e. if compressor = True, pressure should increase or work should be positive)

Parameters None –

Returns None

Product Block

Product Blocks are used to represent sinks of material in Flowsheets. These can be used as a convenient way to mark the final destination of a material stream and to view the state of that material.

Degrees of Freedom

Product blocks generally have zero degrees of freedom.

Model Structure

Product Blocks consists of a single StateBlock (named properties), each with one Inlet Port (named inlet). Product Blocks also contain References to the state variables defined within the StateBlock

Additional Constraints

Product Blocks write no additional constraints to the model.

Variables

Product blocks add no additional Variables.

Product Class

```
class idaes.generic_models.unit_models.product.Product(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = False. Product blocks are always steady- state.

has_holdup Product blocks do not contain holdup, thus this must be False.

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Product) New instance

ProductData Class

class `idaes.generic_models.unit_models.product.ProductData` (*component*)
Standard Product Block Class

build()
Begin building model.

Parameters None –

Returns None

initialize (*state_args={}*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)
This method calls the initialization method of the state block.

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine

- **optarg** – solver options dictionary object (default={ ‘tol’: 1e-6})
- **solver** – str indicating which solver to use during initialization (default = ‘ipopt’)

Returns None

Pump

The Pump model is a *PressureChanger*, where the configuration is set so that the “compressor” option can only be True, and the default “thermodynamic_assumption” is “pump.” See the *PressureChanger documentation* for details.

Example

The example below demonstrates the basic Pump model usage:

```
import pyomo.environ as pyo
from idaes.
↳core import FlowsheetBlock
from idaes.generic_models.
↳unit_models import Pump
from idaes.generic_models.
↳properties import iapws95

m = pyo.ConcreteModel()
m.fs_
↳= FlowsheetBlock(default=
↳{"dynamic": False})
m.fs.properties = iapws95.
↳Iapws95ParameterBlock()
m.fs.unit_
↳= Pump(default={"property_
↳package": m.fs.properties})

m.fs.unit.
↳inlet.flow_mol[0].fix(100)
m.fs.unit.
↳inlet.enth_mol[0].fix(2000)
m.fs.unit.inlet.
↳pressure[0].fix(101325)

m.fs.unit.deltaP.fix(100000)
m.fs.unit.
↳efficiency_pump.fix(0.8)
```

Separator

The IDAES Separator unit model represents operations where a single stream is split into multiple flows. The Separator model supports separation using split fractions, or by ideal separation of flows. The Separator class can be used to create either a stand-alone separator unit, or as part of a unit model where a flow needs to be separated.

Degrees of Freedom

Separator units have a number of degrees of freedom based on the separation type chosen.

- If *split_basis* = 'phaseFlow', degrees of freedom are generally $(no.outlets - 1) \times no.phases$
- If *split_basis* = 'componentFlow', degrees of freedom are generally $(no.outlets - 1) \times no.components$
- If *split_basis* = 'phaseComponentFlow', degrees of freedom are generally $(no.outlets - 1) \times no.phases \times no.components$
- If *split_basis* = 'totalFlow', degrees of freedom are generally $(no.outlets - 1) \times no.phases \times no.components$

Typical fixed variables are:

- split fractions.

Model Structure

The IDAES Separator unit model does not use ControlVolumes, and instead writes a set of material, energy and momentum balances to split the inlet stream into a number of outlet streams. Separator models have a single inlet Port (named inlet) and a user-defined number of outlet Ports (by default named outlet_1, outlet_2, etc.).

Mixed State Block

If a mixed state block is provided in the construction arguments, the Mixer model will use this as the StateBlock for

the mixed stream in the resulting balance equations. This allows a Mixer unit to be used as part of a larger unit operation by linking to an existing StateBlock.

Ideal Separation

The IDAES Separator model supports ideal separations, where all of a given subset of the mixed stream is sent to a single outlet (i.e. split fractions are equal to zero or one). In these cases, no Constraints are necessary for performing the separation, as the mixed stream states can be directly partitioned to the outlets.

Ideal separations will not work for all choices of state variables, and thus will not work for all property packages. To use ideal separations, the user must provide a map of what part of the mixed flow should be partitioned to each outlet. The *ideal_split_map* should be a dict-like object with keys as tuples matching the *split_basis* argument and values indicating which outlet this subset should be partitioned to.

Variables

Separator units have the following variables (*o* indicates index by outlet):

Variable Name	Symbol	Notes
split_fraction	$\phi_{t,o,*}$	Indexing sets depend upon <i>split_basis</i>

Constraints

Separator units have the following Constraints, unless *ideal_separation* is True.

- If *material_balance_type* is *component-Phase*:

material_splitting_eqn(*t*, *o*, *p*, *j*):

$$F_{in,t,p,j} = \phi_{t,p,*} \times F_{t,o,p,j}$$

- If *material_balance_type* is *component-Total*:

material_splitting_eqn(t, o, j):

$$\sum_p F_{in,t,p,j} = \sum_p \phi_{t,p,*} \times F_{t,o,p,j}$$

- If *material_balance_type* is *total*:

material_splitting_eqn(t, o):

$$\sum_p \sum_j F_{in,t,p,j} = \sum_p \sum_j \phi_{t,p,*} \times F_{t,o,p,j}$$

If *energy_split_basis* is *equal_temperature*:

temperature_equality_eqn(t, o):

$$T_{in,t} = T_{t,o}$$

If *energy_split_basis* is *equal_molar_enthalpy*:

molar_enthalpy_equality_eqn(t, o):

$$h_{in,t} = h_{t,o}$$

If *energy_split_basis* is *enthalpy_split*:

molar_enthalpy_splitting_eqn(t, o):

$$\sum_p h_{in,t,p} * sf_{t,o,p} = \sum_p h_{t,o,p}$$

pressure_equality_eqn(t, o):

$$P_{in,t} = P_{t,o}$$

Separator Class

```
class idaes.generic_models.unit_models.separator.Separator(*args, **kwds)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = False. Product blocks are always steady- state.

has_holdup Product blocks do not contain holdup, thus this must be False.

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

outlet_list A list containing names of outlets, **default** - None. **Valid values:** { **None** - use num_outlets argument, **list** - a list of names to use for outlets. }

num_outlets Argument indicating number (int) of outlets to construct, not used if outlet_list arg is provided, **default** - None. **Valid values:** { **None** - use outlet_list arg instead, or default to 2 if neither argument provided, **int** - number of outlets to create (will be named with sequential integers from 1 to num_outlets). }

split_basis Argument indicating basis to use for splitting mixed stream, **default** - SplittingType.totalFlow. **Valid values:** { **SplittingType.totalFlow** - split based on total flow (split fraction indexed only by time and outlet), **SplittingType.phaseFlow** - split based on phase flows (split fraction indexed by time, outlet and phase), **SplittingType.componentFlow** - split based on component flows (split fraction indexed by time, outlet and components), **SplittingType.phaseComponentFlow** - split based on phase-component flows (split fraction indexed by both time, outlet, phase and components). }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****Mate-**

MaterialBalanceType.none - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance.}

has_phase_equilibrium Argument indicating whether phase equilibrium should be calculated for the resulting mixed stream, **default** - False. **Valid values:** { **True** - calculate phase equilibrium in mixed stream, **False** - do not calculate equilibrium in mixed stream. }

energy_split_basis Argument indicating basis to use for splitting energy this is not used for when `ideal_separation == True`. **default** - `EnergySplittingType.equal_temperature`. **Valid values:** { **EnergySplittingType.equal_temperature** - outlet temperatures equal inlet, **EnergySplittingType.equal_molar_enthalpy** - outlet molar enthalpies equal inlet, **EnergySplittingType.enthalpy_split** - apply split fractions to enthalpy flows. Does not work with component or phase-component splitting. }

ideal_separation Argument indicating whether ideal splitting should be used. Ideal splitting assumes perfect separation of material, and attempts to avoid duplication of StateBlocks by directly partitioning outlet flows to ports, **default** - False. **Valid values:** { **True** - use ideal splitting methods. Cannot be combined with `has_phase_equilibrium = True`, **False** - use explicit splitting equations with split fractions. }

ideal_split_map Dictionary containing information on how extensive variables should be partitioned when using ideal splitting (`ideal_separation = True`). **default** - None. **Valid values:** { **dict** with keys of indexing set members and values indicating which outlet this combination of keys should be partitioned to. E.g. {("Vap", "H2"): "outlet_1"} }

mixed_state_block An existing state block to use as the source stream from the Separator block, **default** - None. **Valid values:** { **None** - create a new StateBlock for the mixed stream, **StateBlock** - a StateBlock to use as the source for the mixed stream. }

construct_ports Argument indicating whether model should construct Port objects linked the mixed state and all outlet states, **default** - True. **Valid values:** { **True** - construct Ports for all states, **False** - do not construct Ports. }

- **initialize** (*dict*) - Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Separator) New instance

SeparatorData Class

class `idaes.generic_models.unit_models.separator.SeparatorData` (*component*)

This is a general purpose model for a Separator block with the IDAES modeling framework. This block can be used either as a stand-alone Separator unit operation, or as a sub-model within another unit operation.

This model creates a number of StateBlocks to represent the outgoing streams, then writes a set of phase-component material balances, an overall enthalpy balance (2 options), and a momentum balance (2 options) linked to a mixed-state StateBlock. The mixed-state StateBlock can either be specified by the user (allowing use as a sub-model), or created by the Separator.

When being used as a sub-model, Separator should only be used when a set of new StateBlocks are required for the streams to be separated. It should not

be used to separate streams to go to multiple ControlVolumes in a single unit model - in these cases the unit model developer should write their own splitting equations.

add_energy_splitting_constraints (*mixed_block*)

Creates constraints for splitting the energy flows - done by equating temperatures in outlets.

add_inlet_port_objects (*mixed_block*)

Adds inlet Port object if required.

Parameters *mixed state*

StateBlock object (*a*) -

Returns None

add_material_splitting_constraints (*mixed_block*)

Creates constraints for splitting the material flows

add_mixed_state_block ()

Constructs StateBlock to represent mixed stream.

Returns New StateBlock object

add_momentum_splitting_constraints (*mixed_block*)

Creates constraints for splitting the momentum flows - done by equating pressures in outlets.

add_outlet_port_objects (*outlet_list, outlet_blocks*)

Adds outlet Port objects if required.

Parameters *list of outlet*

StateBlock objects (*a*) -

Returns None

add_outlet_state_blocks (*outlet_list*)

Construct StateBlocks for all outlet streams.

Parameters *of strings to use as*

StateBlock names (*list*) -

Returns list of StateBlocks

add_split_fractions (*outlet_list, mixed_block*)

Creates outlet Port objects and tries to partition mixed stream flows between these

Parameters

- **representing the mixed flow to be split**
(StateBlock) -

- **list of names for outlets**
(*a*) –

Returns None

build()

General build method for Separator-Data. This method calls a number of sub-methods which automate the construction of expected attributes of unit models.

Inheriting models should call *super().build*.

Parameters None –

Returns None

create_outlet_list()

Create list of outlet stream names based on config arguments.

Returns list of strings

get_mixed_state_block()

Validates StateBlock provided in user arguments for mixed stream.

Returns The user-provided StateBlock or an Exception

initialize (*outlvl=0, optarg={}, state_args=None, solver='ipopt', hold_state=False*)

Initialization routine for separator (default solver ipopt)

Keyword Arguments

- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default=None)
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')
- **state_args** – unused, but retained for consistency with other initialization methods
- **hold_state** – flag indicating whether the initialization routine should unfix any state variables fixed during initialization, **default** - False.
Valid values: **True** - states variables are not unfixed, and a dict of returned containing flags for which states were fixed during initialization, **False** - state variables are unfixed after initialization by calling the *release_state* method.

Returns If `hold_states` is `True`, returns a dict containing flags for which states were fixed during initialization.

model_check()

This method executes the `model_check` methods on the associated state blocks (if they exist). This method is generally called by a unit model as part of the unit's `model_check` method.

Parameters None –

Returns None

partition_outlet_flows (*mb, outlet_list*)

Creates outlet Port objects and tries to partition mixed stream flows between these

Parameters

- **representing the mixed flow to be split**
(*StateBlock*) –
- **list of names for outlets**
(*a*) –

Returns None

release_state (*flags, outlvl=0*)

Method to release state variables fixed during initialization.

Keyword Arguments

- **flags** – dict containing information of which state variables were fixed during initialization, and should now be unfixed. This dict is returned by `initialize` if `hold_state = True`.
- **outlvl** – sets output level of logging

Returns None

StateJunction Block

The IDAES `StateJunction` block represents a pass-through unit or simple pipe with no holdup. The primary use for this unit is in conceptual design applications for linking Arcs to/from different process alternatives.

Degrees of Freedom

StateJunctions have no degrees of freedom.

Model Structure

A StateJunction consists of a single StateBlock with two Ports (inlet and outlet), where the state variables in the state block are simultaneously connected to both Ports.

Additional Constraints

StateJunctions write no additional constraints beyond those in the StateBlock.

Variables

StateJunctions have no additional variables.

StateJunction Class

```
class idaes.generic_models.unit_models.statejunction.StateJunction(*args,  
                                                                **kwds)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this unit will be dynamic or not, **default** = False.

has_holdup Indicates whether holdup terms should be constructed or not. **default** - False. StateJunctions do not have defined volume, thus this must be False.

property_package Property parameter object used to define property state block, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (StateJunction) New instance

StateJunctionData Class

class idaes.generic_models.unit_models.statejunction.**StateJunctionData** (*component*)
Standard StateJunction Unit Model
Class

build()
Begin building model. :param None:

Returns None

initialize (*state_args={}*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)
This method initializes the StateJunction block by calling the initialize method on the property block.

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine

- **optarg** – solver options dictionary object (default={ 'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

Stoichiometric (Yield) Reactor

The IDAES Stoichiometric reactor model represents a unit operation where a single material stream undergoes some chemical reaction(s) subject to a set of extent or yield specifications.

Degrees of Freedom

Stoichiometric reactors generally have degrees of freedom equal to the number of reactions + 1.

Typical fixed variables are:

- reaction extents or yields (1 per reaction),
- reactor heat duty (has_heat_transfer = True only).

Model Structure

The core Stoichiometric reactor unit model consists of a single ControlVolume0DBlock (named control_volume) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Variables

Stoichiometric reactors units add the following variables:

Variable	Name	Notes
Q_t	heat	Only if has_heat_transfer = True, reference to control_volume.heat
ΔP_t	pressure change	Only if has_pressure_change = True, reference to control_volume.deltaP

Constraints

Stoichiometric reactor units write no additional Constraints beyond those written by the control_volume Block.

StoichiometricReactor Class

```
class idaes.generic_models.unit_models.stoichiometric_reactor.StoichiometricReactor(*args,  
**kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use

total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - **EnergyBalanceType.useDefault**. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - **MomentumBalanceType.pressureTotal**. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_heat_of_reaction Indicates whether terms for heat of reaction should be constructed, **default** - **False**. **Valid values:** { **True** - include heat of reaction terms, **False** - exclude heat of reaction terms.}

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - **False**. **Valid values:** { **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - **False**. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change

terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object.}

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object.}

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (StoichiometricReactor) New instance

StoichiometricReactorData Class

class `idaes.generic_models.unit_models.stoichiometric_reactor.StoichiometricReactorData` (*component*)

Standard Stoichiometric Reactor Unit Model Class This model assumes that all given reactions are irreversible, and that each reaction has a fixed `rate_reaction` extent which has to be specified by the user.

build()

Begin building model (pre-DAE transformation). :param None:

Returns None

Translator Block

Translator blocks are used in complex flowsheets where the user desires to use different property packages for different parts of the flowsheet. In order to link two streams using different property packages, a translator block is required.

The core translator block provides a general framework for constructing Translator Blocks, however users need to add constraints to map the incoming states to the outgoing states as required by their specific application.

Degrees of Freedom

The degrees of freedom of Translator blocks depends on the property packages being used, and the user should write a sufficient number of constraints mapping inlet states to outlet states to satisfy these degrees of freedom.

Model Structure

The core Translator Block consists of two State Blocks, names `properties_in` and `properties_out`, which are linked to two Ports names `inlet` and `outlet` respectively.

Additional Constraints

The core Translator Block writes no additional constraints. Users should add constraints to their instances as required.

Variables

Translator blocks add no additional Variables.

Translator Class

```
class idaes.generic_models.unit_models.translator.Translator(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Translator blocks are always steady-state.

has_holdup Translator blocks do not contain holdup.

outlet_state_defined Indicates whether unit model will fully define outlet state. If False, the outlet property package will enforce constraints such as sum of mole fractions and phase equilibrium. **default** - True. **Valid values:** { **True** - outlet state will be fully defined, **False** - outlet property package should enforce summation and equilibrium constraints. }

has_phase_equilibrium Indicates whether outlet property package should enforce phase equilibrium constraints. **default** - False. **Valid values:** { **True** - outlet property package should calculate phase equilibrium, **False** - outlet property package should not calculate phase equilibrium. }

inlet_property_package Property parameter object used to define property calculations for the incoming stream, **default** - None. **Valid values:** { **PhysicalParameterObject** - a **PhysicalParameterBlock** object. }

inlet_property_package_args A **ConfigBlock** with arguments to be passed to the property block associated with the incoming stream, **default** - None. **Valid values:** { see property package for documentation. }

outlet_property_package Property parameter object used to define property calculations for the outgoing stream, **default** - None. **Valid values:** { **PhysicalParameterObject** - a **PhysicalParameterBlock** object. }

outlet_property_package_args A **ConfigBlock** with arguments to be passed to the property block associated with the outgoing stream, **default** - None. **Valid values:** { see property package for documentation. }

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Translator) New instance

TranslatorData Class

class `idaes.generic_models.unit_models.translator.TranslatorData` (*component*)
Standard Translator Block Class

build()
Begin building model.

Parameters None –

Returns None

initialize (*state_args_in={}*, *state_args_out={}*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)
This method calls the initialization

method of the state blocks.

Keyword Arguments

- **state_args_in** – a dict of arguments to be passed to the inlet property package (to provide an initial state for initialization (see documentation of the specific property package) (default = {})).
- **state_args_out** – a dict of arguments to be passed to the outlet property package (to provide an initial state for initialization (see documentation of the specific property package) (default = {})).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={ ‘tol’: 1e-6})
- **solver** – str indicating which solver to use during initialization (default = ‘ipopt’)

Returns None

Turbine

The Turbine model is a *PressureChanger*, where the configuration is set so that the “compressor” option can only be False, and the default “thermodynamic_assumption” is “isentropic.” See the *PressureChanger documentation* for details.

Example

The example below demonstrates the basic Turbine model usage:

```
import pyomo.environ as pyo
from idaes.
↳core import FlowsheetBlock
from idaes.generic_models.
↳unit_models import Turbine
from idaes.generic_models.
↳properties import iapws95

m = pyo.ConcreteModel()
m.fs_
↳= FlowsheetBlock(default=
↳{"dynamic": False})
```

(continues on next page)

(continued from previous page)

```

m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.fs.unit =
↳ Turbine(default={"property_
↳ package": m.fs.properties})

m.fs.unit.
↳ inlet.flow_mol[0].fix(1000)
m.fs.unit.inlet.
↳ enth_mol[0].fix(iapws95.
↳ htpx(T=800*pyo.units.
↳ K, P=1e7*pyo.units.Pa))
m.fs.unit.
↳ inlet.pressure[0].fix(1e7)
m.fs.unit.deltaP.fix(-2e6)
m.fs.unit.entropy_
↳ isentropic.fix(0.9)

```

Valve

This section describes the generic adiabatic valve model. By default the model is based on molar flow, but the pressure-flow equation and the flow basis is configurable. This model inherits the *PressureChanger model* with the adiabatic options. Beyond the base pressure changer model this provides a pressure flow relation as a function of the valve opening fraction.

Example

```

from pyomo.environ_
↳ import ConcreteModel,
↳ SolverFactory,
↳ TransformationFactory

from idaes.
↳ core import FlowsheetBlock
from idaes.generic_models.
↳ unit_models import Valve
from idaes.generic_models.
↳ properties import iapws95
import idaes.
↳ core.util.scaling as iscale

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()

```

(continues on next page)

(continued from previous page)

```

m.fs.valve_
↳ Valve(default={"property_
↳ package": m.fs.properties})
fin = 900 # mol/s
pin = 200000 # Pa
pout = 100000 # Pa
tin = 300 # K
hin_
↳ iapws95.htpx(T=tin*units.
↳ K, P=pin*units.Pa) # J/mol
# Calculate the flow_
↳ coefficient to give 1000_
↳ mol/s flow with given P
cv = 1000/
↳ math.sqrt(pin - pout)/0.5
# set inlet
m.fs.valve.
↳ inlet.enth_mol[0].fix(hin)
m.fs.valve.
↳ inlet.flow_mol[0].fix(fin)
m.fs.valve.
↳ inlet.flow_mol[0].unfix()
m.fs.valve.
↳ inlet.pressure[0].fix(pin)
m.fs.valve.outlet.
↳ pressure[0].fix(pout)
m.fs.valve.Cv.fix(cv)
m.fs.valve.
↳ valve_opening.fix(0.5)
iscale.calculate_
↳ scaling_factors(m)
m.fs.
↳ valve.initialize(outlvl=1)

solver =_
↳ pyo.SolverFactory("ipopt")
solver.
↳ options = {"nlp_scaling_
↳ method": "user-scaling"}
solver(m, tee=True)

```

Variables

Variable	Symbol	Index Sets	Doc
Cv	C_v	None	Valve coefficient
valve_opening	x	time	The fraction that the valve is open from 0 to 1

The Cv variable is highly recommended but can be omitted in custom pressure-flow relations.

Expressions

Expression	Sym- bol	Index Sets	Doc
valve_function	$f(x)$	time	This is a valve function that describes how the fraction open affects flow.

Built-in Valve Functions

Standard valve functions can be specified by providing a `ValveFunctionType` enumerated type to the `valve_function_callback` argument. Standard functions are given below.

`ValveFunctionType.linear`

$$f(x) = x$$

`ValveFunctionType.
quick_opening`

$$f(x) = \sqrt{x}$$

`ValveFunctionType.
equal_percentage`

$$f(x) = \alpha^{x-1}$$

For the equal-percentage valve function an additional variable `alpha` is defined which by default is fixed and set to 100.

Custom Valve Functions

In general, the valve opening should be restricted to range from 0 to 1. The valve function should be a named expression attached to the valve model called `valve_function` which takes the valve opening and computes a value that goes from approximately zero when valve opening is 0 to 1 when the valve opening is one. The valve function can have parameters as needed, so custom valve functions are defined using a callback function.

The callback function should take an object of the `Valve` class as an argument and add the `valve_function`

named expression. Any additional parameters can also be added. The standard equal-percentage valve function is provided below as an example. The callback can be provided for the `valve_function_callback` configuration option.

```
def equal_percentage_cb(b):
    """
    Equal percentage
    ↪ valve function callback.
    """
    #
    ↪ Parameters can be defined
    ↪ as Var or Param. If
    ↪ Var is used the parameter
    ↪ # can
    ↪ be included in a parameter
    ↪ estimation problem.
    b.alpha = pyo.
    ↪ Var(initialize=100, doc=
    ↪ "Valve function parameter")
    b.alpha.fix()
    @b.Expression(b.
    ↪ flowsheet().config.time)
    ↪
    ↪ def valve_function(b2, t):
    ↪     return b2.alpha **
    ↪ (b2.valve_opening[t] - 1)
```

Constraints

The pressure flow relation is added to the inherited constraints from the *PressureChanger model*.

The default pressure-flow relation is given below where F is the molar flow. The default valve function assumes an incompressible fluid of constant density. In this case the fluid specific gravity is included in the flow coefficient. For rigorous modeling of valves with gases, it is recommended that a custom pressure-flow equation be specified.

$$F^2 = C_v^2 (P_{in} - P_{out}) f(x)^2$$

Custom Pressure Flow Relations

Other pressure-flow equations can be specified via callback supplied to the unit configuration option `pressure_flow_callback`. The callback allows both the form and flow basis of the pressure-flow equation to be specified.

The callback can add parameters and variables as needed. It is recommended that only the `pressure_flow_equation` be specified as additional constraints would not be scaled by the valve model's scaling routines. The pressure flow relation generally should be written in the form below to facilitate scaling where F is flow variable.

$$f_1(F) = f_2(P_{in}, P_{out})$$

The callback takes a Valve model object as an argument. There are three attributes that the `pressure_flow_callback` should define:

1. `flow_var` a time indexed reference to the flow variable basis,
2. `pressure_flow_equation_scale` a function that takes `flow_var` and defines the form of the flow term
3. `pressure_flow_equation` the pressure flow relation constraint.

The first two items, `flow_var` and `pressure_flow_equation_scale`, are not directly used in the model, but are used by the model scaling routine.

The example callback below is the model default pressure-flow equation.

```
def pressure_
    ↳ flow_default_callback(b):
        """
        Add the default pressure_
        ↳ flow relation constraint.
        ↳ This will be used in the
        valve model, a custom_
        ↳ callback is provided.
        """
    ↳
    ↳ umeta = b.config.property_
    ↳ package.get_metadata().
    ↳ get_derived_units
```

(continues on next page)

(continued from previous page)

```

    b.Cv = pyo.Var(
        initialize=0.1,
        doc=
↪ "Valve flow coefficient",
        units=umeta(
↪ "amount") / umeta("time
↪ ") / umeta("pressure") ** 0.5
        )
    b.Cv.fix()

    b.flow_var = pyo.
↪ Reference(b.control_volume.
↪ properties_in[:].flow_mol)
    b.pressure_flow_equation_
↪ scale = lambda x : x**2

    @b.Constraint(b.
↪ flowsheet().config.time)
    def pressure_
↪ flow_equation(b2, t):
        ↪
↪     Po = b2.control_volume.
↪     properties_out[t].pressure
        ↪
↪     Pi = b2.control_volume.
↪     properties_in[t].pressure
        ↪
↪     F = b2.control_volume.
↪     properties_in[t].flow_mol
        Cv = b2.Cv
        ↪
↪     fun = b2.valve_function[t]
        ↪     return F ** 2 == Cv
↪ ** 2 * (Pi - Po) * fun ** 2

```

Initialization

This just calls the initialization routine from PressureChanger. Either an outlet pressure value or deltaP can be specified to aid the initialization.

Valve Class

```
class idaes.generic_models.unit_models.valve.Valve(*args, **kws)
    Adiabatic valves
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.

- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`.

Valid values: { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = `False`.

Valid values: { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

compressor Indicates whether this unit should be considered a compressor (`True` (default), pressure increase) or an expander (`False`, pressure decrease).

thermodynamic_assumption Flag to set the thermodynamic assumption to use for the unit. - `ThermodynamicAssumption.isothermal` (default) - `ThermodynamicAssumption.isentropic` - `ThermodynamicAssumption.pump` - `ThermodynamicAssumption.adiabatic`

property_package Property parameter object used to define property calculations, **default** - `useDefault`. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a `PropertyParameterBlock` object. }

property_package_args A `ConfigBlock` with arguments to be passed to a property block(s) and used when constructing these, **default** - `None`. **Valid values:** { see property package for documentation. }

valve_function_callback This takes either an enumerated valve function type in: { `ValveFunctionType.linear`, `ValveFunctionType.quick_opening`, `ValveFunctionType.equal_percentage`,

ValveFunctionType.custom} or a callback function that takes a valve model object as an argument and adds a time-indexed valve_function expression to it. Any additional required variables, expressions, or constraints required can also be added by the callback.

pressure_flow_callback This callback function takes a valve model object as an argument and adds a time-indexed valve_function expression to it. Any additional required variables, expressions, or constraints required can also be added by the callback.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Valve) New instance

ValveData Class

```
class idaes.generic_models.unit_models.valve.ValveData (component)
```

```
build()
```

Parameters None –

Returns None

```
calculate_scaling_factors()
```

Calculate pressure flow constraint scaling from flow variable scale.

```
initialize (state_args={}, outlvl=0, solver='ipopt', optarg={'max_iter': 30, 'tol': 1e-06})
```

Initialize the valve based on a deltaP guess.

Parameters

- **state_args** (*dict*) – Initial state for property initialization
- **outlvl** – sets output level of initialization routine

- **solver** (*str*) – Solver to use for initialization
- **optarg** (*dict*) – Solver arguments dictionary

Control Models

This section contains documentation for core IDAES control models.

Contents

Proportional-Integral-Derivative (PID) Controller

The IDAES framework contains a basic PID control implementation, which is described in this section.

Example

The following code demonstrated the creation of a PIDBlock, but for simplicity, it does not create a dynamic process model. A full example of a dynamic process with PID control is being prepared for the IDAES examples repository and will be referenced here once completed.

The valve opening is the controlled output variable and pressure “1” is the measured variable. The controller output for the valve opening is restricted to be between 0 and 1. The measured and output variables should be indexed only by time. Fortunately there is no need to create new variables if the variables are in a property block or not indexed only by time. Pyomo’s Reference objects can be used to create references to existing variables with the proper indexing as shown in the example.

The `calculate_initial_integral` option calculates the integral error in the first time step to match the initial controller output. This keeps the controller output from immediately jumping to a new value. Unless the initial integral error is known, this option should usually be True.

The controller should be added after the DAE expansion is done. There are several variables in the controller that are usually meant to be fixed; as shown in the example, they are gain, time_i, time_d, and setpoint. For more information about the variables, expressions, and parameters in the PIDBlock, model see *Variables and Expressions*.

```
from idaes.  
↳generic_models.control_  
↳import PIDBlock, PIDForm  
from idaes.  
↳core import FlowsheetBlock  
import pyomo.environ as pyo  
  
m = pyo.ConcreteModel(name=  
↳"PID Example")  
m.fs_  
↳= FlowsheetBlock(default=  
↳{"dynamic  
↳":True, "time_set":[0,10]})  
  
m.fs.valve_  
↳opening = pyo.Var(m.fs.  
↳time, doc="Valve opening")  
m.fs.pressure = pyo.  
↳Var(m.fs.time, [1,2], doc=  
↳"Pressure in unit 1 and 2")  
  
pyo.TransformationFactory(  
↳'dae.finite_  
↳difference').apply_to(  
    m.fs,  
    nfe=10,  
    wrt=m.fs.time,  
    scheme='BACKWARD',  
)  
  
m.fs.measured_  
↳variable = pyo.Reference(m.  
↳fs.pressure[:,1])  
  
m.fs.ctrl = PIDBlock(  
    default={  
        "pv  
↳":m.fs.measured_variable,  
        "output  
↳":m.fs.valve_opening,  
        "upper":1.0,  
        "lower":0.0,  
        "calculate_  
↳initial_integral":True,  
        "pid_  
↳form":PIDForm.velocity,  
    }
```

(continues on next page)

(continued from previous page)

```
)
m.fs.ctrl.gain.fix(1e-6)
m.fs.ctrl.time_i.fix(0.1)
m.fs.ctrl.time_d.fix(0.1)
m.fs.ctrl.setpoint.fix(3e5)
```

Controller Windup

The current PID controller model has no integral windup prevention. This will be added to the model in the near future.

Class Documentation

class `idaes.generic_models.control.pid_controller.PIDBlock(*args, **kwargs)`

This is a PID controller block. The PID Controller block must be added after the DAE transformation.

Args: rule (function): A rule function or None. Default rule calls build(). concrete (bool): If True, make this a toplevel model. **Default** - False. ctype (str): Pyomo ctype of the block. **Default** - "Block" default (dict): Default ProcessBlockData config

Keys

pv A Pyomo Var, Expression, or Reference for the measured process variable. Should be indexed by time.

output A Pyomo Var, Expression, or Reference for the controlled process variable. Should be indexed by time.

upper The upper limit for the controller output, default=1

lower The lower limit for the controller output, default=0

calculate_initial_integral Calculate the initial integral term value if true, otherwise provide a variable `err_i0`, which can be fixed, default=True

pid_form Velocity or standard form

initialize (dict): ProcessBlockData config for individual elements. Keys

are BlockData indexes and values are dictionaries described under the "default" argument above.

idx_map (function): Function to take the index of a BlockData element and

return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns: (PIDBlock) New instance

```
class idaes.generic_models.control.pid_controller.PIDBlockData (component)
```

build()

Build the PID block

Variables and Expressions

Symbol	Name in Model	Description
$v_{sp}(t)$	setpoint[t]	Setpoint variable (usually fixed)
$v_{mv}(t)$	pv[t]	Measured process variable reference
$u(t)$	output[t]	Controller output variable reference
$K_p(t)$	gain[t]	Controller gain (usually fixed)
$T_i(t)$	time_i[t]	Integral time (usually fixed)
$T_d(t)$	time_d[t]	Derivative time (usually fixed)
$e(t)$	err[t]	Error expression (setpoint - pv)
–	err_d[t]	Derivative error expression
–	err_i[t]	Integral error expression (standard form)
–	err_d0	Initial derivative error value (fixed)
$e_{integral}(0)$	err_i0	Initial integral error value (fixed)
–	err_i_end	Last initial integral error expression
–	limits["h"]	Upper limit of output parameter
–	limits["l"]	Lower limit of output parameter
–	smooth_eps	Smooth min/max parameter

Formulation

There are two forms of the PID controller equation. The standard formulation can result in equations with very large summations. In the velocity form of the equation the controller output can be calculated based only on the previous state.

The two forms of the equations are equivalent, but the choice of form will affect robustness and solution time. It is not necessarily clear that the velocity form of the equation is always more numerically favorable, however it would usually be the default choice. Both

forms are provided in case the standard form works better in some situations.

Standard Formulation

The PID controller equations are given by the following equations

$$e(t) = v_{sp}(t) - v_{mv}(t)$$

$$u(t) = K_p \left[e(t) + \frac{1}{T_i} \int_0^t e(s) ds + T_d \frac{de(t)}{dt} \right]$$

The PID equation now must be discretized.

$$u(t_i) = K_p \left[e(t_i) + \frac{e_{integral}(0)}{T_i} + \frac{1}{T_i} \sum_{j=0}^{i-1} \Delta t_j \frac{e(t_j) + e(t_{j+1})}{2} + T_d \frac{e(t_i) - e(t_{i-1})}{\Delta t_{i-1}} \right]$$

Velocity Formulation

The velocity formulation of the PID equation may also be useful. The way the equations are written in the PID model, the integral term is a summation expression and as time increases the integral term will build up an increasing number of terms potentially becoming very large. This potentially has two affects, increasing round off error and computation time. The velocity formulation allows the controller output to be calculated based on the previous output.

First the usual PID controller equation can be rearranged to solve for the integral error.

$$\frac{1}{T_i} \int_0^t e(s) ds = \frac{u(t)}{K_p} - e(t) - T_d \frac{de(t)}{dt}$$

The PID equation for some time $(t + \Delta t)$ is

$$u(t + \Delta t) = K_p \left[e(t + \Delta t) + \frac{1}{T_i} \int_0^{t+\Delta t} e(s) ds + T_d \frac{de(t + \Delta t)}{dt} \right]$$

$$u(t + \Delta t) = K_p \left[e(t + \Delta t) + \frac{1}{T_i} \int_t^{t+\Delta t} e(s) ds + \frac{1}{T_i} \int_0^t e(s) ds + T_d \frac{de(t + \Delta t)}{dt} \right]$$

$$u(t + \Delta t) = u(t) + K_p \left[e(t + \Delta t) - e(t) + \frac{1}{T_i} \int_t^{t+\Delta t} e(s) ds + T_d \left(\frac{de(t + \Delta t)}{dt} - \frac{de(t)}{dt} \right) \right]$$

Now we can discretize the equation using the trapezoid rule for the integral.

$$u(t + \Delta t) = u(t) + K_p \left[e(t + \Delta t) - e(t) + \frac{\Delta t}{T_i} \left(\frac{e(t + \Delta t) + e(t)}{2} \right) + T_d \left(\frac{de(t + \Delta t)}{dt} - \frac{de(t)}{dt} \right) \right]$$

Since the derivative error term will require the error at the previous time step to calculate, this form will still result in a large summation being formed since in the model there is no derivative error variable. To avoid this problem, the derivative error term can equivalently be replaced with the derivative of the negative measured process variable.

$$u(t + \Delta t) = u(t) + K_p \left[e(t + \Delta t) - e(t) + \frac{\Delta t}{T_i} \left(\frac{e(t + \Delta t) + e(t)}{2} \right) + T_d \left(\frac{dv_{mv}(t + \Delta t)}{dt} - \frac{dv_{mv}(t)}{dt} \right) \right]$$

Now the velocity form of the PID controller equation can be calculated at each time point from just the state at the previous time point.

Substitution

In both the proportional and integral terms, error can be replaced with the negative measured process variable yielding equivalent results. This substitution is provided by the PID class and is done by default.

Output Limits

Smooth min and smooth max expressions are used to keep the controller output between a minimum and maximum value.

Power Generation Model Library

The IDAES Process Modeling Framework contains a library of models specifically developed for modeling power generation systems. These models all built off of the core IDAES modeling framework and model libraries.

Unit Models

Feedwater Heater (0D)

The FWH0D model is a 0D feedwater heater model suitable for steady state modeling. It is intended to be used primarily with the [IAWPS95](#) property package. The feedwater heater is split

into three sections the condensing section is required while the desuperheating and drain cooling sections are optional. There is also an optional mixer for adding a drain stream from another feedwater heater to the condensing section. The figure below shows the layout of the feedwater heater. All but the condensing section are optional.

Fig. 15: Feedwater Heater

Example

The example below shows how to setup a feedwater heater with all three sections. The feedwater flow rate, steam conditions, heat transfer coefficients and areas are not necessarily realistic.

```
import pyomo.environ as pyo
from idaes.
↳core import FlowsheetBlock
from idaes.
↳generic_models.unit_models.
↳heat_exchanger import (
    delta_temperature_
↳underwood_callback,
    delta_
↳temperature_lmtcd_callback)
from idaes.generic_models.
↳properties import iapws95
from idaes.power_generation.
↳unit_models import FWHOD

def make_fwh_model():
    ↳
    ↳model = pyo.ConcreteModel()
    ↳model.fs_
    ↳= FlowsheetBlock(default={
        "dynamic": False,
        "default_
↳property_package": iapws95.
↳Iapws95ParameterBlock() })
    ↳model.fs.properties_
    ↳= model.fs.config.
    ↳default_property_package
    ↳model.
    ↳fs.fwh = FWHOD(default={
        ↳
        ↳"has_desuperheat":True,
        ↳
        ↳"has_drain_cooling":True,
        ↳
        ↳"has_drain_mixer":True,
```

(continues on next page)

(continued from previous page)

```

    "property_package
↳ ":model.fs.properties})

    model.fs.fwh.desuperheat.
↳ inlet_1.flow_mol.fix(100)
    model.fs.fwh.desuperheat.
↳ inlet_1.flow_mol.unfix()
    model.
↳ fs.fwh.desuperheat.inlet_
↳ 1.pressure.fix(201325)
    model.fs.fwh.desuperheat.
↳ inlet_1.enth_mol.fix(60000)
    model.fs.fwh.drain_
↳ mix.drain.flow_mol.fix(1)
    model.fs.fwh.drain_mix.
↳ drain.pressure.fix(201325)
    model.fs.fwh.drain_mix.
↳ drain.enth_mol.fix(20000)
    model.fs.fwh.cooling.
↳ inlet_2.flow_mol.fix(400)

↳
↳ model.fs.fwh.cooling.inlet_
↳ 2.pressure.fix(101325)
    model.fs.fwh.cooling.
↳ inlet_2.enth_mol.fix(3000)
    model.fs.
↳ fwh.condense.area.fix(1000)
    model.fs.fwh.condense.
↳ overall_heat_transfer_
↳ coefficient.fix(100)
    model.fs.fwh.
↳ desuperheat.area.fix(1000)
    model.fs.fwh.desuperheat.
↳ overall_heat_transfer_
↳ coefficient.fix(10)
    model.fs.
↳ fwh.cooling.area.fix(1000)
    model.fs.fwh.cooling.
↳ overall_heat_transfer_
↳ coefficient.fix(10)

    model.fs.fwh.initialize()
    return(model)

# create a feedwater heater_
↳ model with all optional_
↳ units and initialize
model = make_fwh_model()

```

Model Structure

The condensing section uses the *FWHCondensing0D* model to calculate a steam flow rate such that all steam is condensed in the condensing section. This allows turbine steam extraction rates to be calculated. The other sections are regular *HeatExchanger* models. The table below shows the unit models which make up the feedwater heater, and the option to include or exclude them.

Unit	Option	Doc
condense	–	Condensing section (<i>FWHCondensing0D</i>)
desuperheat	has_desuperheat	Desuperheating section (<i>HeatExchanger</i>)
cooling	has_drain_cooling	Drain cooling section (<i>HeatExchanger</i>)
drain_mix	has_drain_mixer	Mixer for steam and other FWH drain (<i>Mixer</i>)

Degrees of Freedom

The area and overall_heat_transfer_coefficient should be fixed or constraints should be provided to calculate overall_heat_transfer_coefficient. If the inlets are also fixed except for the inlet steam flow rate (inlet_1.flow_mol), the model will have 0 degrees of freedom.

See *FWH0D* and *FWH0DData* for full Python class details.

Feedwater Heater (Condensing Section 0D)

The condensing feedwater heater is the same as the *HeatExchanger* model with one additional constraint to calculate the inlet flow rate such that all the entering steam is condensed. This model is suitable for steady state modeling, and is intended to be used with the *IAPWS95* property package. For dynamic modeling, the 1D feedwater heater models should be used (not yet publicly available).

Degrees of Freedom

Usually area and overall_heat_transfer_coefficient are fixed or constraints provided to calculate overall_heat_transfer_coefficient. If the inlets are also fixed except for the inlet steam flow rate (inlet_1.flow_mol), the model will have 0 degrees of freedom.

Variables

The variables are the same as *HeatExchanger*.

Constraints

In addition to the *HeatExchanger* constraints, there is one additional constraint to calculate the inlet steam flow such that all steam condenses. The constraint is called *extraction_rate_constraint*, and is defined below.

$$h_{steam,out} = h_{sat,liquid}(P)$$

Where h is molar enthalpy, and the saturated liquid enthalpy is a function of pressure.

FWHCondensing0D Class

```
class idaes.power_generation.unit_models.feedwater_heater_0D.FWHCondensing0D(*args,  
                                                                           **kwds)
```

Feedwater Heater Condensing Section
The feedwater heater condensing section model is a normal 0D heat exchanger model with an added constraint to calculate the steam flow such that the outlet of shell is a saturated liquid.

Args: rule (function): A rule function or None. Default rule calls build(). concrete (bool): If True, make this a

toplevel model. **Default** - False. ctype (str): Pyomo ctype of the block. **Default** - "Block" default (dict): Default ProcessBlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { useDefault - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { useDefault - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

hot_side_name Hot side name, sets control volume and inlet and outlet names

cold_side_name Cold side name, sets control volume and inlet and outlet names

hot_side_config A config block used to construct the hot side control volume. This config can be given by the hot side name instead of hot_side_config.

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { MaterialBalanceType.useDefault - refer to property package for default balance type **MaterialBalanceType.none - exclude material balances, MaterialBalanceType.componentPhase - use phase component balances, MaterialBalanceType.componentTotal - use total component balances, MaterialBalanceType.elementTotal - use total element balances, MaterialBalanceType.total - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { EnergyBalanceType.useDefault - refer to property package for default

balance_type ****EnergyBalanceType**.**none** - exclude energy balances, **EnergyBalanceType**.**enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType**.**enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType**.**energyTotal** - single energy balance for material, **EnergyBalanceType**.**energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - **MomentumBalanceType**.**pressureTotal**. **Valid values:** { **MomentumBalanceType**.**none** - exclude momentum balances, **MomentumBalanceType**.**pressureTotal** - single pressure balance for material, **MomentumBalanceType**.**pressurePhase** - pressure balances for each phase, **MomentumBalanceType**.**momentumTotal** - single momentum balance for material, **MomentumBalanceType**.**momentumPhase** - momentum balances for each phase.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = **False**. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - **False**. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations, **default** - **useDefault**. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a **PropertyParameterBlock** object.}

property_package_args A **ConfigBlock** with arguments to be passed to a property block(s) and used when constructing these, **default** - **None**. **Valid values:** { see property package for documentation.}

cold_side_config A config block used to construct the cold side control volume. This config can be given by the cold side name instead of cold_side_config.

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.useDefault`. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum

balances for each phase.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object.}

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

delta_temperature_callback Callback for for temperature difference calculations

flow_pattern Heat exchanger flow pattern, **default** - HeatExchangerFlowPattern.countercurrent. **Valid values:** { **HeatExchangerFlowPattern.countercurrent** - countercurrent flow, **HeatExchangerFlowPattern.cocurrent** - cocurrent flow, **HeatExchangerFlowPattern.crossflow** - cross flow, factor times countercurrent temperature difference.}

initialize (dict): ProcessBlockData config for individual elements. Keys

are BlockData indexes and values are dictionaries described under the “default” argument above.

idx_map (function): Function to take the index of a BlockData element and

return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns: (FWHCondensing0D) New instance

FWHCondensing0DData Class

class `idaes.power_generation.unit_models.feedwater_heater_0D.FWHCondensing0DData` (*component*)

build()
Building model

Parameters None –

Returns None

initialize (**args, **kwargs*)
Use the regular heat exchanger initialization, with the extraction rate constraint deactivated; then it activates the constraint and calculates a steam inlet flow rate.

Turbine (Isentropic)

This is a steam power generation turbine model for the basic isentropic turbine calculations.

It is the basis of the *TurbineInletStage*, *TurbineOutletStage*

and, *TurbineOutletStage* *<technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (OutletStage)>*, and, *TurbineOutletStage* *<technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (Stage)>* models.

Variables

Variable	Symbol	Index Sets	Doc
efficiency_isentropic	η_{isen}	time	Isentropic efficiency
deltaP	ΔP	time	Pressure change ($P_{out} - P_{in}$) [Pa]
ratioP	P_{ratio}	time	Ratio of discharge pressure to inlet pressure $\left(\frac{P_{out}}{P_{in}}\right)$

Expressions

This model provides two expressions that are not available in the pressure changer model.

Expression	Symbol	Index Sets	Doc
h_is	h_{is}	time	Isentropic outlet molar enthalpy [J/mol]
delta_enth_isentropic	Δh_{is}	time	Isentropic enthalpy change ($h_{is} - h_{in}$) [J/mol]
work_isentropic	w_{is}	time	Isentropic work (W)

Constraints

In addition to the mass and energy balances provided by the control volume the following equation is used to calculate the outlet enthalpy, so work comes from the control volume energy balance.

$$h_{out} = h_{in} - \eta_{is} (h_{in} - h_{is})$$

Initialization

To initialize the turbine model, a reasonable guess for the inlet condition and deltaP and efficiency should be set by setting the appropriate variables.

TurbineStage Class

```
class idaes.power_generation.unit_models.helm.turbine.HelmIsentropicTurbine(*args,
                                                                              **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.

- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`.

Valid values: { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = `False`.

Valid values: { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - `False`. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations, **default** - `useDefault`. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a `PropertyParameterBlock` object. }

property_package_args A `ConfigBlock` with arguments to be passed to a property block(s) and used when constructing these, **default** - `None`. **Valid values:** { see property package for documentation. }

has_work_transfer True if model has a work transfer term.

has_heat_transfer True if model has a heat transfer term.

- **initialize** (*dict*) - `ProcessBlockData` config for individual elements. Keys are `BlockData` indexes and values are dictionaries described under the “default” argument above.

- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HelmIsentropicTurbine) New instance

TurbineStageData Class

class `idaes.power_generation.unit_models.helm.turbine.HelmIsentropicTurbineData` (*component*)

Basic isentropic 0D turbine model. This inherits the heater block to get a lot of unit model boilerplate and the mass balance, energy balance and pressure equations. This model is intended to be used only with Helmholtz EOS property packages in mixed or single phase mode with P-H state vars.

Since this inherits BalanceBlockData, and only operates in steady-state or pseudo-steady-state (for dynamic models) the following mass, energy and pressure equations are implicitly written.

- 1) **Mass Balance:** $0 = \text{flow_mol_in}[t] - \text{flow_mol_out}[t]$
- 2) **Energy Balance:** $0 = (\text{flow_mol}[t]*h_mol[t])_in - (\text{flow_mol}[t]*h_mol[t])_out + Q_in + W_in$
- 3) **Pressure:** $0 = P_in[t] + \text{deltaP}[t] - P_out[t]$

build ()

Add model equations to the unit model. This is called by a default block construction rule when the unit model is created.

initialize (*outlvl=0, solver='ipopt', optarg={'tol': 1e-06}*)

For simplicity this initialization requires you to set values for the efficiency, inlet, and one of pressure ratio, pressure change or outlet pressure.

Turbine (Inlet Stage)

This is a steam power generation turbine model for the inlet stage. It inherits its *HelmIsentropicTurbine* <technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (Isentropic)>.

The turbine inlet model is based on:

Liese, (2014). “Modeling of a Steam Turbine Including Partial Arc Admission for Use in a Process Simulation Software Environment.” Journal of Engineering for Gas Turbines and Power. v136.

Example

```
from pyomo.environ_
↳ import ConcreteModel,
↳ SolverFactory,
↳ TransformationFactory,
↳ units
from idaes.
↳ core import FlowsheetBlock
from idaes.power_generation.
↳ unit_models.helm import_
↳ HelmTurbineInletStage
from idaes.generic_models.
↳ properties import iapws95

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.
↳ fs.
↳ turb_
↳ =_
↳ HelmTurbineInletStage(default=
↳ {"property_
↳ package": m.fs.properties})
hin_
↳ = iapws95.htpx(T=880*units.
↳ K, P=2.4233e7*units.Pa)
```

(continues on next page)

(continued from previous page)

```

# set inlet
m.fs.turb.
  ↳inlet[:].enth_mol.fix(hin)
m.fs.turb.inlet[:].
  ↳flow_mol.fix(26000/4.0)
m.fs.turb.inlet[:].
  ↳pressure.fix(2.4233e7)
m.fs.
  ↳turb.eff_nozzle.fix(0.95)
m.fs.turb.
  ↳blade_reaction.fix(0.9)
m.fs.turb.flow_
  ↳coeff.fix(1.053/3600.0)
m.fs.turb.
  ↳blade_velocity.fix(110.0)
m.fs.turb.
  ↳efficiency_mech.fix(0.98)

m.fs.turb.initialize()

```

Degrees of Freedom

Usually the inlet stream, or the inlet stream minus flow rate plus discharge pressure are fixed. There are also a few variables which are turbine parameters and are usually fixed, like flow coefficients. See the variables section for more information.

Model Structure

The turbine inlet model contains one *ControlVolumeODBlock* block called `control_volume` and its inheritor *HelmIsentropicTurbine* <technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (Isentropic)>.

Variables

The variables below are defined in the HelmlIsentropicTurbine model.

Variable	Sym- bol	Index Sets	Doc
blade_reaction	R	None	Blade reaction
eff_nozzle	η_{nozzle}	None	Nozzle efficiency
efficiency_mech	η_{mech}	None	Mechanical Efficiency (accounts for losses in bearings. . .)
flow_coeff	C_{flow}	None	Turbine stage flow coefficient [kg*C ^{0.5} /Pa/s]
blade_velocity	V_{rbl}	None	Turbine blade velocity (should be constant while running) [m/s]
delta_enth_isentropic	Δh_{isen}	time	Isentropic enthalpy change through stage [J/mol]

The table below shows important variables inherited from the pressure changer model.

Variable	Symbol	Index Sets	Doc
efficiency_isentropic	η_{isen}	time	Isentropic efficiency
deltaP	ΔP	time	Pressure change ($P_{out} - P_{in}$) [Pa]
ratioP	P_{ratio}	time	Ratio of discharge pressure to inlet pressure $\left(\frac{P_{out}}{P_{in}}\right)$

Expressions

Variable	Sym- bol	Index Sets	Doc
power_thermo	\dot{w}_{thermo}	time	Turbine stage power output not including mechanical loss [W]
power_shaft	\dot{w}_{shaft}	time	Turbine stage power output including mechanical loss (bearings. . .) [W]
steam_entering_velocity	V_0	time	Steam velocity entering stage [m/s]

The expression defined below provides a calculation for steam velocity entering the stage, which is used in the efficiency calculation.

$$V_0 = 1.414 \sqrt{\frac{-(1 - R)\Delta h_{isen}}{WT_{in}\eta_{nozzel}}}$$

Constraints

In addition to the constraints inherited from the

HelmTurbineStage

<technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (Stage)>, this model contains two more constraints, one to estimate efficiency and one pressure-flow relation. From the isentropic pressure changer model, these constraints eliminate the need to specify efficiency and either inlet flow or outlet pressure.

The isentropic efficiency is given by:

$$\eta_{isen} = 2 \frac{V_{rbl}}{V_0} \left[\left(\sqrt{1-R} - \frac{V_{rbl}}{V_0} \right) + \sqrt{\left(\sqrt{1-R} - \frac{V_{rbl}}{V_0} \right)^2 + R} \right]$$

The pressure-flow relation is given by:

$$\dot{m} = C_{flow} \frac{P_{in}}{\sqrt{T_{in} - 273.15}} \sqrt{\frac{\gamma}{\gamma-1} \left[\left(\frac{P_{out}}{P_{in}} \right)^{\frac{2}{\gamma}} - \left(\frac{P_{out}}{P_{in}} \right)^{\frac{\gamma+1}{\gamma}} \right]}$$

Initialization

The initialization method for this model will save the current state of the model before commencing initialization and reloads it afterwards. The state of the model will be the same after initialization, only the initial guesses for unfixed variables will be changed and optionally a flow coefficient value can be calculated. To initialize this model, provide a starting value for the inlet port variables. Then provide a guess for one of: discharge pressure, `deltaP`, or `ratioP`. Since it can be hard to determine a proper flow coefficient, the `calculate_cf` argument of the `initialize()` method can be set to `True`, and the `deltaP` guess will be used to calculate and set a corresponding flow coefficient.

The model should initialize readily, but it is possible to provide a flow coefficient that is incompatible with the given flow rate resulting in an infeasible problem.

TurbineInletStage Class

```
class idaes.power_generation.unit_models.helm.turbine_inlet.HelmTurbineInletStage(*args,  
                                     **kws)  
    Inlet stage steam turbine model
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use

total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = `False`. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - `False`. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations, **default** - `useDefault`. **Valid values:** { **useDefault** - use default

package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object.}

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

has_work_transfer True if model a has work transfer term.

has_heat_transfer True if model has a heat transfer term.

- **initialize** (*dict*) - Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HelmTurbineInletStage) New instance

TurbineInletStageData Class

class `idaes.power_generation.unit_models.helm.turbine_inlet.HelmTurbineInletStageData` (*component*)

build()

Add model equations to the unit model. This is called by a default block construction rule when the unit model is created.

initialize (*state_args={}*, *outlvl=0*, *solver='ipopt'*, *optarg={'max_iter': 30, 'tol': 1e-06}*, *calculate_cf=False*)

Initialize the inlet turbine stage model. This deactivates the specialized constraints, then does the isentropic turbine initialization, then reactivates the constraints and solves. This initialization uses a flow value guess, so some reasonable flow guess should be specified prior to initialization.

Parameters

- **state_args** (*dict*) – Initial state for property initialization
- **outlvl** (*int*) – Amount of output (0 to 3) 0 is lowest
- **solver** (*str*) – Solver to use for initialization
- **optarg** (*dict*) – Solver arguments dictionary
- **calculate_cf** (*bool*) – If True, use the flow and pressure ratio to calculate the flow coefficient.

Turbine (Outlet Stage)

This is a steam power generation turbine model for the outlet stage. The turbine outlet model is based on:

Liese, (2014). “Modeling of a Steam Turbine Including Partial Arc Admission for Use in a Process Simulation Software Environment.” Journal of Engineering for Gas Turbines and Power. v136.

Example

```
from pyomo.environ_
↳ import ConcreteModel,
↳ SolverFactory
from idaes.
↳ core import FlowsheetBlock
from idaes.power_generation.
↳ unit_models.helm import_
↳ HelmTurbineOutletStage
from idaes.generic_models.
↳ properties import iapws95

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.
↳ fs.
↳ turb_
↳ =_
↳ HelmTurbineOutletStage(default=
↳ {"property_
↳ package": m.fs.properties})
```

(continues on next page)

(continued from previous page)

```
# set inlet
m.fs.turb.inlet[:].
  ↳enth_mol.fix(47115)
m.fs.turb.inlet[:].
  ↳flow_mol.fix(15000)
m.fs.turb.
  ↳inlet[:].pressure.fix(8e4)

m.fs.turb.initialize()
```

Degrees of Freedom

Usually the inlet stream, or the inlet stream minus flow rate plus discharge pressure are fixed. There are also a few variables which are turbine parameters and are usually fixed. See the variables section for more information.

Model Structure

The turbine inlet model contains *ControlVolume0DBlock* called *turb_outstage_control_volume* and inherits the *HelmIsentropicTurbine* <technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (Isentropic)>.

Variables

The variables are defined in the *TurbineInletStage* model. Additional variables

are in
 inherited
 from the
 :HelmIsen-
 tropicTur-
 bine <tech-
 nical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine
 (Isentropic)> model.

Variable	Symbol	Index Sets	Doc
eff_dry	η_{dry}	None	Turbine efficiency when no liquid is present.
efficiency_mech	η_{mech}	None	Mechanical Efficiency (accounts for losses in bearings...)
flow_coeff	C_{flow}	None	Turbine stage flow coefficient [kg*C ^{0.5} /Pa/s]
design_exhaust_flow_vol	$V_{des,exhaust}$	None	Design volumetric flow out of stage [m ³ /s]

The table below shows important variables inherited from the pressure changer model.

Variable	Symbol	Index Sets	Doc
efficiency_isentropic	η_{isen}	time	Isentropic efficiency
deltaP	ΔP	time	Pressure change ($P_{out} - P_{in}$) [Pa]
ratioP	P_{ratio}	time	Ratio of discharge pressure to inlet pressure $\left(\frac{P_{out}}{P_{in}}\right)$

Expressions

Variable	Sym- bol	Index Sets	Doc
power_thermo	\dot{w}_{thermo}	time	Turbine stage power output not including mechanical loss [W]
power_shaft	\dot{w}_{shaft}	time	Turbine stage power output including mechanical loss (bearings...) [W]
tel	TEL	time	Total exhaust loss [J/mol]

The expression defined below provides a total exhaust loss.

$$TEL = 1 \times 10^6 * (-0.0035f^5 + 0.022f^4 - 0.0542f^3 + 0.0638f^2 - 0.0328f + 0.0064)$$

Where f is the total volumetric flow of the exhaust divided by the design flow.

Constraints

In addition to the constraints inherited from the *PressureChanger model* with the isentropic options, this model contains two more constraints, one to estimate efficiency and one pressure-flow relation. From the isentropic pressure changer model, these constraints eliminate the need to specify efficiency and either inlet flow or outlet pressure.

The isentropic efficiency is given by:

$$\eta_{isen} = \eta_{dry} x (1 - 0.65(1 - x)) * \left(1 + \frac{TEL}{\Delta h_{isen}} \right)$$

Where x is the steam quality (vapor fraction).

The pressure-flow relation is given by the Stodola Equation:

$$\dot{m} \sqrt{T_{in} - 273.15} = C_{flow} P_{in} \sqrt{1 - Pr^2}$$

Initialization

The initialization method for this model will save the current state of the model before commencing initialization and reloads it afterwards. The state of the model will be the same after initialization, only the initial guesses for unfixed variables will be changed except for optional calculation of the flow coefficient. To initialize this model, provide a starting value for the inlet port variables. Then provide a guess for one of: discharge pressure, `deltaP`, or `ratioP`. Since a good flow coefficient can be difficult to determine, the `calculate_cf` option will calculate and set a flow coefficient based on the specified inlet flow and `deltaP`.

The model should initialize readily, but it is possible to provide a flow coefficient that is incompatible with the given flow rate resulting in an infeasible problem.

TurbineOutletStage Class

```
class idaes.power_generation.unit_models.helm.turbine_outlet.HelmTurbineOutletStage(*args,
                                                                                   **kws)
```

Outlet stage steam turbine model

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:**

{ **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:**

{ see property package for documentation.}

has_work_transfer True if model has a work transfer term.

has_heat_transfer True if model has a heat transfer term.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HelmTurbineOutletStage) New instance

TurbineOutletStageData Class

class `idaes.power_generation.unit_models.helm.turbine_outlet.HelmTurbineOutletStageData` (*component*)

build()

Add model equations to the unit model. This is called by a default block construction rule when the unit model is created.

initialize (*state_args={}*, *outlvl=0*, *solver='ipopt'*, *optarg={'max_iter': 30, 'tol': 1e-06}*, *calculate_cf=True*)

Initialize the outlet turbine stage model. This deactivates the specialized constraints, then does the isentropic turbine initialization, then reactivates the constraints and solves.

Parameters

- **state_args** (*dict*) – Initial state for property initialization
- **outlvl** – sets output level of initialization routine
- **solver** (*str*) – Solver to use for initialization
- **optarg** (*dict*) – Solver arguments dictionary

Turbine (Stage)

This is a steam power generation turbine model for intermediate stages between the inlet and outlet. It inherits *HelmIsentropicTurbine* <technical_specs/model_libraries/power_generation/unit_models/turbine_inlet:Turbine (Isentropic)>.

Example

```
from pyomo.environ_
↳ import ConcreteModel,
↳ SolverFactory

from idaes.
↳ core import FlowsheetBlock
from idaes.power_generation.
↳ unit_models.helm_
↳ import HelmTurbineStage
from idaes.generic_models.
↳ properties import iapws95

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.fs.turb_
↳ = HelmTurbineStage(default=
↳ {"property_
↳ package": m.fs.properties})
# set inlet
m.fs.turb.inlet[:].
↳ enth_mol.fix(70000)
m.fs.turb.inlet[:].
↳ flow_mol.fix(15000)
m.fs.turb.
↳ inlet[:].pressure.fix(8e6)
m.fs.turb.entropy_
↳ isentropic[:].fix(0.8)
```

(continues on next page)

(continued from previous page)

```
m.fs.turb.ratioP[:].fix(0.7)
m.fs.turb.initialize()
```

Variables

Variable	Sym- bol	Index Sets	Doc
efficiency_mech	η_{mech}	None	Mechanical efficiency (accounts for losses in bearings...)
efficiency_isentropic	η_{isen}	time	Isentropic efficiency
deltaP	ΔP	time	Pressure change ($P_{out} - P_{in}$) [Pa]
ratioP	P_{ratio}	time	Ratio of discharge pressure to inlet pressure $\left(\frac{P_{out}}{P_{in}}\right)$
shaft_speed	s	time	Shaft speed [hz]

The shaft speed is used to calculate specific speed for more advanced turbine models, the specific speed expression is available, but otherwise has no effect on the model results.

Expressions

This model provides two expressions that are not available in the pressure changer model.

Variable	Sym- bol	Index Sets	Doc
power_thermo	\dot{w}_{thermo}	time	Turbine stage power output not including mechanical loss [W]
power_shaft	\dot{w}_{shaft}	time	Turbine stage power output including mechanical loss (bearings...) [W]
specific_speed	n_s	time	Turbine stage specific speed [dimensionless]

$$n_s = s \dot{v}^{0.5} (w_{isen} / \dot{m}) * (-0.75)$$

Where \dot{m} is the mass flow rate and \dot{v} is the outlet volumetric flow.

Constraints

There are no additional constraints.

Initialization

To initialize the turbine model, a reasonable guess for the inlet condition and deltaP and efficiency should be set by setting the appropriate variables.

TurbineStage Class

```
class idaes.power_generation.unit_models.helm.turbine_stage.HelmTurbineStage(*args,  
                                                                           **kwds)
```

Basic steam turbine model

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude

material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.useDefault`. **Valid values:** { **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = `False`. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - `False`. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change

terms.}

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

has_work_transfer True if model a has work transfer term.

has_heat_transfer True if model has a heat transfer term.

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HelmTurbineStage) New instance

TurbineStageData Class

```
class idaes.power_generation.unit_models.helm.turbine_stage.HelmTurbineStageData (component)
```

build()

Add model equations to the unit model. This is called by a default block construction rule when the unit model is created.

initialize (*outlvl=0, solver='ipopt', optarg={'max_iter': 30, 'tol': 1e-06}*)

Initialize the turbine stage model. This deactivates the specialized constraints, then does the isentropic turbine initialization, then reactivates the constraints and solves.

Parameters

- **outlvl** – sets output level of initialization routine
- **solver** (*str*) – Solver to use for initialization
- **optarg** (*dict*) – Solver arguments dictionary

Turbine (Multistage)

This is a composite model for a power plant turbine with high, intermediate and low pressure sections. This model contains an inlet stage with throttle valves for partial arc admission and optional splitters for steam extraction.

The figure below shows the layout of the multistage turbine model. Optional splitters provide for steam extraction. The splitters can have two or more outlets (one being the main steam outlet). The streams that connect one stage to the next can also be omitted. This allows for connecting additional unit models (usually reheaters) between stages.

Fig. 16: MultiStage Turbine Model

Example

This example sets up a turbine multistage turbine model similar to what could be found in a power plant steam cycle. There are 7 high-pressure stages, 14 intermediate-pressure stages, and 11 low-pressure stages. Steam extractions are provided after stages hp4, hp7, ip5, ip14, lp4, lp7, lp9, lp11. The extraction at ip14 uses a splitter with three outlets, one for the main steam, one for the boiler feed pump, and one for a feed-water heater. There is a disconnection between the HP and IP sections so that steam can be sent to a reheater. In this example, a heater block is a stand-in for a reheater model.

```

from pyomo.environ_
↳ import (ConcreteModel,
↳ SolverFactory,
↳ TransformationFactory,
↳
↳ Constraint, value)
from pyomo.network import Arc

from idaes.
↳ core import FlowsheetBlock
from idaes.
↳ unit_models import Heater
from idaes.power_generation.
↳ unit_models.helm import_
↳ HelmTurbineMultistage
from idaes.generic_models.
↳ properties import iapws95

solver_
↳ = SolverFactory('ipopt')
solver.
↳ options = {'tol': 1e-6}

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.
↳ fs.
↳ turb_
↳ =_
↳ HelmTurbineMultistage(default=
↳ {
↳     "property_
↳ package": m.fs.properties,
↳     "num_hp": 7,
↳     "num_ip": 14,
↳     "num_lp": 11,
↳     "hp_
↳ split_locations": [4, 7],
↳     "ip_
↳ split_locations": [5, 14],
↳     "lp_split_
↳ locations": [4, 7, 9, 11],
↳     "hp_disconnect":_
↳ [7], # 7 is last stage in_
↳ hp so disconnect hp from ip
↳     "ip_split_
↳ num_outlets": {14:3}})
# Add reheater_
↳ (for example using_
↳ a simple heater block)
m.fs.reheat =_
↳ Heater(default={"property_
↳ package": m.fs.properties})

```

(continues on next page)

(continued from previous page)

```

# Add Arcs (streams)
↳to connect the HP and IP
↳sections through reheater
m.fs.hp_to_
↳reheat = Arc(source=m.fs.
↳turb.hp_split[7].outlet_1,

↳
↳destination=m.
↳fs.reheat.inlet)
m.fs.reheat_
↳to_ip = Arc(source=m.
↳fs.reheat.outlet,

↳
↳destination=m.
↳fs.turb.ip_stages[1].inlet)
# Set the
↳turbine inlet conditions
↳and an initial flow guess
p = 2.4233e7
hin
↳= iapws95.htpx(T=880, P=p)
m.fs.turb.inlet_split.
↳inlet.enth_mol[0].fix(hin)
m.fs.turb.inlet_split.inlet.
↳flow_mol[0].fix(26000)
m.fs.turb.inlet_split.
↳inlet.pressure[0].fix(p)

# Set the inlet of the ip
↳section for initialization,
↳since it is disconnected
p = 7.802e+06
hin
↳= iapws95.htpx(T=880, P=p)
m.fs.turb.ip_stages[1].inlet.
↳enth_mol[0].value = hin
m.fs.turb.ip_stages[1].inlet.
↳flow_mol[0].value = 25220.0
m.fs.turb.ip_stages[1].
↳inlet.pressure[0].value = p
# Set the efficiency and
↳pressure ratios of stages
↳other than inlet and outlet
for i, s
↳in turb.hp_stages.items():
    s.ratioP[:] = 0.88
    s.entropy_
↳isentropic[:] = 0.9
for i, s
↳in turb.ip_stages.items():
    s.ratioP[:] = 0.85
    s.entropy_
↳isentropic[:] = 0.9
for i, s
↳in turb.lp_stages.items():
    s.ratioP[:] = 0.82
    s.entropy_
↳isentropic[:] = 0.9

```

(continues on next page)

(continued from previous page)

```

# Usually these fractions_
↳ would be determined by the_
↳ boiler feed water heater
# network.
↳ Since this example_
↳ doesn't include them,
↳ just fix split fractions
turb.hp_
↳ split[4].split_fraction[0,
↳ "outlet_2"].fix(0.03)
turb.hp_
↳ split[7].split_fraction[0,
↳ "outlet_2"].fix(0.03)
turb.ip_
↳ split[5].split_fraction[0,
↳ "outlet_2"].fix(0.04)
turb.ip_
↳ split[14].split_fraction[0,
↳ "outlet_2"].fix(0.04)
turb.ip_
↳ split[14].split_fraction[0,
↳ "outlet_3"].fix(0.15)
turb.lp_
↳ split[4].split_fraction[0,
↳ "outlet_2"].fix(0.04)
turb.lp_
↳ split[7].split_fraction[0,
↳ "outlet_2"].fix(0.04)
turb.lp_
↳ split[9].split_fraction[0,
↳ "outlet_2"].fix(0.04)
turb.lp_
↳ split[11].split_fraction[0,
↳ "outlet_2"].fix(0.04)
# unfix inlet flow for_
↳ pressure driven simulation
turb.inlet_split.
↳ inlet.flow_mol.unfix()
# Set the inlet steam mixer_
↳ to use the constraints_
↳ that the pressures of all
# inlet streams are equal
turb.inlet_mix.use_
↳ equal_pressure_constraint()
# Initialize turbine
turb.initialize(outlvl=1)
# Copy_
↳ conditions out of turbine_
↳ to initialize the reheater
for t in m.fs.time:
    m.fs.reheat.
↳ inlet.flow_mol[t].value = \
    ↳ value(turb.hp_split[7].
↳ outlet_1_state[t].flow_mol)
    m.fs.reheat.
↳ inlet.enth_mol[t].value = \

```

(continues on next page)

(continued from previous page)

```

    ↳ value(turb.hp_split[7].
↳outlet_1_state[t].enth_mol)
    m.fs.reheat.
↳inlet.pressure[t].value = \

    ↳ value(turb.hp_split[7].
↳outlet_1_state[t].pressure)
# initialize the reheater
m.fs.
↳reheat.initialize(outlvl=4)
# Add constraint
↳to the reheater to result
↳in 880K outlet temperature
def reheat_T_rule(b, t):
    return m.fs.reheat.
↳control_volume.properties_
↳out[t].temperature == 880
m.fs.reheat.temperature_out_
↳equation = Constraint(m.
↳fs.reheat.time_ref,
    rule=reheat_T_rule)
# Expand the Arcs connecting
↳the turbine to the reheater
TransformationFactory(
↳"network.
↳expand_arcs").apply_to(m)
# Fix the
↳outlet pressure (usually
↳determined by condenser)
m.fs.turb.outlet_stage.
↳control_volume.properties_
↳out[0].pressure.fix()

# Solve the pressure driven
↳flow model with reheat
solver.solve(m, tee=True)

```

Unit Models

The multistage turbine model contains the models in the table below. The splitters for steam extraction are not present if a turbine section contains no steam extractions.

Unit	Index Sets	Doc
inlet_split	None	Splitter to split the main steam feed into steams for each arc (<i>Separator</i>)
throttle_valve	Admission Arcs	Throttle valves for each admission arc (<i>HelmValve</i>)
inlet_stage	Admission Arcs	Parallel inlet turbine stages that represent admission arcs (<i>TurbineInlet</i>)
inlet_mix	None	Mixer to combine the streams from each arc back to one stream (<i>Mixer</i>)
hp_stages	HP stages	Turbine stages in the high-pressure section (<i>TurbineStage</i>)
ip_stages	IP stages	Turbine stages in the intermediate-pressure section (<i>TurbineStage</i>)
lp_stages	LP stages	Turbine stages in the low-pressure section (<i>TurbineStage</i>)
hp_splits	subset of HP stages	Extraction splitters in the high-pressure section (<i>Separator</i>)
ip_splits	subset of IP stages	Extraction splitters in the high-pressure section (<i>Separator</i>)
lp_splits	subset of LP stages	Extraction splitters in the high-pressure section (<i>Separator</i>)
outlet_stage	None	The final stage in the turbine, which calculates exhaust losses (<i>Turbine-Outlet</i>)

Initialization

The initialization approach is to sequentially initialize each sub-unit using the outlet of the previous model. Before initializing the model, the inlet of the turbine, and any stage that is disconnected should be given a reasonable guess. The efficiency and pressure ration of the stages in the HP, IP and LP sections should be specified. For the inlet and outlet stages the flow coefficient should be specified. Valve coefficients should also be specified. A reasonable guess for split fractions should also be given for any extraction splitters present. The most likely cause of initialization failure is flow coefficients in inlet stage, outlet stage, or valves that do not pair well with the specified flow rates.

The flow coefficients for the inlet and outlet stage can be difficult to determine, therefore the initialization arguments `calculate_outlet_cf` and `calculate_outlet_cf` are provided. If these are True, the first stage flow coefficient is calculated from the flow and pressure ratio guesses, and the outlet flow coefficient is calculated from the exhaust pressure and flow.

TurbineMultistage Class

class `idaes.power_generation.unit_models.helm.turbine_multistage.HelmTurbineMultistage` (*args, **kws)
 Multistage steam turbine with optional reheat and extraction

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Only False, in a dynamic flowsheet this is psuedo-steady- state.

has_holdup Only False, in a dynamic flowsheet this is psuedo-steady- state.

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flowsheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

num_parallel_inlet_stages Number of parallel inlet stages to simulate partial arc admission. **Default**=4

throttle_valve_function The type of valve function, if custom provide an expression rule with the valve_function_rule argument. **default** - ValveFunctionType.linear **Valid values** - { ValveFunctionType.linear, ValveFunctionType.quick_opening, ValveFunctionType.equal_percentage, ValveFunctionType.custom }

throttle_valve_function_callback A callback to add a custom valve function

to the throttle valves or None. If a callback is provided, it should take the valve block data as an argument and add a valve_function expressions to it. Default=None

num_hp Number of high pressure stages not including inlet stage

num_ip Number of intermediate pressure stages

num_lp Number of low pressure stages not including outlet stage

hp_split_locations A list of index locations of splitters in the HP section. The indexes indicate after which stage to include splitters. 0 is between the inlet stage and the first regular HP stage.

ip_split_locations A list of index locations of splitters in the IP section. The indexes indicate after which stage to include splitters.

lp_split_locations A list of index locations of splitters in the LP section. The indexes indicate after which stage to include splitters.

hp_disconnect HP Turbine stages to not connect to next with an arc. This is usually used to insert additional units between stages on a flowsheet, such as a reheater

ip_disconnect IP Turbine stages to not connect to next with an arc. This is usually used to insert additional units between stages on a flowsheet, such as a reheater

lp_disconnect LP Turbine stages to not connect to next with an arc. This is usually used to insert additional units between stages on a flowsheet, such as a reheater

hp_split_num_outlets Dict, hp split index: number of splitter outlets, if not 2

ip_split_num_outlets Dict, ip split index: number of splitter outlets, if not 2

lp_split_num_outlets Dict, lp split index: number of splitter outlets, if not 2

- **initialize** (*dict*) – Process-BlockData config for individual

elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.

- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HelmTurbineMultistage) New instance

TurbineMultistageData Class

```
class idaes.power_generation.unit_models.helm.turbine_multistage.HelmTurbineMultistageData (comp
```

build()

General build method for UnitModel-BlockData. This method calls a number of sub-methods which automate the construction of expected attributes of unit models.

Inheriting models should call *super().build*.

Parameters None –

Returns None

```
initialize (outlvl=0, solver='ipopt', flow_iterate=2, optarg={'max_iter': 35, 'tol': 1e-06}, copy_disconneted_flow=True, copy_disconneted_pressure=True, calculate_outlet_cf=False, calculate_inlet_cf=False)
Initialize
```

Parameters

- **outlvl** – logging level default is NOTSET, which inherits from the parent logger
- **solver** – the NL solver, default is “ipopt”
- **flow_iterate** – If not calculating flow coefficients, this is the number of times to update the flow and repeat initialization (1 to 5 where 1 does not update the flow guess)
- **optarg** – solver arguments, default is {“tol”: 1e-6, “max_iter”: 35}
- **copy_disconneted_flow** – Copy the flow through the disconnected stages default is True

- **copy_disconneted_pressure** – Copy the pressure through the disconnected stages default is True
- **calculate_outlet_cf** – Use the flow initial flow guess to calculate the outlet stage flow coefficient, default is False,
- **calculate_inlet_cf** – Use the inlet stage ratioP to calculate the flow coefficient for the inlet stage default is False

Returns None

throttle_cv_fix(*value*)

Fix the throttle valve coefficients. These are generally the same for each of the parallel stages so this provides a convenient way to set them.

Parameters value – The value to fix the turbine inlet flow coefficients at

turbine_inlet_cf_fix(*value*)

Fix the inlet turbine stage flow coefficient. These are generally the same for each of the parallel stages so this provides a convenient way to set them.

Parameters value – The value to fix the turbine inlet flow coefficients at

turbine_outlet_cf_fix(*value*)

Fix the inlet turbine stage flow coefficient. These are generally the same for each of the parallel stages so this provides a convenient way to set them.

Parameters value – The value to fix the turbine inlet flow coefficients at

HelmValve

This is a steam power generation turbine model for the stages between the inlet and outlet.

Example

```

from pyomo.environ_
↳ import ConcreteModel,
↳ SolverFactory,
↳ TransformationFactory

from idaes.
↳ core import FlowsheetBlock
from idaes.power_
↳ generation.unit_models.
↳ helm import HelmValve
from idaes.generic_models.
↳ properties import iapws95
from idaes.ui.
↳ report import degrees_of_
↳ freedom, active_equalities

solver_
↳ = SolverFactory('ipopt')
solver.
↳ options = {'tol': 1e-6}

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()
m.fs.
↳ valve = HelmValve(default=
↳ {"property_
↳ package": m.fs.properties})

hin = iapws95.
↳ htpx(T=880, P=2.4233e7)
# set inlet
m.fs.valve.
↳ inlet enth_mol[0].fix(hin)
m.fs.valve.inlet.
↳ flow_mol[0].fix(26000/4.0)
m.fs.valve.inlet.
↳ pressure[0].fix(2.5e7)
m.fs.valve.Cv.fix(0.01)
m.fs.valve.
↳ valve_opening.fix(0.5)
m.fs.
↳ valve.initialize(outlvl=1)

```

Variables

This model adds a variable to account for mechanical efficiency to the base PressureChanger model.

Variable	Symbol	Index Sets	Doc
Cv	C_v	None	Valve coefficient for liquid [mol/s/Pa ^{0.5}] for vapor [mol/s/Pa]
valve_opening	x	time	The fraction that the valve is open from 0 to 1

Expressions

Currently this model provides two additional expressions, with are not available in the pressure changer model.

Expression	Symbol	Index Sets	Doc
valve_function	$f(x)$	time	This is a valve function that describes how the fraction open affects flow.

Constraints

The pressure flow relation is added to the inherited constraints from the *PressureChanger model*.

If the `phase` option is set to "Liq" the following equation describes the pressure-flow relation.

$$\frac{1}{s_f^2} F^2 = \frac{1}{s_f^2} C_v^2 (P_{in} - P_{out}) f(x)^2$$

If the `phase` option is set to "Vap" the following equation describes the pressure-flow relation.

$$\frac{1}{s_f^2} F^2 = \frac{1}{s_f^2} C_v^2 (P_{in}^2 - P_{out}^2) f(x)^2$$

Initialization

This just calls the initialization routine from PressureChanger, but it is wrapped in a function to ensure the state after initialization is the same as before initialization. The arguments to the initialization method are the same as PressureChanger.

HelmValve Class

```
class idaes.power_generation.unit_models.helm.valve_steam.HelmValve(*args,
                                                                **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **MaterialBalanceType.useDefault** - refer to property package for default balance type ****MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.useDefault. **Valid values:**

{ **EnergyBalanceType.useDefault** - refer to property package for default balance type ****EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_phase_equilibrium Indicates whether terms for phase equilibrium should be constructed, **default** = False. **Valid values:** { **True** - include phase equilibrium terms **False** - exclude phase equilibrium terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flow-sheet, **PropertyParameterObject** - a PropertyParameterBlock object. }

property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:**

{ see property package for documentation.}

has_work_transfer True if model a has work transfer term.

has_heat_transfer True if model has a heat transfer term.

valve_function The type of valve function, if custom provide an expression rule with the `valve_function_rule` argument. **default** - `ValveFunctionType.linear` **Valid values** - {`ValveFunctionType.linear`, `ValveFunctionType.quick_opening`, `ValveFunctionType.equal_percentage`, `ValveFunctionType.custom`}

valve_function_callback This is a callback that adds a valve function. The callback function takes the valve block data argument.

phase Expected phase of fluid in valve in {"Liq", "Vap"}

- **initialize** (*dict*) - Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the "default" argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (`HelmValve`) New instance

HelmValveData Class

class `idaes.power_generation.unit_models.helm.valve_steam.HelmValveData` (*component*)

Basic adiabatic OD valve model. This inherits the balance block to get a lot of unit model boilerplate and the mass balance, energy balance and pressure equations. This model is intended to be used only with Helmholtz EOS property packages in mixed or single phase mode with P-H state vars.

Since this inherits `BalanceBlockData`, and only operates in steady-state or

pseudo-steady-state (for dynamic models) the following mass, energy and pressure equations are implicitly written.

- 1) **Mass Balance:** $0 = \text{flow_mol_in}[t] - \text{flow_mol_out}[t]$
- 2) **Energy Balance:** $0 = (\text{flow_mol}[t] * h_{\text{mol}}[t])_{\text{in}} - (\text{flow_mol}[t] * h_{\text{mol}}[t])_{\text{out}}$
- 3) **Pressure:** $0 = P_{\text{in}}[t] + \text{deltaP}[t] - P_{\text{out}}[t]$

build()

Add model equations to the unit model. This is called by a default block construction rule when the unit model is created.

initialize (*outlvl=0, solver='ipopt', optarg={'tol': 1e-06}*)

For simplicity this initialization requires you to set values for the efficiency, inlet, and one of pressure ratio, pressure change or outlet pressure.

BoilerHeatExchanger

The BoilerHeatExchanger model can be used to represent boiler heat exchangers in sub-critical and super critical power plant flowsheets (i.e. economizer, primary superheater, secondary superheater, finishing superheater, reheater, etc.). The model consists of a shell and tube crossflow heat exchanger, in which the shell is used as the gas side and the tube is used as the water or steam side. Rigorous heat transfer calculations (convective heat transfer for shell side, and convective heat transfer for tube side) and shell and tube pressure drop calculations have been included.

The BoilerHeatExchanger model can be imported from `idaes.power_generation.unit_models`, while additional rules and utility functions can be imported from `idaes.power_generation.unit_models.boiler_heat_exchanger`.

Example

The example below demonstrates how to initialize the BoilerHeatExchanger model, and override the default temperature difference calculation.

```
# Import Pyomo libraries
from pyomo.environ_
↳ import ConcreteModel,
↳ SolverFactory, value
# Import IDAES core
from idaes_
↳ core import FlowsheetBlock
# Import Unit Model Modules
from idaes.generic_models_
↳ properties import iapws95
# import_
↳ ideal flue gas prop pack
from idaes.power_
↳ generation.properties_
↳ IdealProp_FlueGas import_
↳ FlueGasParameterBlock
# Import_
↳ Power Plant HX Unit Model
from idaes.power_
↳ generation.unit_models_
↳ boiler_heat_exchanger_
↳ import BoilerHeatExchanger,
↳ TubeArrangement, \
    DeltaTMethod
import pyomo.environ_
↳ as pe # Pyomo environment
from idaes.core import_
↳ FlowsheetBlock, StateBlock
from idaes.unit_models.heat_
↳ exchanger import delta_
↳ temperature_amtd_callback
from idaes.generic_models_
↳ properties import iapws95

# Create a Concrete Model_
↳ as the top level object
m = ConcreteModel()

# Add a flowsheet_
↳ object to the model
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})

# Add property packages_
↳ to flowsheet library
m.fs.prop_water = iapws95.
↳ Iapws95ParameterBlock()
m.fs.prop_fluegas_
↳ = FlueGasParameterBlock()
```

(continues on next page)

(continued from previous page)

```

# Create unit models
m.
↳ fs.
↳ ECON_
↳ =
↳ BoilerHeatExchanger(default=
↳
↳     {"side_1_property_
↳ package": m.fs.prop_water,
↳
↳     "side_2_property_package
↳ ": m.fs.prop_fluegas,
↳
↳     "has_
↳ pressure_change": True,
↳
↳     "has_holdup": False,
↳
↳     "delta_
↳ T_method": DeltaTMethod.
↳ counterCurrent,
↳
↳     "tube_arrangement
↳ ": TubeArrangement.inLine,
↳
↳     "side_
↳ 1_water_phase": "Liq",
↳
↳     "has_radiation": False})

# Set Inputs
# BFW Boiler Feed_
↳ Water inlet temeperature_
↳ = 555 F = 563.706 K
# inputs_
↳ based on NETL Baseline_
↳ Report v3 (SCPC 650 MW net,
↳ no carbon capture case)
h = iapws95.
↳ httpx(563.706, 2.5449e7)
m.fs.ECON.
↳ side_1_inlet.flow_mol[0].
↳ fix(24678.26) # mol/s
m.fs.ECON.side_
↳ 1_inlet.enth_mol[0].fix(h)
m.fs.ECON.
↳ side_1_inlet.pressure[0].
↳ fix(2.5449e7) # Pa

# FLUE GAS Inlet_
↳ from Primary Superheater
FGrate = 28.
↳ 3876e3 # mol/s equivalent_
↳ of ~1930.08 klb/hr
# Use FG molar composition_
↳ to set component flow_
↳ rates (baseline report)

```

(continues on next page)

(continued from previous page)

```

m.fs.ECON.side_
  ↳ 2_inlet.flow_component[0,
  ↳ "H2O"].fix(FGrate*8.69/100)
m.fs.ECON.side_2_
  ↳ inlet.flow_component[0,"CO2
  ↳ ].fix(FGrate*14.49/100)
m.fs.ECON.side_
  ↳ 2_inlet.flow_component[0,
  ↳ "N2"].fix(FGrate*(8.69
  ↳
  ↳
  ↳ +14.49+2.47+0.06+0.2)/100)
m.fs.ECON.side_
  ↳ 2_inlet.flow_component[0,
  ↳ "O2"].fix(FGrate*2.47/100)
m.fs.ECON.side_
  ↳ 2_inlet.flow_component[0,
  ↳ "NO"].fix(FGrate*0.0006)
m.fs.ECON.side_
  ↳ 2_inlet.flow_component[0,
  ↳ "SO2"].fix(FGrate*0.002)
m.fs.ECON.side_
  ↳ 2_inlet.temperature[0].
  ↳ fix(682.335) # K
m.fs.ECON.
  ↳ side_2_inlet.pressure[0].
  ↳ fix(100145) # Pa
# economizer design
↳ variables and parameters
ITM = 0.0254
↳ # inch to meter conversion
# Based on
↳ NETL Baseline Report Rev3
m.fs.ECON.tube_
  ↳ di.fix((2-2*0.188)*ITM)
  ↳ # calc inner diameter
#
  ↳
  ↳ (2 = outer diameter,
  ↳ thickness = 0.188)
m.fs.
  ↳ ECON.tube_thickness.fix(0.
  ↳ 188*ITM) # tube thickness
m.fs.
  ↳ ECON.pitch_x.fix(3.5*ITM)
# pitch_y = (54.5) gas path
↳ transverse width /columns
m.fs.
  ↳ ECON.pitch_y.fix(5.03*ITM)
m.fs.ECON.tube_
  ↳ length.fix(53.41*12*ITM) #
  ↳ use tube length (53.41 ft)
m.fs.ECON.tube_nrow.fix(36*2.
  ↳ 5) # use to
  ↳ match baseline performance
m.fs.ECON.
  ↳ tube_ncol.fix(130)
  ↳ # 130 from NETL report

```

(continues on next page)

(continued from previous page)

```

m.fs.ECON.nrow_inlet.fix(2)
m.fs.ECON.
↳delta_elevation.fix(50)
# parameters
# heat transfer_
↳resistance due to tube_
↳side fouling (water scales)
m.fs.ECON.
↳tube_rfouling = 0.000176
# heat transfer resistance_
↳due to tube shell_
↳fouling (ash deposition)
m.fs.ECON.
↳shell_rfouling = 0.00088
if m.fs.ECON.config.
↳has_radiation is True:
    m.fs.
↳ECON.emissivity_wall.fix(0.
↳7) # wall emissivity
# correction_
↳factor for overall_
↳heat transfer coefficient
m.fs.ECON.
↳fcorrection_htc.fix(1.5)
# correction_
↳factor for pressure_
↳drop calc tube side
m.fs.ECON.fcorrection_
↳dp_tube.fix(1.0)
# correction_
↳factor for pressure_
↳drop calc shell side
m.fs.ECON.fcorrection_
↳dp_shell.fix(1.0)

# Initialize the model
m.fs.ECON.initialize()

```

Degrees of Freedom

Aside from the inlet conditions, a heat exchanger model usually has two degrees of freedom, which can be fixed for it to be fully specified. Things that are frequently fixed are two of:

- heat transfer area,
- heat transfer coefficient, or
- temperature approach.

In order to capture off design conditions and heat transfer coefficients at ramp up/down or load following conditions, the BoilerHeatExanger model

includes rigorous heat transfer calculations. Therefore, additional degrees of freedom are required to calculate Nusselt, Prandtl, Reynolds numbers, such as:

- tube_di (inner diameter)
- tube length
- tube number of rows (tube_nrow), columns (tube_ncol), and inlet flow (nrow_inlet)
- pitch in x and y axis (pitch_x and pitch_y, respectively)

If pressure drop calculation is enabled, additional degrees of freedom are required:

- elevation with respect to ground level (delta_elevation)
- tube fouling resistance (tube_r_fouling)
- shell fouling resistance (shell_r_fouling)

Model Structure

The `BoilerHeatExchanger` model contains two `ControlVolume0DBlock` blocks. By default the gas side is named `shell` and the water/steam side is named `tube`. These names are configurable. The sign convention is that duty is positive for heat flowing from the hot side to the cold side.

The control volumes are configured the same as the `ControlVolume0DBlock` in the *Heater model*. The `BoilerHeatExchanger` model contains additional constraints that calculate the amount of heat transferred from the hot side to the cold side.

The `BoilerHeatExchanger` has two inlet ports and two outlet ports. By default these are `shell_inlet`, `tube_inlet`, `shell_outlet`, and `tube_outlet`. If the user supplies different hot and cold side names the inlet and outlets are named accordingly.

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	time	Heat transferred from hot side to the cold side
area	A	None	Heat transfer area
U	U	time	Heat transfer coefficient
delta_temperature	ΔT	time	Temperature difference, defaults to LMTD

Note: `delta_temperature` may be either a variable or expression depending on the callback used. If the specified cold side is hotter than the specified hot side this value will be negative.

Constraints

The default constraints can be overridden by providing *alternative rules* for the heat transfer equation, temperature difference, heat transfer coefficient, shell and tube pressure drop. This section describes the default constraints.

Heat transfer from shell to tube:

$$Q = U A \Delta T$$

Temperature difference is:

$$\Delta T = \frac{\Delta T_1 - \Delta T_2}{\log_e \left(\frac{\Delta T_1}{\Delta T_2} \right)}$$

The overall heat transfer coefficient is calculated as a function of convective heat transfer shell and tube, and wall conduction heat transfer resistance.

Convective heat transfer equations:

$$\frac{1}{U} * fcorrection_{htc} = \left[\frac{1}{hconv_{tube}} + \frac{1}{hconv_{shell}} + r + tube_{rfouling} + shell_{rfouling} \right]$$

$$hconv_{tube} = \frac{Nu_{tube} k}{tube_{di}}$$

$$Nu_{tube} = 0.023 Re_{tube}^{0.8} Pr_{tube}^{0.4}$$

$$Pr_{tube} = \frac{Cp \mu}{k M w}$$

$$Re_{tube} = \frac{tube_{di} V \rho}{\mu}$$

$$hconv_{shell} = \frac{Nu_{shell} k_{fluegas}}{tube_{do}}$$

$$Nu_{shell} = f_{arrangement} 0.33 Re_{tube}^{0.6} Pr_{tube}^{0.3333}$$

$$Pr_{shell} = \frac{Cp\mu}{kMw}$$

$$Re_{shell} = \frac{tube_{do} V \rho}{\mu}$$

$$tube_{do} = 2 * tube_{thickness} + tube_{di}$$

Wall heat conduction resistance equation:

$$r = 0.5 * tube_{do} * \log\left(\frac{tube_{do}}{tube_{di}}\right) * k$$

where:

- hconv_tube : convective heat transfer resistance tube side (fluid water/steam) (W / m² / K)
- hconv_shell : convective heat transfer resistance shell side (fluid Flue Gas) (W / m² / K)
- Nu : Nusselt number
- Pr : Prandtl number
- Re : Reynolds number
- V: velocity (m/s)
- tube_di : inner diameter of the tube (m)
- tube_do : outer diameter of the tube (m) (expression calculated by the model)
- tube_thickness : tube thickness (m)
- r = wall heat conduction resistance (K m² / W)
- k : thermal conductivity of the tube wall (W / m / K)
- ρ : density (kg/m³)
- μ : viscosity (kg/m/s)
- tube_r_fouling : tube side fouling resistance (K m² / W)
- shell_r_fouling : shell side fouling resistance (K m² / W)
- fcorrection_htc: correction factor for overall heat transfer
- f_arrangement: tube arrangement factor

Note: by default fcorrection_htc is set to 1, however, this variable can be used to match unit performance (i.e. as a parameter estimation problem using real plant data).

Tube arrangement factor is a config argument with two different type of arrangements supported at the moment: 1.- In-line tube arrangement factor ($f_{\text{arrangement}} = 0.788$), and 2.- Staggered tube arrangement factor ($f_{\text{arrangement}} = 1$). $f_{\text{arrangement}}$ is a parameter that can be adjusted by the user.

The `BoilerHeatExchanger` includes an argument to compute heat transfer due to radiation of the flue gases. If `has_radiation = True` the model builds additional heat transfer calculations that will be added to the `hconv_shell` resistances. Radiation effects are calculated based on the gas gray fraction and gas-surface radiation (between gas and shell).

$$Gas_{grayfrac} = f(gas_{emissivity})$$

$$frad_{gasgrayfrac} = f(wall_{emissivity}, gas_{emissivity})$$

$$hconv_{shell,rad} = f(k_{boltzmann}, frad_{gasgrayfrac}, T_{gasin}, T_{gasout}, T_{fluidin}, T_{fluidout})$$

Note: Gas emissivity is calculated with surrogate models (see more details in `boiler_heat_exchanger.py`). Radiation = True when flue gas temperatures are higher than 700 K (for example, when the model is used for units like Primary superheater, Reheater, or Finishing Superheater; while Radiation = False when the model is used to represent the economizer in a power plant flowsheet).

If pressure change is set to True, ΔP_{turn} and $friction_{factor}$ are calculated

Tube side:

$$\Delta P_{tube} = \Delta P_{tube_{friction}} + \Delta P_{tube_{turn}} - elevation * g * \frac{\rho_{in} + \rho_{out}}{2}$$

$$\Delta P_{tube_{friction}} = f(tube_{di}, \rho, V_{tube}, number_{of tubes}, tube_{length})$$

$$\Delta P_{tube_{turn}} = f(\rho, v_{tube}, k_{loss_{turn}})$$

where:

- $k_{loss_{turn}}$: pressure loss coefficient of a tube u-turn
- g : is the acceleration of gravity 9.807 (m/s²)

Shell side:

$$\Delta P_{shell} = 1.4 \Delta P_{shell_{friction}} \rho V_{shell}^2$$

$\Delta P_{shellfriction}$ is calculated based on the tube arrangement type:

$$\text{In-line: } \Delta P_{shellfriction} = 0.044 + \frac{0.08 \left(\frac{P_x}{tube_{do}} \right)}{0.43 + \frac{1.13}{\left(\frac{P_y}{tube_{do}} - 1 \right)}} \frac{1}{Re^{0.15}}$$

$$\text{Staggered: } \Delta P_{shellfriction} = 0.25 + \frac{0.118}{\left(\frac{P_y}{tube_{do}} - 1 \right)^{1.08}} \frac{1}{Re^{0.16}}$$

Figure. Tube Arrangement

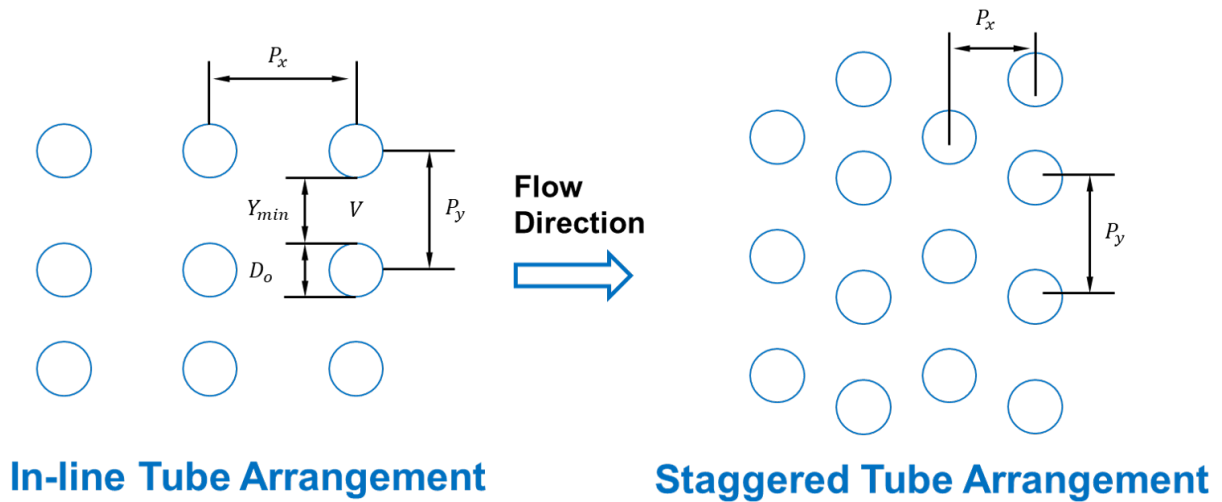


Fig. 17: Tube Arrangement

Class Documentation

Note: The `hot_side_config` and `cold_side_config` can also be supplied using the name of the hot and cold sides (`shell` and `tube` by default) as in [the example](#).

```
class idaes.power_generation.unit_models.boiler_heat_exchanger.BoilerHeatExchanger(*args,
**kwds)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **True** - construct holdup terms, **False** - do not construct holdup terms }

side_1_property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flowsheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

side_1_property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

side_2_property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flowsheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

side_2_property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

material_balance_type Indicates what type of material balance should be constructed, **default** - MaterialBalanceType.componentPhase. **Valid values:** { **MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use

total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - **EnergyBalanceType.enthalpyTotal**. **Valid values:** { **EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - **MomentumBalanceType.pressureTotal**. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - **False**. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

delta_T_method Flag indicating type of flow arrangement to use for delta **default** - **DeltaTMethod.counterCurrent** **Valid values:** { **DeltaTMethod.counterCurrent** }

tube_arrangement Tube arrangement could be in-line and staggered

side_1_water_phase Define water phase for property calls

has_radiation Define if side 2 gas radiation is to be considered

- **initialize** (*dict*) – Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the “default” argument above.
- **idx_map** (*function*) – Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (BoilerHeatExchanger) New instance

class `idaes.power_generation.unit_models.boiler_heat_exchanger.BoilerHeatExchangerData` (*component*
Standard Heat Exchanger Unit Model
Class

build()
Build method for Boiler heat exchanger model

Parameters None –

Returns None

initialize (*state_args_1*={}, *state_args_2*={}, *outlvl*=0, *solver*='ipopt', *optarg*={'max_iter': 100, 'tol': 1e-06})
General Heat Exchanger initialisation routine.

Keyword Arguments

- **state_args_1** – a dict of arguments to be passed to the property package(s) for side 1 of the heat exchanger to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **state_args_2** – a dict of arguments to be passed to the property package(s) for side 2 of the heat exchanger to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
 - 0 = no output (default)
 - 1 = return solver state for each step in routine
 - 2 = return solver state for each step in subroutines

- 3 = include solver output information (tee=True)
- **optarg** – solver options dictionary object (default={ ‘tol’: 1e-6})
- **solver** – str indicating which solver to use during initialization (default = ‘ipopt’)

Returns None

model_check()

Model checks for unit - calls model checks for both control volume Blocks.

Parameters None –

Returns None

WaterWall Model

Introduction

The waterwall section model simulates the water hydraulics and the heat transfer inside typical membrane waterwall tubes. The fluid flowing inside the tubes is either liquid water or a mixture of liquid water and steam (two-phase flow). A boiler is typically discretized in multiple zones or sections along its height and a waterwall section model represents one section of the waterwall. It is usually coupled with IDAES’ 1-D fire-side model to solve the wall temperatures and heat transfer rate in each section. Figure 1 shows a schematic representation of the integrated boiler fire-side and fluid-side models, in which the sum of the net radiation and convective heat fluxes (q_{rad}^{fire} and q_{conv}^{fire}) at the slag outer layer is an output of the fire-side model and an input of the waterwall section model (the fluid-side model) and the temperature of the outer slag layer $T_{w,slag}$ is an output of the fluid-side model and an input (boundary condition) of the fire-side model. The heat conduction through the slag and tube layers is a part of the fluid-side model. At a steady state, the amount of the heat transferred at the outer slag surface (q_{rad}^{fire} and q_{conv}^{fire}) is equal to the heat conducted through the slag and tube layers, which is equal to the heat convected to the fluid q_{conv}^{fluid} .

Property package: This model requires the Helmholtz EoS (IAPWS95) property package with the mixed phase option, therefore, the phase equilibrium

calculations are handled by the property package.

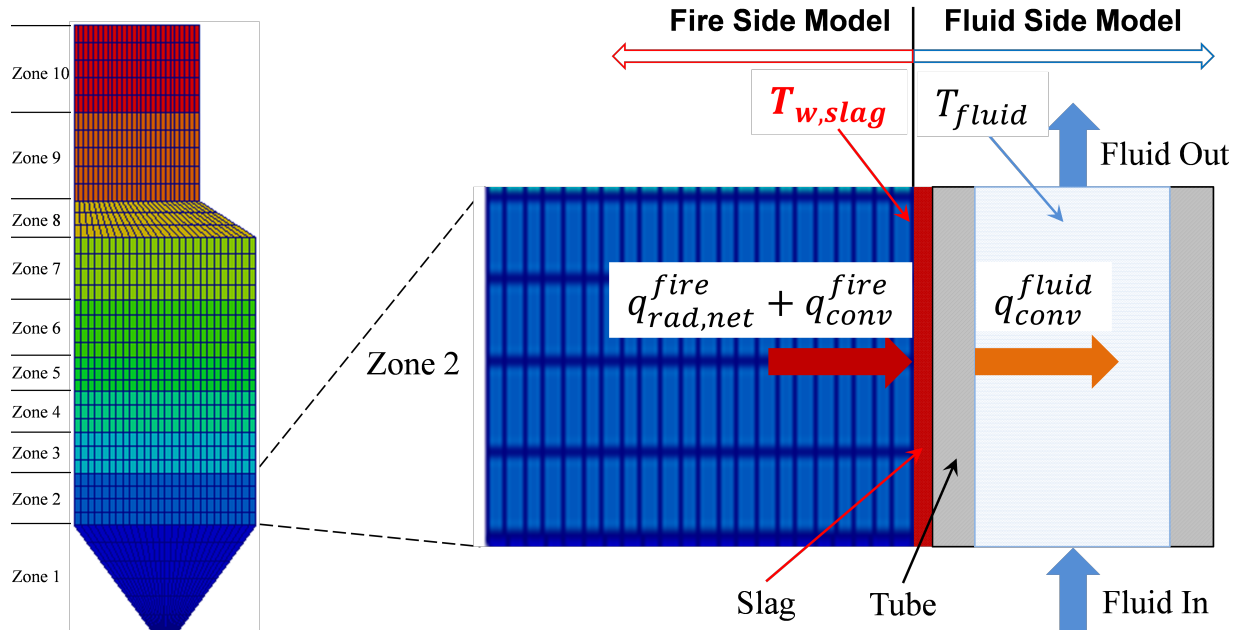


Fig. 18: Figure 1. Coupling of fire-side zones and fluid-side waterwall sections modeled in IDAES

Figure 1 shows the schematic representation of the coupled fire-side and fluid-side waterwall model. Since, the waterwall section is connected with a fire-side section of the boiler, the user must set the number of waterwall sections equal to the number of zones in the fire-side model. The first (the lowest) section is generally connected with the water liquid from a downcomer for a subcritical boiler or the water from the outlet of an economizer for a supercritical boiler. Other sections are generally connected with their neighboring sections below them via Pyomo Arcs. Finally, the last section is generally connected either to a drum for a subcritical boiler or a superheater for a supercritical boiler.

Model inputs (variable name):

- number of zones (ww_zones)
- number of tubes around the perimeter of the boiler (number_tubes)
- heat duty of individual zone from fire-side model (sum of net radiation and convection) (heat_fireside)

- tube dimensions (length, inside diameter and thickness) (tube_length, tube_diameter, tube_thickness)
- projected membrane wall area (projected_area)
- fin dimension of membrane wall (width and thickness) (fin_length, fin_thickness)
- slag layer thickness (slag_thickness)
- water/steam flow rate and states at inlet (flow_mol, enth_mol, pressure)
- properties of slag and tube metal (thermal conductivity, heat capacity, density) (therm_cond_slag, therm_cond_metal, dens_metal, dens_slag)
- pressure drop correction factor (fcorrection_dp)

Model Outputs:

- temperatures of tube metal at inner wetted surface and at center of the tube thickness (temp_tube_boundary, temp_tube_center)
- temperatures of slag layer at outer surface and at the center of the slag layer (temp_slag_boundary, temp_slag_center)
- pressure drop through each section and heat added to each section (deltaP)
- water/steam flow rate and states at outlet (flow_mol, enth_mol, pressure)

Figure 2 illustrates the physics and main variables in a single waterwall section model. This model assumes that the net radiation and convective heat fluxes are given from the fire-side model for the corresponding zone. The membrane wall geometry and slag layer thickness are the given input variables along with the fluid inlet flow rate and state conditions. In case of 2-phase flow, the volume fraction of the vapor phase is calculated based on an empirical correlation that calculates the slip velocity between the two phases due to their density difference. The pressure drop of the 2-phase flow is calculated based on the liquid-only velocity, Reynolds number and friction factors corrected for the

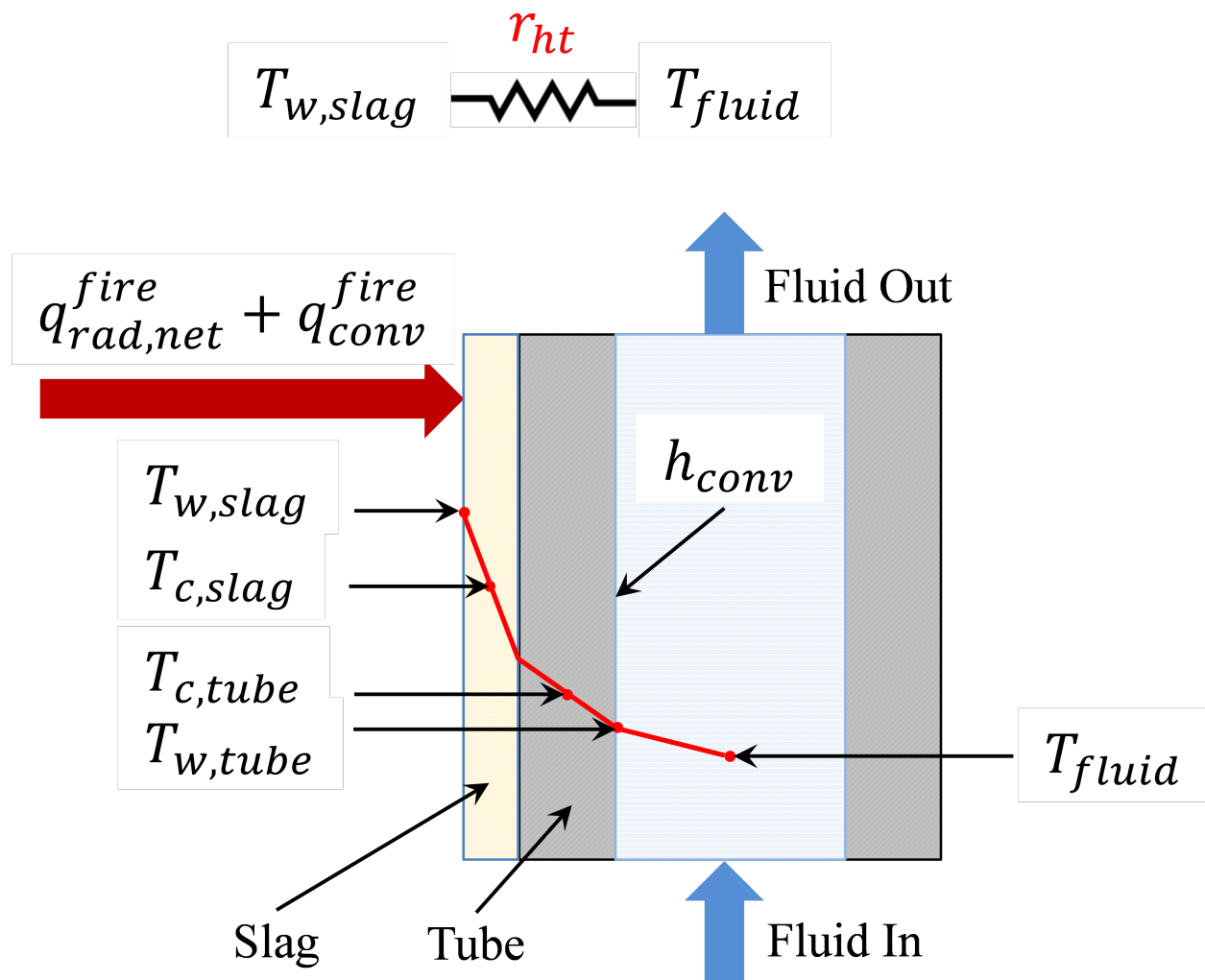


Fig. 19: Figure 2. Illustration of a waterwall section model and its main variables

volume fraction of vapor. Likewise, the convective heat transfer coefficient h_{conv} on the fluid side is calculated from empirical correlations for nucleate boiling with forced convection enhancement factor and pool boiling suppression factor. The overall heat transfer coefficient is the reciprocal of the overall heat transfer resistance (r_{ht} in Figure 2). Finally, The heat duty from fire-side model is provided by the user, while the heat conduction model solves the slag outer surface temperature $T_{w,slag}$, slag layer center point temperature $T_{c,slag}$, tube center point temperature $T_{c,tube}$, and tube inner wall temperature $T_{w,tube}$. The center point temperatures are used to calculate the energy stored in the slag and tube layers for dynamic simulations. The single waterwall section model eventually calculates the heat transfer rate to the fluid, pressure drop of the fluid in the waterwall section, and the slag outer wall temperature, which is required as the boundary condition input for the fire-side model. The heat duty from fire-side model is generally obtained from a surrogate model, the surrogate model must be trained using a rigorous 2-D model under different operating conditions. The rigorous 2-D simulates the heat conduction through the complicated geometry of the slag and tube layers and heat convection between the inner tube wall and the fluid. [1]

[1] Ma, J., Eason, J. P., Dowling, A. W., Biegler, L. T., Miller, D. C. (2016). Development of a first-principles hybrid boiler model for oxy-combustion power generation system. Int. J. of Greenhouse Gas Contr., 46, pp. 136-157.

Degrees of Freedom

As mentioned above, the water wall section model has been modeled as an index set block, therefore, the number of zones must be selected during the construction of this model. Each waterwall section is then considered a single model. Aside from the inlet conditions and tube dimensions, a waterwall section model usually has two degrees of freedom, which can be fixed for it to be fully specified. Things that are frequently fixed are two of:

- tube dimensions and number of tubes,
- heat loss to the water wall,
- ash or slag thickness

Variables

Variable	Sym- bol	Index Sets	Doc
heat_duty	Q	time	Heat transferred from flue gas to tube side fluid
projected_area	A	None	Heat transfer area (total projected area based on tube shape)
hconv, hconv_liquid	h_{conv}	time	Overall convective heat transfer coefficient and hconv_liquid for liquid only
temp_slag_boundary	$T_{w,slag}$	time	Temperature of the slag

Constraints

The main constraints here show the heat flux and convective heat transfer model. This model calculates the slag temperature, slag center temperature, tube boundary temperature, tube center temperatures, and heat flux from fire side to the water/steam side. Finally, a two phase flow model is considered, including water boiling effects in the convective heat transfer coefficient calculations.

Heat flux equation:

$$heat_{flux} = Q * pitch / (projected_{area} * perimeter_{slag})$$

Temperature of slag:

$$T_{w,slag} - T_{c,slag} = heat_{flux} * slag_{resistance}$$

Heat flux interface equation:

$$heat_{flux_{int}} * (slag_{resistance} + metal_{resistance}) = (T_{c,slag} - T_{c,tube})$$

Convective heat flux eqn at tube boundary:

$$heat_{flux_{conv}} * f_{shape_{conv}} * tube_{perimeter} = pitch * h_{conv} * (T_{w,tube} - T_{fluid,in})$$

Tube boundary wall temperature:

$$heat_{flux_{conv}} * metal_{resistance} * tube_{perimeter} = interface_{perimeter} * (T_{c,tube} - T_{w,tube})$$

Heat equation:

$$heat_{duty} = number_{tubes} * heat_{flux_{conv}} * tube_{length} * tube_{perimeter}$$

Convective heat transfer:

$$h_{conv} = h_{convective_{lo}} * enhancement_{factor} + h_{pool} * suppression_{factor}$$

Pressure drop:

$$\Delta P = \Delta P_{friction} + \Delta P_{gravity}$$

Convective heat transfer liquid only:

$$h_{conv_{lo}} = f(tube_{diameter}, N_{Re}, N_{Pr}, k)$$

Enhancement factor:

$$enhancement_{factor} = f(boiling_{number})$$

Pool boiling heat tranfer coefficient:

$$h_{pool} = f(MW, reduced_{pressure}, heat_{flux_{conv}})$$

Prandtl number:

$$Pr_{tube} = \frac{C_p \mu}{k M_w}$$

Reynolds number:

$$Re_{tube} = \frac{tube_{di} V \rho}{\mu}$$

where:

- hconv : convective heat transfer coefficient tube side (fluid water/steam) (W / m² / K)
- hconv_liquid : convective heat transfer coefficient for liquid only
- projected_area : total projected wall area of waterwall section (m²)
- Pr : Prandtl number (liquid only)
- Re : Reynolds number (liquid only)
- V: fluid velocity (m/s, liquid only)
- k : thermal conductivity of the fluid (W / m / K)
- MW: molecular weight of water (kmol/kg)
- μ : viscosity (kg/m/s)

Note that at the flowsheet level first waterwall section is connected to the economizer, arcs connecting section 2 to n-1 have to be constructed by the user, and the outlet of section n is connected to the drum model or superheater (subcritical and supercritical plant, respectively)

Dynamic Model

The dynamic model version of the waterwall section model can be constructed by selecting dynamic=True. If dynamic = True, the energy accumulation of slag and metal, material accumulation holdups are constructed. Therefore, a dynamic initialization method has been developed *set_initial_conditions* to initialize the holdup terms.

HelmPhaseSeparator Model

Introduction

The HelmPhaseSeparator model consists of a simple phase separator to be used only with the Helmholtz equation of state. The two-phase mixture at the inlet is separated into the vapor and liquid streams at the two corresponding outlets. This simple unit includes one state block (mixed_state) for the inlet, and two state blocks, one for liquid (liq_state) and the other for vapor (vap_state). Note that this water-specific flash model replaces IDAES' generic flash unit operation model.

Model inputs:

- mixed_state, variables (flow_mol, enth_mol, and pressure), port name = inlet

Model Outputs:

- liq_state, variables (flow_mol, enth_mol, and pressure), port name = liq_outlet
- vap_state, variables (flow_mol, enth_mol, and pressure), port name = vap_outlet

Degrees of Freedom

The Helm-PhaseSepa-rator model consist of nine vari-ables and six con-straints.

By fix-ing the in-let state (self.inlet.flow_mol,self.inlet.flow_mol, and self.inlet.flow_mol) or three de-grees of freedom, the system will be fully specified.

Variables

Variable	Symbol	Index Sets	Doc
flow_mol	F	time	molar flowrate
enth_mol	h	time	molar enthalpy
pressure	P	time	pressure

Constraints

The phase separator model uses the IAPWS95 property package to calculate the vapor fraction and enthalpies of the vapor and liquid phases at the inlet of the unit. The flowrates of the vap_outlet and liq_outlet streams are calculate as the products of the inlet flow rate and corresponding phase fractions for vapor and liquid, respectively. The enthalpies of the vapor and liquid phases in the inlet stream are assigned to the enthalpies of the vap_outlet and the liq_outlet streams, respectively. The pressure of the two outlet streams are identical to that of the inlet stream.

Material Balances: Vapor State:

$$flow_mol_{mixed_state} * vapor_frac_{mixed_state} = flow_mol_{vap_outlet}$$

Liquid State:

$$flow_mol_{mixed_state} * (1 - vapor_frac_{mixed_state}) = flow_mol_{liq_outlet}$$

Energy Balances:

$$enth_mol_phase_{mixed_state}[Vap] = enth_mol_{vap_state}$$

$$enth_mol_phase_{mixed_state}[Liq] = enth_mol_{liq_state}$$

Momentum Balances:

$$pressure_{mixed_state}[Liq] = pressure_{liq_state} = pressure_{vap_state}$$

Drum Model

Introduction

The drum model consists of three main sub-unit operations:

- 1) a flash model to separate the saturated steam from the saturated liquid water in the water/steam mixture,

- 2) a mixer model to mix saturated liquid water with feed water, and
- 3) a water tank model to calculate drum level and pressure drop.

First the water/steam mixture from boiler waterwall tubes (risers) enters the flash model and leaves in two separate streams (liquid water and steam). Then, the saturated water from the flash model is mixed with the feed water stream (typically from the economizer or a water pipe linking the economizer and the drum) and leave the mixer model in a single mixed stream. Finally, the mixed stream enters the water tank of the drum and leaves the vessel through the multiple downcomers (see Figure 1).

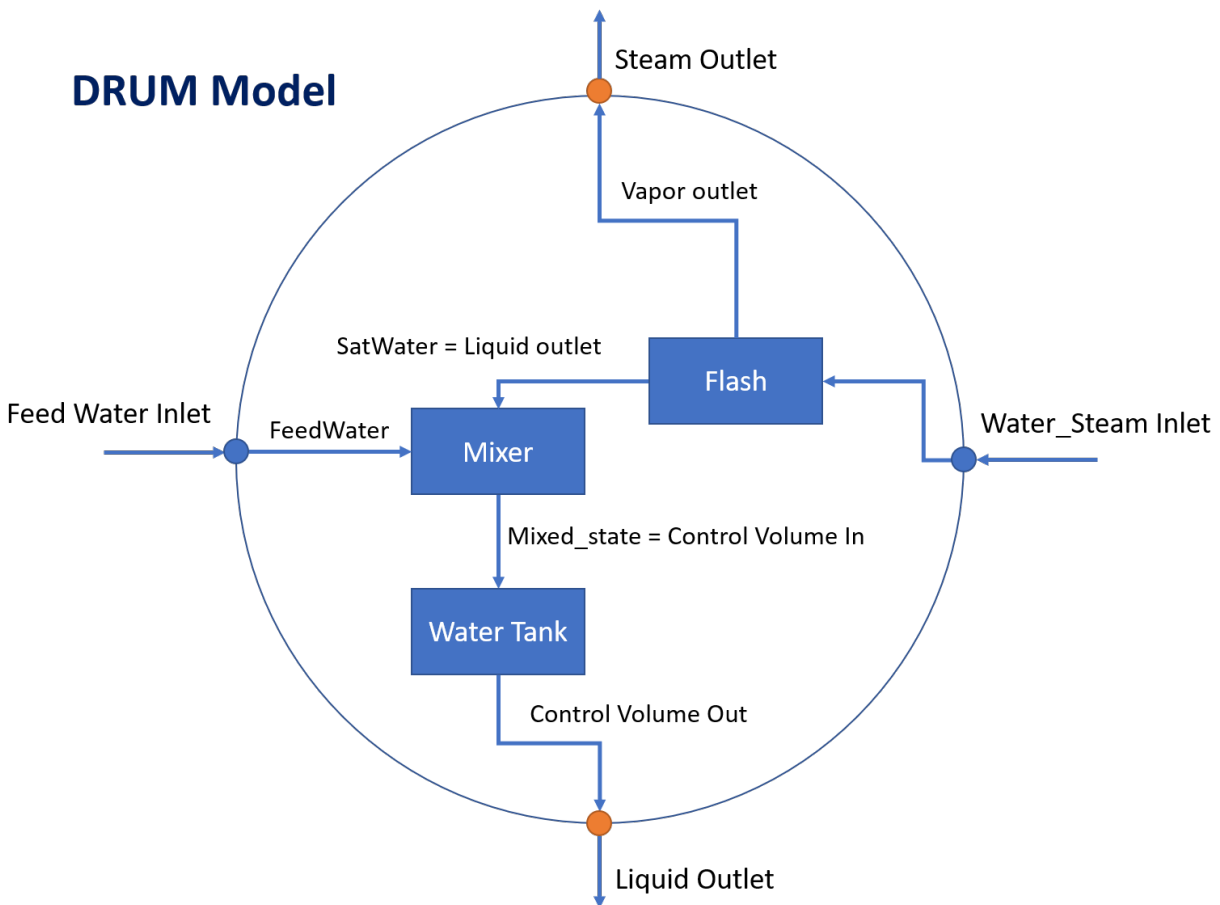


Fig. 20: Figure 1. Schematic representation of a Drum modeled in IDAES

Inlet Ports:

- water_steam_inlet: water/steam mixture from waterwall

- feedwater_inlet: feedwater from economizer/pipe

Outlet Ports:

- liquid_outlet: liquid to downcomer
- steam_outlet: saturated steam leaving the drum

Variables

Model inputs (variable name):

- water/steam inlet (water_steam_inlet: flow_mol, enth_mol, pressure)
- feedwater inlet (feedwater_inlet: flow_mol, enth_mol, pressure)
- drum diameter (drum_diameter)
- drum length (drum_length)
- number of downcomer tubes (number_downcomers)
- downcomer diameter (downcomer_diameter)
- drum level (drum_level)
- heat duty (heat_duty)

Model Outputs:

- vapor outlet (vap_outlet: flow_mol, enth_mol, pressure)
- liquid outlet (liq_outlet: flow_mol, enth_mol, pressure)

Constraints

As mentioned above, the drum model imports a *HelmPhaseSeparator* and mixer models, specific documentation for these models can be obtained in: Once the water enters the tank model the main equations calculate water velocity and pressure drop calculation due to gravity based on water level and contraction to downcomer. Water level (drum_level) is either fixed for steady state simulation or calculated for dynamic model (Dynamic = True)

Main assumptions:

- 1) Heat loss is a variable given by the user (zero heat loss can be specified if adiabatic)
- 2) Pressure change due to gravity based on water level and contraction to downcomer is calculated
- 3) Water level is either fixed for steady-state model or calculated for dynamic model
- 4) Assume $\text{enthalpy_in} == \text{enthalpy_out} + \text{heat loss} + \text{energy accumulation}$
- 5) Subcooled water from economizer and saturated water from waterwall are well mixed before entering the drum

Pressure equality constraint:

$$P_{SaturatedWater} = P_{FeedWater}$$

Pressure drop in unit:

$$\text{delta}P = \text{delta}P_{contraction} + \text{delta}P_{gravity}$$

$$\text{delta}P_{gravity} = f(\rho_{liquid}, \text{accelerationgravity}, \text{drum_level})$$

$$\text{delta}P_{contraction} = f(\rho_{liquid}, V)$$

where: * V: fluid velocity (m/s, liquid only)

Note that the model builds an Pyomo Arc to connect the Liquid_outlet from the self.aFlash unit to the SaturatedWater inlet port of the mixer, and the mixed_state (Mixer outlet) is directly constructed as the Drum *control_volume.properties_in*. Once the Drum model is constructed, the mixer and flash blocks can be found as *self.aDrum.aMixer* and *self.aDrum.aFlash*

Degrees of Freedom

Once the unit dimensions have been fixed, the model generally has 5 degrees of freedom. The water/steam mixture inlet state (flow_mol, enth_mol, and pressure) and feewater inlet state (flow_mol and enth_mol). The feedwater inlet pressure is usually free due to the pressure equality mentioned above.

Dynamic Model

The dynamic model version of the drum model can be constructed by selecting `dynamic=True`. If `dynamic = True`, material accumulation, energy accumulation, and drum level must be calculated. Therefore, a dynamic initialization method has been developed *set_initial_conditions* to initialize the holdup terms.

Downcomer Model

Introduction

The Downcomer model consists of a simple pipe model (or a set of pipes) where the inlet stream is the Drum outlet and the outlet stream connects with the WaterWall (section 1). The model simply calculates the pressure change due to friction and gravity, which involves the calculation of fluid velocity, Reynolds number, and friction factor (using Darcy's correlation).

Property package: This model requires the Helmholtz EoS (IAPWS95) property package with the mixed phase option, therefore, the phase equilibrium calculations are handled by the property package.

Model inputs (variable name):

- inlet stream (`flow_mol`, `enth_mol`, `pressure`)
- number of downcomer pipes (`number_downcomers`) same as Drum model
- height of the tubes (`height`)
- inner diameter of the tubes (`diameter`)
- heat duty (`heat_duty`), `heat_duty = 0` if adiabatic

Model Outputs:

- outlet stream (`flow_mol`, `enth_mol`, `pressure`)
- pressure change (`deltaP`) due to gravity and friction

Degrees of Freedom

By specifying the inlet conditions and downcomer dimensions, the model will be fully specified. Things that are frequently fixed are:

- inlet state vars (generally flow_mol, enth_mol, pressure)
- heat_duty to the downcomer (if applicable)
- number_downcomers, height, diameter

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	time	Heat transferred from flue gas to tube side fluid
deltaP	δP	time	Pressure change in the unit

Constraints

The main constraints in the model calculate the pressure drop, which is given by deltaP_friction and deltaP_gravity.

Pressure drop:

$$\Delta P = \Delta P_{friction} + \Delta P_{gravity}$$

Friction:

$$\Delta P_{friction} = f(Friction_{factor}, \rho_{liquid}, V, diameter)$$

Friction factor (Darcy's correlation):

$$Friction_{factor} = \frac{0.3164}{Re^{0.25}}$$

deltaP gravity:

$$\Delta P_{gravity} = \rho_{liquid} * acceleration_{gravity} * height$$

Reynolds number:

$$Re = \frac{tube_{di} V \rho}{\mu}$$

where:

- Re : Reynolds number (liquid)
- V: fluid velocity (m/s, liquid)
- ρ_{liquid} : mass density of liquid (kg/m³)
- μ : viscosity (kg/m/s)

Dynamic Model

The downcomer dynamic model can be constructed by selecting `dynamic=True` and `holdup=True`. If `dynamic` and `holdup = True`, the energy accumulation and material accumulation variables are constructed, which are the derivatives of the corresponding holdup terms with respect to time and are included in the material and energy conservation equations. A dynamic model also requires the specification of initial conditions related to the accumulation variables. The user needs to provide the initial values for the accumulation variables at all time points and fix the initial conditions to solve the dynamic problem. Therefore, a dynamic initialization method has been developed *set_initial_conditions* to initialize the values of the time-indexed accumulation variables to zero and fix the variables at the first time point to zero.

Steam Heater Model

Introduction

The steam heater model consists of a heater model with rigorous heat transfer calculations on the tube side, while the heat duty from fire side is either fixed or provided by the boiler fire side model. The model is usually coupled with the IDAES 1-D fire-side model to solve the wall temperatures and heat transfer rate. If coupled with the fire side model, this model is similar to the `water_wall` section model. The sum of the net radiation and convective heat fluxes (q_{rad}^{fire} and q_{conv}^{fire}) at the slag outer layer is an output of the fire-side model and an input of the steam heater model (the fluid-side model). While, the temperature of the outer slag layer $T_{w,slag}$ is an output of the fluid-side model and an input (boundary condition) of the fire-side model. The heat conduction through the slag and tube layers is a part of the fluid-side model. At a steady state, the amount of the heat transferred at the

outer slag surface (q_{rad}^{fire} and q_{conv}^{fire}) is equal to the heat conducted through the slag and tube layers, which is equal to the heat convected to the fluid q_{conv}^{fluid} .

Model inputs (variable name):

- number of tubes (number_tubes)
- heat duty from fire-side model (sum of net radiation and convection) (heat_fireside)
- tube dimensions (length, inside diameter and thickness) (tube_length, tube_diameter, tube_thickness)
- fin dimension of membrane wall (width and thickness) (fin_length, fin_thickness)
- slag layer thickness (slag_thickness)
- water/steam flow rate and states at inlet (flow_mol, enth_mol, pressure)
- properties of slag and tube metal (thermal conductivity, heat capacity, density) (therm_cond_slag, therm_cond_metal, dens_metal, dens_slag)
- pressure drop correction factor (fcorrection_dp)

Model Outputs:

- temperatures of tube metal at inner wetted surface and at center of the tube thickness (temp_tube_boundary, temp_tube_center)
- temperatures of slag layer at outer surface and at the center of the slag layer (temp_slag_boundary, temp_slag_center)
- pressure drop through each section and heat added to the tube (deltaP and heat_duty, respectively)
- water/steam flow rate and states at outlet (flow_mol, enth_mol, pressure)

Degrees of Freedom

As mentioned above, the steam heater model includes rigorous heat transfer, therefore, detailed tube and unit dimensions are required. Aside from the inlet conditions and tube dimensions, the steam heater model usually has two degrees of freedom, heat flux from fire side and slag thickness, which can be fixed for it to be fully specified.

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	time	Heat transferred from flue gas to tube side fluid
hconv	h_{conv}	time	Overall convective heat transfer coefficient
temp_slag_boundary	$T_{w,slag}$	time	Temperature of the slag
projected_area	A	None	Heat transfer area (total projected area based on tube shape)

Constraints

The main constraints here show the heat flux, convective heat transfer model, and pressure drop. This model calculates the slag temperature, slag center temperature, tube boundary temperature, tube center temperatures, and heat flux from fire side to the water/steam side.

Heat flux equation:

$$heat_{flux} = Q * pitch / (projected_{area} * perimeter_{slag})$$

Temperature of slag:

$$T_{w,slag} - T_{c,slag} = heat_{flux} * slag_{resistance}$$

Heat flux interface equation:

$$heat_{flux_int} * (slag_{resistance} + metal_{resistance}) = (T_{c,slag} - T_{c,tube})$$

Convective heat flux eqn at tube boundary:

$$heat_{flux_conv} * fshape_{conv} * tube_{perimeter} = pitch * h_{conv} * (T_{w,tube} - T_{fluid,in})$$

Tube boundary wall temperature:

$$heat_{flux_conv} * metal_{resistance} * tube_{perimeter} = interface_{perimeter} * (T_{c,tube} - T_{w,tube})$$

Heat equation:

$$heat_{duty} = number_{tubes} * heat_{flux_conv} * tube_{length} * tube_{perimeter}$$

Pressure drop:

$$\Delta P = \Delta P_{friction} + \Delta P_{gravity}$$

Convective heat transfer:

$$h_{conv} = f(tube_{diameter}, N_{Re}, N_{Pr}, k)$$

Prandtl number:

$$Pr_{tube} = \frac{C_p \mu}{k M_w}$$

Reynolds number:

$$Re_{tube} = \frac{tube_{di} V \rho}{\mu}$$

where:

- hconv : convective heat transfer coefficient tube side (fluid water/steam) (W / m² / K)
- projected_area : total projected wall area (m²)
- Pr : Prandtl number
- Re : Reynolds number
- V: fluid velocity (m/s)
- k : thermal conductivity of the fluid (W / m / K)
- MW: molecular weight of water/steam (kmol/kg)

Note that at the flowsheet level first waterwall section is connected to the economizer, arcs connecting section 2 to n-1 have to be constructed by the user, and the outlet of section n is connected to the drum model or superheater (subcritical and supercritical plant, respectively)

Dynamic Model

The dynamic model version of the steam heater model can be constructed by selecting dynamic=True. If dynamic = True, the energy accumulation of slag and metal, material accumulation holdups are constructed. Therefore, a dynamic initialization method has been developed *set_initial_conditions* to initialize the holdup terms.

Boiler Fire Side Model

Introduction

The boiler fire side model consists of a hybrid model, including first principle equations and surrogate models. The surrogate models determine the heat flux to the individual water wall zones, heat flux to platen superheater, heat flux to roof superheater, NO_x formation in PPM, and unburned carbon in fly ash. Meanwhile, the flue gas outlet conditions (flowrate, temperature, and pressure) are determined by complete mass and energy balances.

The processes inside a coal-fired boiler are very complicated involving combustion of the fuel by an oxidizer, typically air, and the transfer of the heat released from the combustion to waterwall, roof, and platen superheater, if any. The reacting flow inside the boiler is turbulent flow with both gas-phase homogenous reactions and gas-solid heterogenous reactions. At high combustion temperatures, the homogenous gas-phase reactions can be assumed to reach chemical equilibrium while the heterogenous reactions involving devolatilization, oxidation of char by O₂, and gasification of the char by H₂O and CO₂ are usually controlled by finite-rate chemistry and mass transfer of the reactants to the external and internal surfaces of the solid fuel. The main heat transfer mechanism inside the boiler is radiative transfer involving both gas phase and solid phase participating media. Due to its complexity, a high-fidelity model such as NETL's 1D/3D hybrid fire-side boiler model should be developed first based on a given geometry of the furnace. A surrogate model could then be generated from the results of multiple high-fidelity model simulations sampled in the input space of the high-fidelity model. Those inputs could possibly include the flow rates of coal, primary and secondary air, lower furnace Stoichiometric ratio, fuel composition, secondary air temperature, and the slag layer wall temperatures of waterwall, roof, and platen superheater. The surrogate model provides the algebraic functions mapping the input variables to the output variables such as heat transfer rates to the waterwall, roof, and platen superheater, unburned carbon in the fly ash, and NO_x mole fraction in the flue gas. As mentioned above, the variables calculated from the surrogate functions include the heat transfer rates to individual waterwall zones, platen superheater, and roof, the unburned carbon in fly ash, and mole fraction of NO in flue gas. The mole fractions of individual species in the flue gas including O₂, N₂, CO₂, H₂O, and SO₂ are calculated based on the mass balance of individual elements including C, H, O, N, and S. Note that Ar in air is ignored here and its mole fraction in air is assigned to N₂. It is also assumed that coal contains C, H, O, N, S elements only (no Cl) and ash in coal is inert (no mineral related reaction is considered). The amount of unburned carbon in fly ash determines the coal burnout (percent of dry-ash-free coal burned). Only the amount of burned coal is considered in calculating the flue gas

composition. Unburned CO in the flue gas is ignored in the current model. To enforce energy balance, the furnace exit gas temperature (FEGT) is not calculated by a surrogate function. It is calculated by the energy balance instead.

Note that the surrogate models are trained off line and imported as a model argument, these surrogate models usually are a function of the coal flowrate, moisture content, stoichiometric ratio (O_2 real/ O_2 reaction), primary air to coal ratio, and wall/slag temperatures among others.

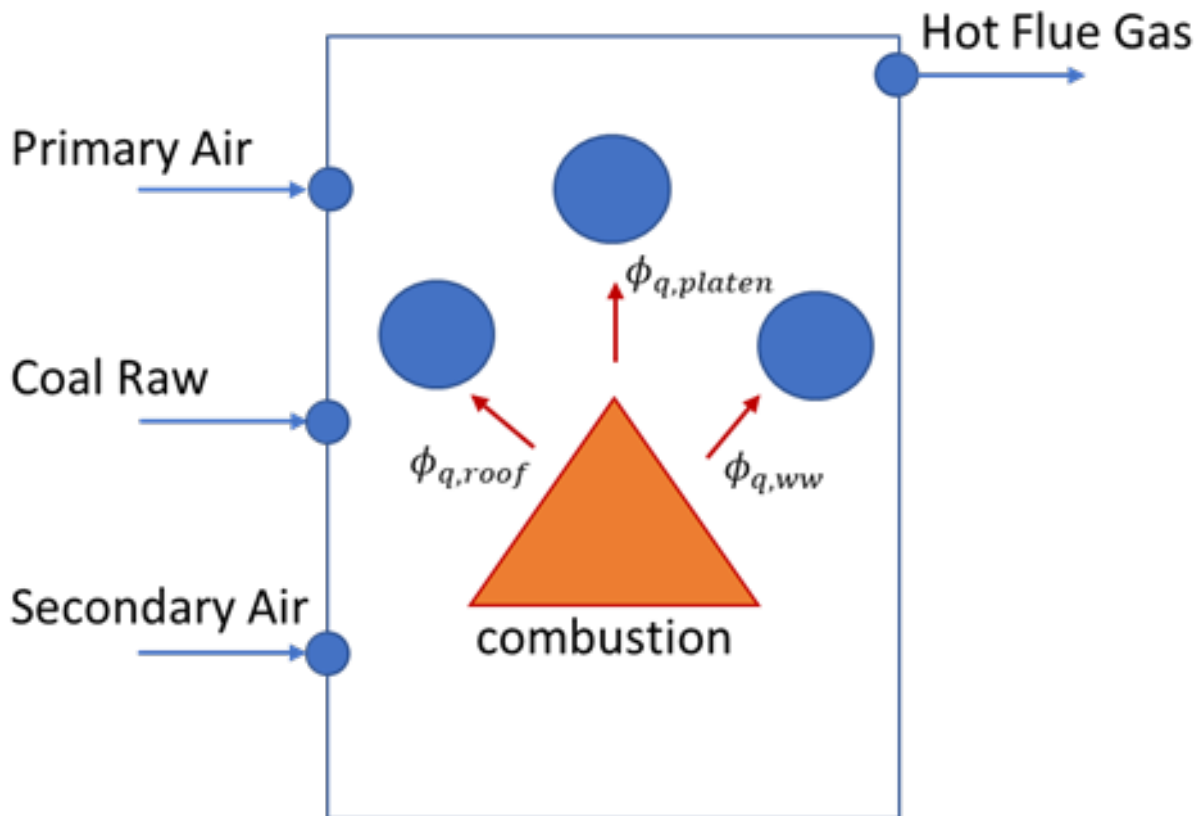


Fig. 21: Figure 1. Schematic representation of a Boiler fire side model

Property package: This model requires the IdealFlueGas property package.

Model Arguments:

- `number_of_zones`: the number of water wall zones are required to maintain the overall energy balance
- `calculate_PA_SA_flows`: Depending on the user's selection, this feature builds different parts of the model (config argument: `calculate_PA_SA_flows=True` or `False`).

- Option1 assumes that users know primary air flowrate and secondary air flowrate (if calculate_PA_SA_flows is False).
- Option2 assumes that the users provide stoichiometric ratio and primary air to coal ratio to calculate primary air and secondary air flowrates (if calculate_PA_SA_flows is True).
- has_platen_superheater: True/False if a platen superheater will be included in the flowsheet
- has_roof_backpass: True if a roof and backpass heater will be included in the flowsheet
- surrogate_dictionary: user must provide a dictionary with surrogate models for water wall zones, platen and roof superheaters, NOx, and flyash

Note that the surrogate dictionary can be either surrogate models (algebraic equations), or fixed values, or variables to calculate the heat flux required for certain performance. For example, this model can be used for data reconciliation to calculate the heat duty to the water wall, platen superheater, and roof.

Model inputs (variable name):

- primary_air_inlet (flow_mol, enth_mol, pressure)
- secondary_air_inlet (flow_mol, enth_mol, pressure)
- number of water wall zones (number_of_zones)
- Coal composition in dry basis (mf_C_coal_dry)
- coal flowrate (coal_flowrate_raw)
- moisture content in the coal (mf_H2O_coal_raw)
- stoichiometric ratio (SR)*
- primary air to coal ratio (ratioPA2coal)*
- heat flux (or heat_duty) to water wall zones, platen superheater, and roof superheater

* not required if calculate_PA_SA_flows is False

Model outputs:

- flue_gas_outlet (flow_mol, enth_mol, pressure)
- heat duty to water wall (ww_heat), platen superheater (platen_heat)*, and roof and backpass (roof_heat)*

Note that platen_heat and roof_heat are only constructed if arguments are equat True

Degrees of Freedom

if calculate_PA_SA_flows is True: By specifying the inlet conditions (primary air and secondary air temperature and pressure), stoichiometric ratio, primary air to coal ratio, coal composition, coal High Heating Value, coal flowrate (raw), moisture content in coal, and surrogate_dictionary, the model will be fully specified. Things that are frequently fixed are:

if calculate_PA_SA_flows is False: This means users “know” or have measurements of the primary air and secondary air, therefore, stoichiometric ratio and primary air to coal ratio are not required to estimate the primary and secondary air. By specifying the primary air inlet (flow_mol_comp, temperature, pressure), secondary air inlet (flow_mol_comp, temperature, pressure), coal composition, coal High Heating Value, coal flowrate (raw), moisture content in coal, and surrogate_dictionary, the model will be fully specified. Things that are frequently fixed are:

Constraints

The main constraints in the model satisfy the energy balance and calculate flue gas outlet conditions (flow_mol_comp, temperature, pressure).

$$Heat_{in} = Heat_{out}$$

$$Heat_{in} = Coal_{massflow} * H_{coal} + Primary_{airmolflow} * enth_{mol_{primaryair}} + Secondary_{airmolflow} * enth_{mol_{secondary}}$$

$$Heat_{out} = flue_gas_{molarflow} * enth_{mol_{fluegas}} + ww_{heat} + platen_{heat} + roof_{heat} + ash_{massflow} * Hs_{flyash}$$

$$ww_{heat} = f(coal_flow, secondaryairtemperature, stoichiometricratio, ratioPA2coal, walltemperature)$$

$$platen_{heat} = f(coal_flow, secondaryairtemperature, stoichiometricratio, ratioPA2coal, walltemperature)$$

$$roof_{heat} = f(coal_flow, secondaryairtemperature, stoichiometricratio, ratioPA2coal, walltemperature)$$

$$NOx = f(coal_flow, secondaryairtemperature, stoichiometricratio, ratioPA2coal, walltemperature)$$

$$flyash = f(coal_flow, secondaryairtemperature, stoichiometricratio, ratioPA2coal, walltemperature)$$

$$T_{coal} = T_{primaryair}$$

where:

- Flow_mol_comp in mol/s
- Temperature in K
- Pressure in Pa
- Heat duty in W
- Coal mass flow after removing the moisture content kg/s

Water Tank

The IDAES water tank model represents a unit operation for storing water. The water tank model supports several shapes including rectangular, vertical and horizontal cylindrical.

Model Structure

The water tank unit model consists of a single `ControlVolume0D` (named `control_volume`) with one Inlet Port (named `inlet`) and one Outlet Port (named `outlet`).

Construction Arguments

Similar to other IDAES unit models, the water tank has the following construction arguments:

Argument	Default Value
dynamic	False
include_holdup	False
material_balance_type	MaterialBalanceType.componentPhase
energy_balance_type	EnergyBalanceType.enthalpyTotal
momentum_balance_type	MomentumBalanceType.pressureTotal
has_heat_transfer	True
has_pressure_change	True
property_package	Parent value
property_package_args	--

Additionally, the water tank model has one specific construction argument to declare the tank shape:

- **tank_type**: configuration argument to define the shape of the tank to be modeled, and accordingly calculate the volume of the filled level. Currently, the supported values are: `simple_tank`, `rectangular_tank`, `vertical_cylindrical_tank`, and `horizontal_cylindrical_tank`. Being `simple_tank` the default value.

Variables

The following variables are added to the model independently of the tank type selected:

Model Inputs (variable name) - symbol:

- water inlet (inlet: `flow_mol`, `enth_mol`, `pressure`)
- tank level (`tank_level`) - l
- heat duty (`heat_duty`) - Q

Model Outputs (variable name):

- water outlet (outlet: `flow_mol`, `enth_mol`, `pressure`)
- pressure drop (`deltaP`) - ΔP

Additionally, some variables are added to the model based on the tank type as indicated below:

tank_type	Variable added
<code>simple_tank</code>	<code>tank_cross_sect_area</code> - A_c
<code>rectangular_tank</code>	<code>tank_width</code> - W , <code>tank_length</code> - L
<code>vertical_cylindrical_tank</code>	<code>tank_diameter</code> - d
<code>horizontal_cylindrical_tank</code>	<code>tank_diameter</code> - d , <code>tank_length</code> - L

Constraints

The main assumptions used in the water tank unit model are:

- 1) Heat loss is a variable given by the user (zero heat loss can be specified if adiabatic)
- 2) Calculate pressure change due to gravity based on water level
- 3) Water level is either fixed for steady-state model or calculated for dynamic model
- 4) Assume $\text{enthalpy_in} == \text{enthalpy_out} + \text{heat loss}$

In addition to the constraints written by the control volume, the water tank model adds two constraints for the pressure drop and the volume of the liquid level in the unit.

Pressure drop constraint:

$$\Delta P = \Delta P_{gravity} = \rho_{liq} * g * l$$

Volume of the liquid constraint:

- 1) for `simple_tank`, `rectangular_tank` and `vertical_cylindrical_tank`:

$$V_{liq} = l * A_c$$

- 2) for `horizontal_cylindrical_tank`:

$$V_{liq} = L * A_t$$

where:

- ρ_{liq} : liquid density
- l : level filled by liquid in the unit
- g : acceleration gravity
- L : tank length
- A_c : cross sectional area of the tank, which for the `simple_tank` is an input variable, while for `rectangular_tank` and `vertical_cylindrical_tank` is an expression calculated by the model
- A_t : area of the circular segment covered by the liquid level at one end of the tank. This is an expression calculated by the model and is only valid for the `horizontal_cylindrical_tank`

The following expressions were used to calculate the tank cross sectional area, and tank area:

- for rectangular_tank: $A_c = W * L$
- for vertical_cylindrical_tank: $A_c = \pi * r^2$, tank_radius (r) is an expression calculated by the model
- for horizontal_cylindrical_tank: $A_t = \cos^{-1}(1-l/r)*r^2 - (r-l)*(2rl-l^2)^{0.5}$

Degrees of Freedom

The degrees of freedom depend on the tank type as the dimension variables are different for each type, but once the dimensions for a specific tank type have been fixed, the model generally has 3-5 degrees of freedom: the inlet state (flow_mol, enth_mol, and pressure), the heat duty whether the config argument has_pressure_change is set to True, and the tank level whether for steady state simulations

Dynamic Model

The dynamic model version of the tank model can be constructed by selecting dynamic=True. If dynamic = True, material accumulation, energy accumulation, and tank level must be calculated. Therefore, a dynamic initialization method has been developed *set_initial_conditions* to initialize the holdup terms.

BoilerHeatExchanger2D

The BoilerHeatExchanger2D model can be used to represent boiler heat exchangers in sub-critical and super critical power plant flowsheets (i.e. economizer, primary superheater, secondary superheater, finishing superheater, reheater, etc.). The model consists of a shell and tube crossflow heat exchanger, in which the shell is used as the gas side and the tube is used as the water or steam side. Due

to the fluid temperature changes along the flow paths inside and outside of the tubes, the velocities of the fluids also change from the inlet to the outlet, causing the changes of heat transfer coefficients and friction factors on both sides along the flow paths. If the flows on both sides can be discretized along the flow paths, local temperature difference between the hot and cold streams (the driving force for heat transfer), local heat transfer coefficients and local friction factors can be used and a more accurate model can be constructed. Figure 1 shows a schematic of the shell and tube cross-flow heat exchanger. In this figure, the hot fluid on the shell side flows from left to right while the cold fluid flows through the tubes up and down. Notice that the cold fluid may enter the tube bundle in multiple rows (2 rows shown in the figure) and flow in parallel. The dash lines show the discretization along the flow path of the hot shell-side flow. The dash lines also cut the tube side flow to multiple segments with the direction of the flow inside the tube switching in two neighboring segments. The flow properties such as heat transfer coefficients and friction factors are calculated in individual discretized elements. Meanwhile the overall flow configuration is either a co-current or counter-current. Counter-current configuration are shown in Figure 1. Since the tube-side flow switches direction from one discretized section to another, pressure drop due the U-turn is also modeled based on the loss coefficient of the U-turn. If the elevation changes between the tube inlet and outlet, the pressure change due to gravity for the tube side fluid is also modeled in each element. Rigorous heat transfer calculations (convective heat transfer for shell side, and convective heat transfer for tube side) and shell and tube pressure drop calculations have been included.

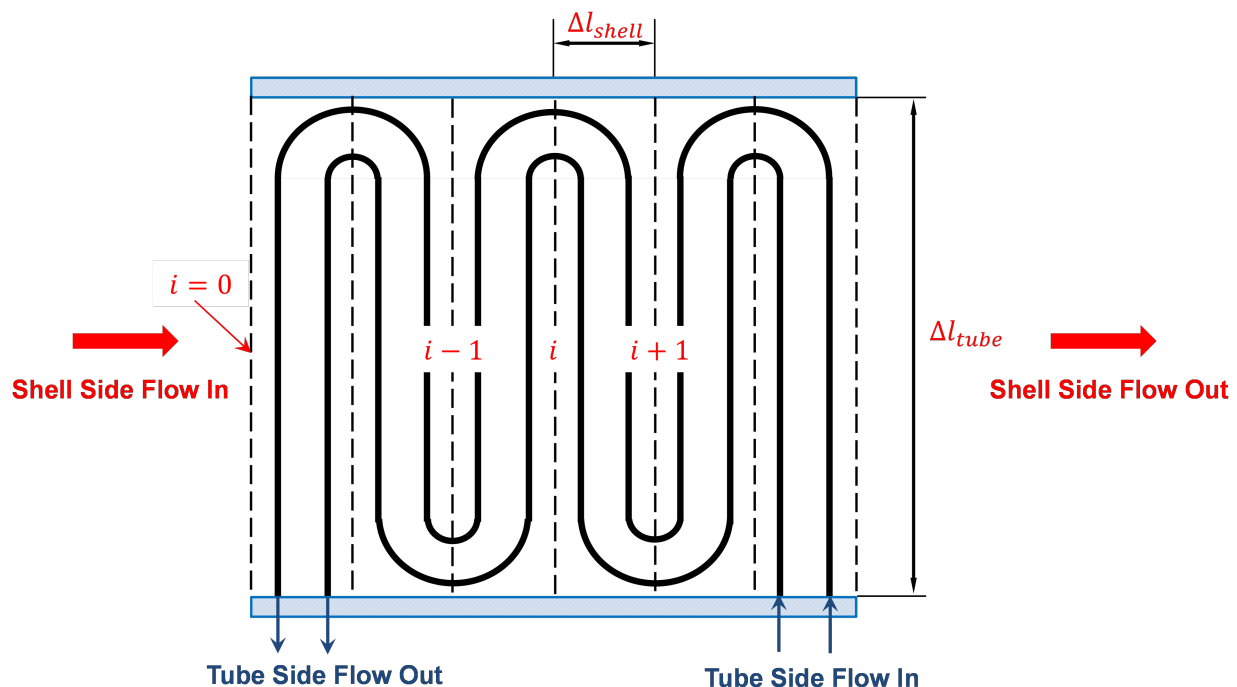


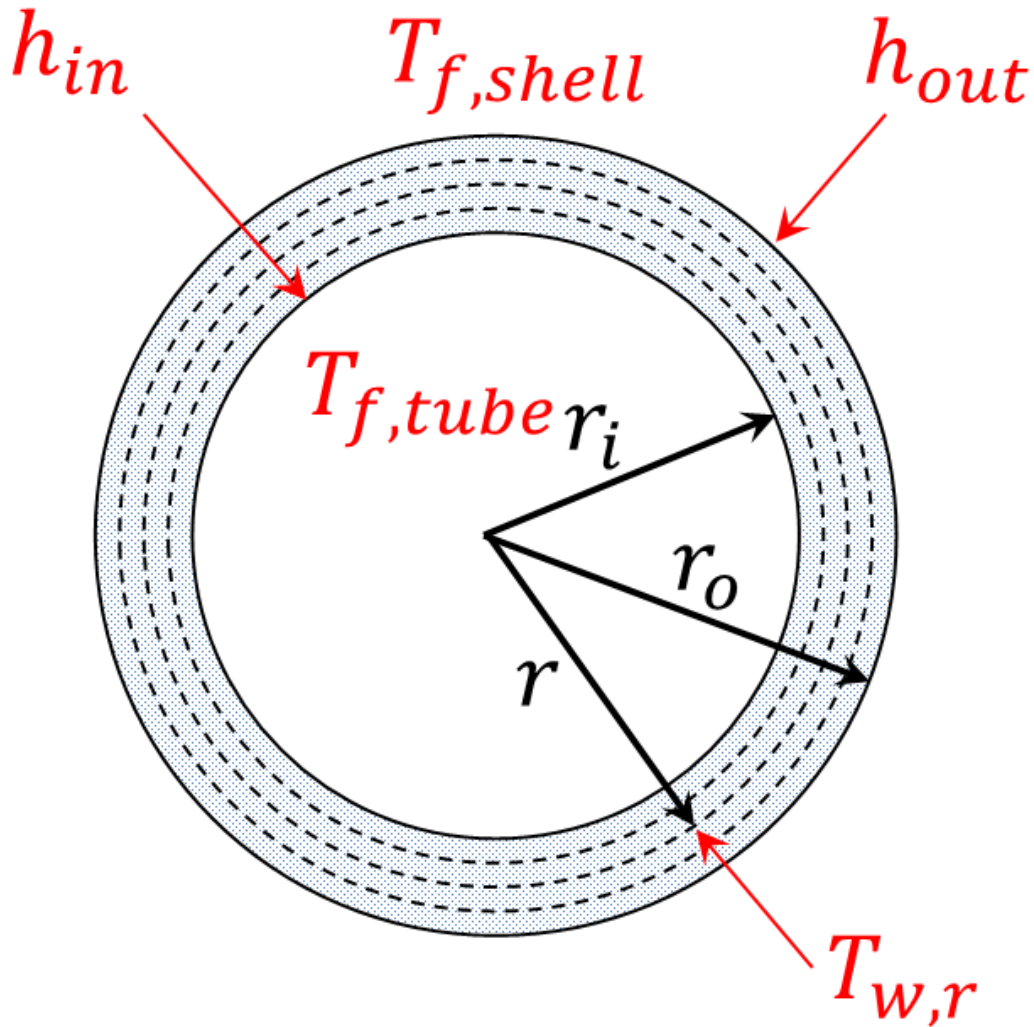
Fig. 22: Cross-flow heat exchanger

In a transient heat transfer process such as in a load ramping operating condition, tube metal wall contains internal

energy and its change with time represents the accumulation term (source or sink) in the energy conservation equation. Due to the high density and high heat capacity of the tube metal, its energy holdup should not be ignored. In other words, the transient tube wall temperatures and its distribution along the wall thickness and along the flow path need to be solved. In addition to the discretization along the flow path direction, the discretization along the tube wall thickness at each discretized flow path section is required, which make the heat exchanger model a 2-D model. Besides, the temperature gradient along the tube thickness is also required to calculate the thermal stress and other equipment health related properties. Figure 2 shows the discretization of tube wall temperature along the tube radius direction. The transient tube wall temperature $T_{(w,r)}$ at each discretized radius r is calculated based on transient heat conduction equation(Eqn. 1), and in the cylindrical coordinate system the heat conduction equation is shown in equation 2.

where, $T_{(w,r)}$ is the tube metal temperature, t is time, α is thermal diffusivity of the tube metal, typically steel, and r is the radius. This partial differential equation can be discretized by Pyomo-DAE in the radius direction. The heat accumulation in the tube metal is represented by the solution of the transient temperatures along the radius direction.

The `HeatExchangerCross-Flow2D_Header` model can be imported from `idaes.power_generation.unit_models`, while additional rules and utility functions can be imported from `idaes.power_generation.unit_models.boiler_heat_exchanger2D`.



$$\frac{\partial T_{w,r}}{\partial t} = \alpha \nabla^2 T_{w,r} \quad (1)$$

$$\frac{\partial T_{w,r}}{\partial t} = \alpha \frac{\partial^2 T_{w,r}}{\partial r^2} + \frac{\alpha}{r} \frac{\partial T_{w,r}}{\partial r} \quad (2)$$

Figure 2. Discretization of heat conduction along tube radius direction

Fig. 23: Cross-flow heat exchanger

Degrees of Freedom

The configuration variables for the 2-D heat exchanger model include the inside diameter of the tube and thickness of the tube. They are used as parameters of the model and have to be declared for discretization in the radius direction. Once declared as configuration arguments, they are not allowed to change (immutable). Other configuration variables include “finite_elements” (the number of elements) in the flow path direction, “radial_elements” (the number of elements in radius direction, “tube_arrangement” for either staggered or in-line arrangement, “has_radiation” if shell-side radiation heat transfer is considered, and “flow_type” for either co-current or counter-current configuration. Additionally, has_header has been added as a configuration argument, when it is True, the health of the water/steam headers is calculated (see header section). The main input variables for the 2-D cross-flow heat exchanger model include design variables such as number of tube segments, number of tube columns, number of tube inlet rows, length of the tube in each segment (each pass), pitches in directions parallel and perpendicular to the shell fluid flow, and elevation change from tube inlet to tube outlet. The thermal and transport properties are also required as well as the mechanical properties if the equipment health model is used. Other required operating variables include fouling resistances on both tube and shell sides, tube wall emissivity if radiation model is turned on, and correction factors for heat transfer and pressure drops on both sides. Given the inlet conditions such as pressures, temperatures and flow rates on both sides, the outlet conditions will be predicted by the model. Meanwhile the temperature and pressure distributions along the flow path direction will be solved on both sides. The 2-D tube wall temperature distribution will also be solved.

In order to capture off design conditions and heat transfer coefficients at ramp up/down or load following conditions, the BoilerHeatExchanger2D model includes rigorous heat transfer calculations. Therefore, additional degrees of freedom are required to calculate Nusselt, Prandtl, Reynolds numbers, such as:

- tube_di (inner diameter)
- tube length
- tube number of rows (tube_nrow), columns (tube_ncol), and inlet flow (nrow_inlet)
- pitch in x and y axis (pitch_x and pitch_y, respectively)

If pressure drop calculation is enabled,

additional degrees of freedom are required:

- elevation with respect to ground level (delta_elevation)
- tube fouling resistance (tube_r_fouling)
- shell fouling resistance (shell_r_fouling)

Model Structure

The `HeatExchangerCrossFlow2D_Header` model contains two `ControlVolume1DBlock` blocks. By default the gas side is named `shell` and the water/steam side is named `tube`. These names are configurable. The sign convention is that duty is positive for heat flowing from the hot side to the cold side.

The control volumes are configured the same as the `ControlVolume1DBlock` in the *Header* model.

The `HeatExchangerCrossFlow2D_Header` model contains additional constraints that calculate the amount of heat transferred from the hot side to the cold side.

The `HeatExchangerCrossFlow2D_Header` has two inlet ports and two outlet ports. By default these are `shell_inlet`, `tube_inlet`, `shell_outlet`, and `tube_outlet`. If the user supplies different hot and cold side names the inlet and outlets are named accordingly.

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	time	Heat transferred from hot side to the cold side
area	A	None	Heat transfer area
U	U	time	Heat transfer coefficient
delta_temperature	ΔT	time	Temperature difference, defaults to LMTD

Note: `delta_temperature` may be either a variable or expression depending on the callback used. If the specified cold side is hotter than the specified hot side this value will be negative.

Constraints

The default constraints can be overridden by providing *alternative rules* for the heat transfer equation, temperature difference, heat transfer coefficient, shell and tube pressure drop. This section describes the default constraints.

Heat transfer from shell to tube:

$$Q = UA\Delta T$$

Temperature difference is:

$$\Delta T = \frac{\Delta T_1 - \Delta T_2}{\log_e \left(\frac{\Delta T_1}{\Delta T_2} \right)}$$

The overall heat transfer coefficient is calculated as a function of convective heat transfer shell and tube, and wall conduction heat transfer resistance.

Convective heat transfer equations:

$$\frac{1}{U} * fcorrection_{htc} = \left[\frac{1}{hconv_{tube}} + \frac{1}{hconv_{shell}} + r + tube_{rfouling} + shell_{rfouling} \right]$$

Tube convective heat transfer (for all elements in tube discretization approach):

$$hconv_{tube} = \frac{Nu_{tube} k}{2tube_{ri}}$$

$$Nu_{tube} = 0.023 Re_{tube}^{0.8} Pr_{tube}^{0.4}$$

$$Pr_{tube} = \frac{Cp\mu}{kMw}$$

$$Re_{tube} = \frac{tube_{ri} 2V\rho}{\mu}$$

Shell convective heat transfer:

$$h_{conv_{shell}} = \frac{Nu_{shell} k_{fluegas}}{tube_{do}}$$

$$Nu_{shell} = f_{arrangement} 0.33 Re_{tube}^{0.6} Pr_{tube}^{0.3333}$$

$$Pr_{shell} = \frac{Cp\mu}{kMw}$$

$$Re_{shell} = \frac{tube_{do} V \rho}{\mu}$$

$$tube_{do} = 2 * tube_{thickness} + tube_{di}$$

Wall heat conduction resistance equation:

$$r = 0.5 * tube_{do} * \log\left(\frac{tube_{do}}{tube_{di}}\right) * k$$

where:

- hconv_tube : convective heat transfer resistance tube side (fluid water/steam) (W / m² / K)
- hconv_shell : convective heat transfer resistance shell side (fluid Flue Gas) (W / m² / K)
- Nu : Nusselt number
- Pr : Prandtl number
- Re : Reynolds number
- V: velocity (m/s)
- tube_di : inner diameter of the tube (m)
- tube_do : outer diameter of the tube (m) (expression calculated by the model)
- tube_thickness : tube thickness (m)
- r = wall heat conduction resistance (K m² / W)
- k : thermal conductivity of the tube wall (W / m / K)
- ρ : density (kg/m³)
- μ : viscosity (kg/m/s)
- tube_r_fouling : tube side fouling resistance (K m² / W)
- shell_r_fouling : shell side fouling resistance (K m² / W)
- fcorrection_htc: correction factor for overall heat transfer
- f_arrangement: tube arrangement factor

Note: by default `fcorrection_htc` is set to 1, however, this variable can be used to match unit performance (i.e. as a parameter estimation problem using real plant data).

Tube arrangement factor is a config argument with two different type of arrangements supported at the moment: 1.- In-line tube arrangement factor (`f_arrangement = 0.788`), and 2.- Staggered tube arrangement factor (`f_arrangement = 1`). `f_arrangement` is a parameter that can be adjusted by the user.

The `HeatExchangerCrossFlow2D_Header` model includes an argument to compute heat transfer due to radiation of the flue gases. If `has_radiation = True` the model builds additional heat transfer calculations that will be added to the `hconv_shell` resistances. Radiation effects are calculated based on the gas gray fraction and gas-surface radiation (between gas and shell).

$$Gas_{grayfrac} = f(gas_{emissivity})$$

$$f_{rad_{gasgrayfrac}} = f(wall_{emissivity}, gas_{emissivity})$$

$$h_{conv_{shell,ad}} = f(k_{boltzmann}, f_{rad_{gasgrayfrac}}, T_{gasin}, T_{gasout}, T_{fluidin}, T_{fluidout})$$

Note: Gas emissivity is calculated with surrogate models (see more details in `boiler_heat_exchanger.py`). Radiation = True when flue gas temperatures are higher than 700 K (for example, when the model is used for units like Primary superheater, Reheater, or Finishing Superheater; while Radiation = False when the model is used to represent the economizer in a power plant flowsheet).

If pressure change is set to True, `deltaP_uturn` and `friction_factor` are calculated

Tube side:

$$\Delta P_{tube} = \Delta P_{tube_{friction}} + \Delta P_{tube_{uturn}} - elevation * g * \frac{\rho_{in} + \rho_{out}}{2}$$

$$\Delta P_{tube_{friction}} = f(tube_{di}, \rho, V_{tube}, numberoftubes, tube_{length})$$

$$\Delta P_{tube_{uturn}} = f(\rho, v_{tube}, k_{lossuturn})$$

where:

- $k_{lossuturn}$: pressure loss coefficient of a tube u-turn
- g : is the acceleration of gravity 9.807 (m/s²)

Shell side:

$$\Delta P_{shell} = 1.4 \Delta P_{shellfriction} \rho V_{shell}^2$$

$\Delta P_{shellfriction}$ is calculated based on the tube arrangement type:

$$\text{In-line: } \Delta P_{shellfriction} = \frac{0.044 + \frac{0.08 \left(\frac{P_x}{tube_{do}} \right)}{0.43 + \frac{1.13}{\left(\frac{P_y}{tube_{do}} - 1 \right)}}}{Re^{0.15}}$$

$$\text{Staggered: } \Delta P_{shellfriction} = \frac{0.25 + \frac{0.118}{\left(\frac{P_y}{tube_{do}} - 1 \right)^{1.08}}}{Re^{0.16}}$$

Figure. Tube Arrangement

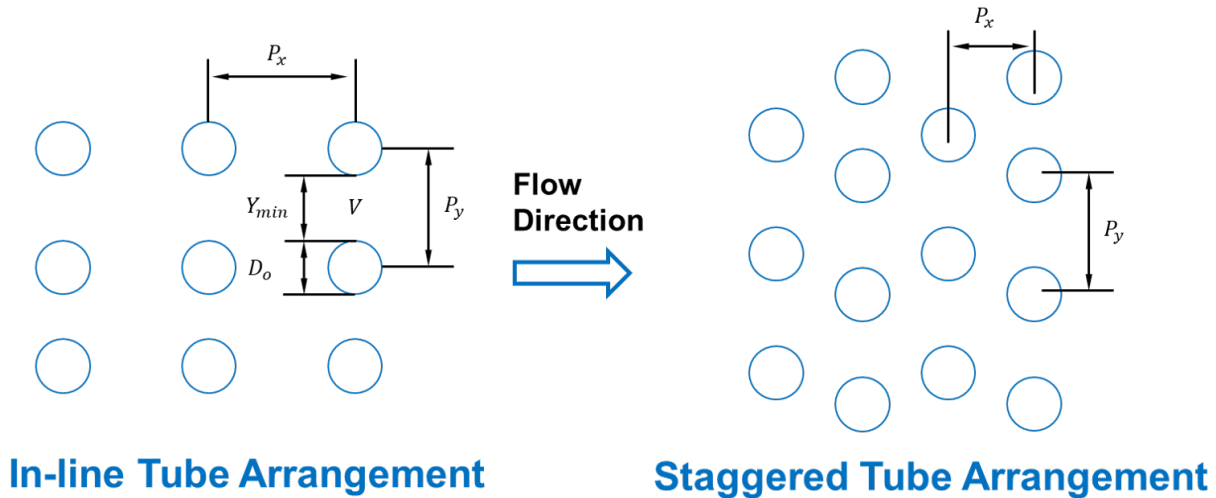


Fig. 24: Tube Arrangement

Header Health Model

The heat exchanger 2D model allows the user to calculate the thermal and mechanical stresses of the water/steam headers connected to the outlet of the tube side. Additionally, the rupture time and fatigue calculation of allowable cycles are computed by the model. A simplified 1D PDE problem is developed to represent the heat conduction transient through the radius of the superheater/reheater headers. Regarding to the flow path configuration (counter-current or co-current) of the 2D heat exchanger, the first or the last discretization point will be used to define the

boundary of the headers. For this example, the last discretization point will be used for the outlet superheater header due to the counter-current flow configuration. Under the assumptions of constant conductivity and no heat generation, the Fourier's equation is converted to the Eq. (h1) for the cylindrical header. The Pyomo.DAE framework is applied to solve the PDE problem. The thermal and mechanical stresses are calculated based on the pressure and temperature difference between both sides of the header which can be used to evaluate the allowable number of cycles of the main body and the critical point of the edge of the hole.

$$\frac{1}{a} \frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial r^2} + \frac{1}{r} \frac{\partial T}{\partial r}$$

where T is wall temperature (K), r is radius (m), and a is the material thermal diffusivity (m^2/s). The material thermal diffusivity is a function of (material thermal conductivity, specific heat, and density)

Detailed description of the mechanical stress calculations and thermal stress calculations can be found in S. Bracco, 2012 and Taler & Duda, 2006, respectively.

Rupture time calculation: The creep phenomenon is an important design consideration in the analysis of structures. At the high temperature operation, the creep is coupled with fatigue due to cycling, the damage will be much higher than that occurring if the same fatigue or creep is working alone. For example, a long-time creep rupture strength values can be derived by using the Manson-Haferd model. However, depending on the investigated material, users can find another correlation to calculate the rupture strength in the open literature.

Fatigue calculation of allowable cycles: For general ferritic and austenitic materials, the calculation of the allowable number of cycles are expressed in the following equation. However, the users can be recommended to find a specific fatigue equation for their own material to obtain a better result. Using the calculated stresses above, the number of allowable cycles of the component can be evaluated based on fatigue assessment standard, such as EN 13445. The detail of the developed approach can be found in Bracco's report (S. Bracco, 2012). This model can be applied for both drum and thick-walled

components such as header. According to the EN 13445 standard, for a single cycle, the allowable number of fatigue cycles N can be computed as:

$$N = \frac{46000}{\Delta\Sigma R_i - 0.63R_m + 11.5}$$

where R_m is the material tensile strength at room temperature while the reference stress range Δ_i depends on the stress range $\Delta\Sigma_i$

Bracco, S. (2012). Dynamic simulation of combined cycles operating in transient conditions: An innovative approach to determine the steam drums life consumption. In Proceedings of the 25th International Conference on Efficiency, Cost, Optimization and Simulation of Energy Conversion Systems and Processes, ECOS 2012. Taler, J., & Duda, P. (2006). Solving direct and inverse heat conduction problems. Solving Direct and Inverse Heat Conduction Problems. Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-540-33471-2>

Class Documentation

class `idaes.power_generation.unit_models.boiler_heat_exchanger_2D.HeatExchangerCrossFlow2D_Head`

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls `build()`.
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = `useDefault`. **Valid values:** { `useDefault` - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if `dynamic = True`,

default - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

shell_side shell side config arguments

material_balance_type Indicates what type of mass balance should be constructed, **default** - `MaterialBalanceType.componentTotal`. **Valid values:** { **MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - `EnergyBalanceType.enthalpyTotal`. **Valid values:** { **EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - `MomentumBalanceType.pressureTotal`. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_pressure_change Indicates whether terms for pressure change should be

constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

tube_side tube side config arguments

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.componentTotal. **Valid values:** { **MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance. }

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.enthalpyTotal. **Valid values:** { **EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase. }

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **MomentumBal-**

anceType.none - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

transformation_method Discretization method to use for DAE transformation. See Pyomo documentation for supported transformations.

transformation_scheme Discretization scheme to use when transforming domain. See Pyomo documentation for supported schemes.

finite_elements Number of finite elements to use when discretizing length domain (default=5). Should set to the number of tube rows

collocation_points Number of collocation points to use per finite element when discretizing length domain (default=3)

flow_type Flow configuration of heat exchanger co_current: shell and tube flows from 0 to 1 counter_current: shell

side flows from 0 to 1 tube side flows
from 1 to 0

tube_arrangement Tube arrangement
could be in-line or staggered

tube_side_water_phase Define water
phase for property calls

has_radiation Define if shell side gas ra-
diation is to be considered

tube_inner_diameter User must define
inner diameter of tube

tube_thickness User must define tube
wall thickness

radial_elements Number of finite ele-
ments to use when discretizing radius
domain (default=5).

header_inner_diameter User must de-
fine inner diameter of header

header_wall_thickness User must define
header wall thickness

header_radial_elements Number of fi-
nite elements to use when discretizing
radius domain (default=5).

has_header If has_header is True, user
must provide header thickness and in-
ner diameter.

- **initialize** (*dict*) – Process-
BlockData config for individual
elements. Keys are BlockData indexes
and values are dictionaries described
under the “default” argument above.
- **idx_map** (*function*) – Function to
take the index of a BlockData element
and return the index in the initialize dict
from which to read arguments. This can
be provided to override the default be-
havior of matching the BlockData index
exactly to the index in initialize.

Returns (HeatExchangerCross-
Flow2D_Header) New instance

class `idaes.power_generation.unit_models.boiler_heat_exchanger_2D.HeatExchangerCrossFlow2D_Head`
Standard Heat Exchanger Cross Flow
Unit Model Class.

build()
Begin building model.

Parameters None –

Returns None

initialize (*shell_state_args=None, tube_state_args=None, outlvl=0, solver='ipopt', optarg={'tol': 1e-06}*)
HeatExchangerCrossFlow1D initialisation routine

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = None).
- **outlvl** – sets output level of initialisation routine
 - 0 = no output (default)
 - 1 = return solver state for each step in routine
 - 2 = return solver state for each step in subroutines
 - 3 = include solver output information (tee=True)
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

Drum 1D Model

Introduction

The 1-D drum model is similar to the 0-D drum model, however, the 1-D heat conduction through the radius or thickness of the drum wall is modeled. The heat conduction through insulation is modeled as steady state and no energy holdup or accumulation in the insulation layer is considered. Dynamics and energy holdup are accounted for in the drum metal.

Similar to the drum model, the drum 1D model consists of three main sub-unit operations:

- 1) a flash model to separate the saturated steam from the saturated liquid water in the water/steam mixture,
- 2) a mixer model to mix saturated liquid water with feed water, and
- 3) a water tank model to calculate drum level and pressure drop.

First the water/steam mixture from boiler waterwall tubes (risers) enters the flash model and leaves in two separate streams (liquid water and steam). Then, the saturated water from the flash model is mixed with the feed water stream (typically from the economizer or a water pipe linking the economizer and the drum) and leave the mixer model in a single mixed stream. Finally, the mixed stream enters the water tank of the drum and leaves the vessel through the multiple downcomers (see Figure 1).

The sub-unit models for the flash and the mixer (Items 1 and 2 in the above list) are identical to the 0-D drum model. The main difference between them is the way the horizontal water tank model is modeled, especially with respect to the heat transfer from the liquid water through the drum metal wall, its insulation layer and to the ambient air. In other words, the drum is not adiabatic. The heat transfer from the liquid water in the drum to the ambient air includes a) convective heat transfer between the liquid water and the wetted inner wall of the drum, b) heat conduction from the inner drum wall to the outer drum wall through the drum metal thickness, c) heat conduction from the insulation layer inner wall to its outer wall, and d) natural heat convection from the insulation outer wall to the ambient air, typical at a room temperature.

Figure 1 is a schematic of the cross-sectional area of the drum. As can be seen in the figure, the liquid water occupies the lower part of the drum and the saturated steam occupies the upper part. The metal wall and insulation layer are also shown in the figure. The water/steam mixture and feed water enter the drum while the saturated steam leaves the drum through the pipes in the upper part and the subcooled water leaves the drum to the downcomers.

Inlet Ports:

- water_steam_inlet: water/steam mixture from waterwall

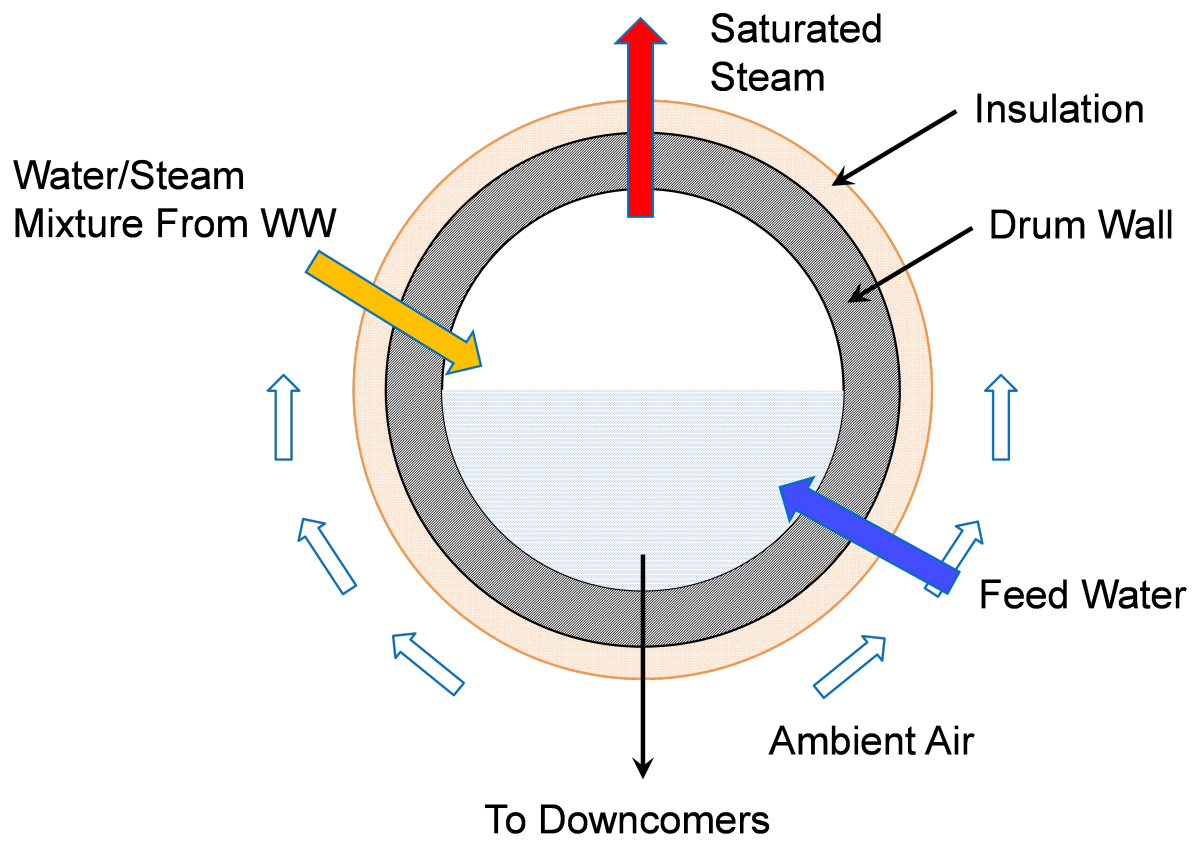


Fig. 25: Figure 1. Schematic of a boiler drum with metal wall and insulation layer

- feedwater_inlet: feedwater from economizer/pipe

Outlet Ports:

- liquid_outlet: liquid to downcomer
- steam_outlet: saturated steam leaving the drum

Variables

Model inputs (variable name):

- water/steam inlet (water_steam_inlet: flow_mol, enth_mol, pressure)
- feedwater inlet (feedwater_inlet: flow_mol, enth_mol, pressure)
- drum diameter (drum_diameter)
- drum length (drum_length)
- number of downcomer tubes (number_downcomers)
- downcomer diameter (downcomer_diameter)
- drum level (drum_level)
- heat duty (heat_duty)

Model Outputs:

- vapor outlet (vap_outlet: flow_mol, enth_mol, pressure)
- liquid outlet (liq_outlet: flow_mol, enth_mol, pressure)

Constraints

As mentioned above, the drum model imports a *HelmPhaseSeparator* and mixer models, specific documentation for these models can be obtained in: Once the water enters the tank model the main equations calculate water velocity and pressure drop calculation due to gravity based on water level and contraction to downcomer. Water level (drum_level) is either fixed for steady state simulation or calculated for dynamic model (Dynamic = True)

Main assumptions:

- 1) Heat loss is a variable given by the user (zero heat loss can be specified if adiabatic)
- 2) Pressure change due to gravity based on water level and contraction to downcomer is calculated
- 3) Water level is either fixed for steady-state model or calculated for dynamic model
- 4) Assume $\text{enthalpy_in} == \text{enthalpy_out} + \text{heat loss} + \text{energy accumulation}$
- 5) Subcooled water from economizer and saturated water from waterwall are well mixed before entering the drum

Pressure equality constraint:

$$P_{SaturatedWater} = P_{FeedWater}$$

Pressure drop in unit:

$$\text{delta}P = \text{delta}P_{contraction} + \text{delta}P_{gravity}$$

$$\text{delta}P_{gravity} = f(\rho_{liquid}, \text{accelerationgravity}, \text{drum_level})$$

$$\text{delta}P_{contraction} = f(\rho_{liquid}, V)$$

where: * V: fluid velocity (m/s, liquid only)

Note that the model builds an Pyomo Arc to connect the Liquid_outlet from the self.aFlash unit to the SaturatedWater inlet port of the mixer, and the mixed_state (Mixer outlet) is directly constructed as the Drum *control_volume.properties_in*. Once the Drum model is constructed, the mixer and flash blocks can be found as *self.aDrum.aMixer* and *self.aDrum.aFlash*

Convective heat transfer: Strictly speaking, the inner drum wall temperature is not uniform along the circumference since the temperature of the wetted lower section is different from that of the upper section in contact with the saturated steam. The heat convection between the liquid water and the inner drum wall is considered as the dominant mechanism compared to the heat convection between the saturated steam and the inner drum wall in the upper dry

section. The main assumption for the 1-D drum model is that the latter part can be ignored and inner drum wall temperature is uniformly distributed. The convective heat transfer coefficient between the liquid water and the inner wetted wall (wetted section only) is calculated based on pool boiling assumption

$$h_{in} = f(Pr, Pred, Mw, T_{wall,in}, T_{liq})$$

where P_{red} is the reduced pressure (ratio of the drum pressure to the critical pressure of water), Mw is the molecular weight of water in mol/g, $T_{(wall,in)}$ is the wetted drum inner wall temperature, and T_{liq} is the liquid water temperature. The heat transfer coefficient of the natural heat convection $h_{(conv,ins)}$ at the outer insulation wall can be calculated from the Nusselt number N_u by.

$$h_{conv,ins} = \frac{N_u k_{air}}{D_{o,ins}}$$

where $D_{(o,ins)}$ is the outside diameter of the insulation layer and k_{air} is the thermal conductivity of air. The N_u for natural convection of a horizontal cylindrical wall is correlated to Rayleigh number R_a and Prandtl number of air $P_{(r,air)}$ by

The Rayleigh number R_a is defined as

$$R_a = \frac{g(T_{wall,out,ins} - T_{amb})D_{o,ins}^3}{\nu_{air} \alpha_{air}}$$

where g is gravity, α_{air} is thermal expansion coefficient of air, $T_{(wall,out,ins)}$ is the outside insulation wall temperature, T_{amb} is the ambient temperature, ν_{air} is kinematic viscosity of air, and α_{air} is thermal diffusivity of air. To simplify the model, the thermal and transport properties of air are assumed to be constant at a film temperature, the average of the room temperature of 25 C and a insulation wall temperature of 80 C.

The equivalent heat transfer coefficient of the natural convection at the drum metal wall outside the boundary ($h_{conv,drum}$) can be calculated from $h_{conv,ins}$ as:

$$h_{conv,drum} = \frac{D_{o,ins} h_{conv,ins}}{D_{o,drum}}$$

where $D_{(o,drum)}$ is the outside diameter of the drum metal wall

The energy accumulation for the insulation layer is ignored due to its low heat capacity compared with the drum metal wall. The heat transfer resistance of the insulation layer based on inner insulation area is considered though

($r_{ht,ins}$). The heat transfer resistance of the insulation layer and the natural heat convection are combined to obtain the equivalent overall heat transfer coefficient at the outer boundary of the drum metal wall (h_{out}).

The heat conduction through the thickness or radius of the drum metal can be described by a transient heat conduction equation of solid as

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

In cylindrical coordinate system, it can be written as

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial r^2} + \frac{\alpha}{r} \frac{\partial T}{\partial r}$$

where T is the drum metal temperature, t is time, α is thermal diffusivity of drum metal, typically steel, and r is the radius. This partial differential equation can be discretized by Pyomo-DAE in the radius direction. The heat accumulation in the drum metal is represented by the solution of the transient temperatures along the radius direction. To solve the transient heat conduction problem, we need to specify the boundary conditions. Figure 2 shows the drum metal wall and the liquid water inside the drum.

Drum Health Model

The model can be used to calculate the stress and allowable number of cycles of both the main body and location of critical point of the opening junction. During transient operation, the component is subject to variations of pressure and temperature which cause thermal stress and thermo-mechanical fatigue. The temperature difference in both sides of the metal causes the thermal stress. Also, the cylinder is subjected to an inside and outside pressure, which can obtain the mechanical stresses. The mechanical and thermal stresses are considered:

- Mechanical stress is calculated using S. Bracco, 2012 reference, and it is a function of the pressure and radius at the inside and outside surfaces.

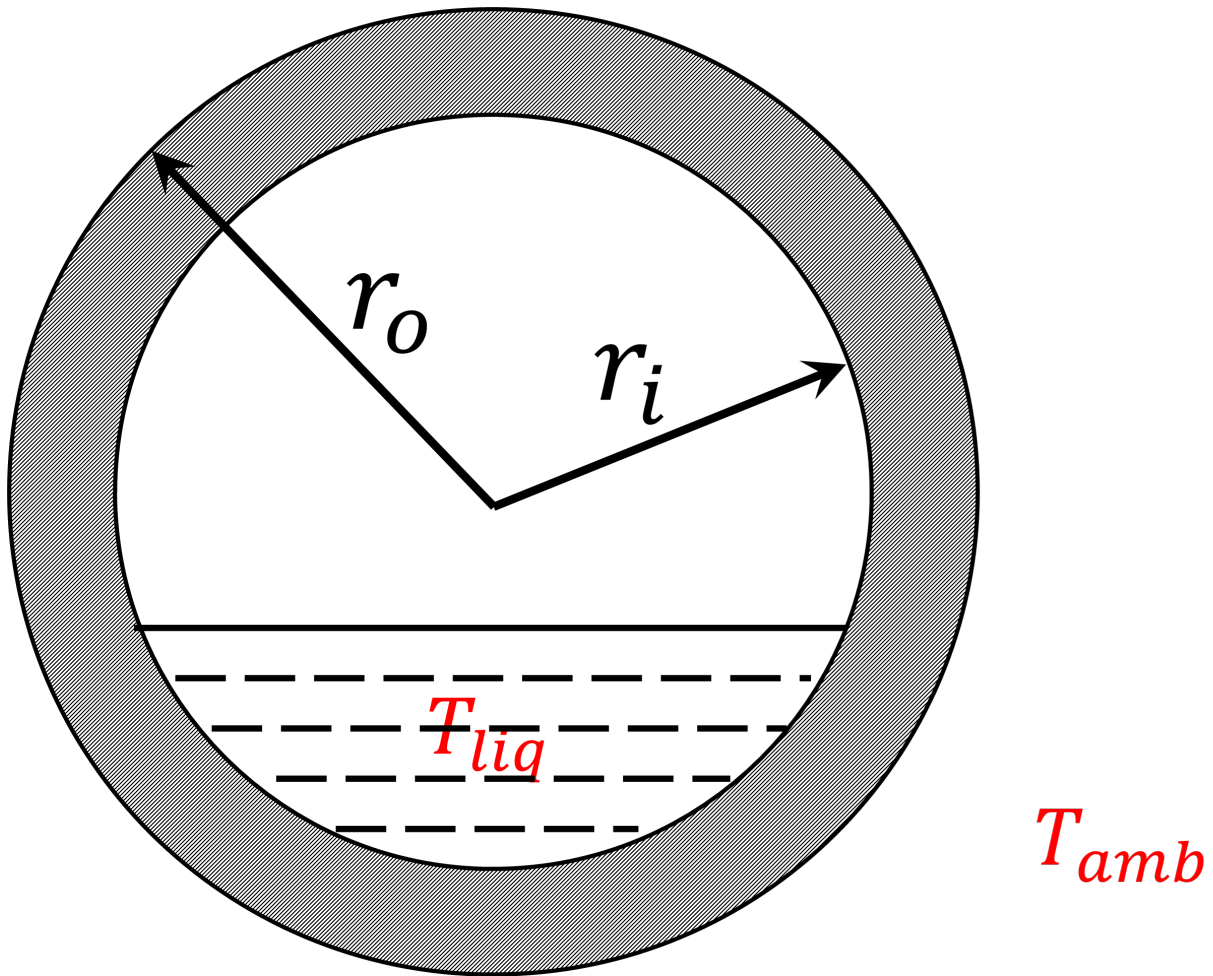


Fig. 26: Figure 2. Drum metal wall with liquid water

- Thermmal stress is calculaed using Taler & Duda, 2006 refernce, and it is a function of the Young modulus, a linear temperature expansion coefficient, and Poisson ratio of the steel material.

Fatigue calculation of allowable cycles: Using the calculated stresses above, the number of allowable cycles of the component can be evaluated based on fatigue assessment standard, such as EN 13445. The detail of the developed approach can be found in Bracco's report (S. Bracco, 2012). This model can be applied for both drum and thick-walled components such as header. According to the EN 13445 standard, for a single cycle, the allowable number of fatigue cycles N can be computed as a function of the material tensile strength at room temperature and a reference stress range.

[1] Bracco, S. (2012). Dynamic simulation of combined cycles operating in transient conditions: An innovative approach to determine the steam drums life consumption. In Proceedings of the 25th International Conference on Efficiency, Cost, Optimization and Simulation of Energy Conversion Systems and Processes, ECOS 2012.

[2] Taler, J., & Duda, P. (2006). Solving direct and inverse heat conduction problems. Solving Direct and Inverse Heat Conduction Problems. Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-540-33471-2>

[3] European standard EN 13445: 2002 (2005). Unfired pressure vessels, (parts 1-5), CEN European Committee for Standardization, Part 3: Design, Issue 14.

Degrees of Freedom

Once the unit dimensions have been fixed, the model generally has 5 degrees of freedom. The water/steam mixture inlet state (flow_mol, enth_mol, and pressure) and feewater inlet state (flow_mol and enth_mol). The feedwater inlet pressure is usually free due to the pressure equality mentioned above.

Dynamic Model

The dynamic model version of the drum model can be constructed by selecting `dynamic=True`. If `dynamic = True`, material accumulation, energy accumulation, and drum level must be calculated. Therefore, a dynamic initialization method has been developed *set_initial_conditions* to initialize the holdup terms.

Water Pipe Model

Introduction

The water pipe model is used to model a water or steam pipe connecting two units in a power plant. It calculates the pressure change between the pipe inlet and outlet due to friction, gravity, and optional expansion or contraction at the end of the pipe. The water pipe model does not provide the equations to calculate the heat loss. However, the user can specify the heat duty if configuration variable “has_heat_transfer” is set to True. When declaring the water pipe model, the user needs to provide typical configuration variables for a control volume, the base class the model is derived from, including “dynamic”, “has_holdup”, “has_heat_transfer”, “has_pressure_change”, etc. While most of configuration variables have default values, the configuration variable for “property_package” has to be given as the IDAES property package for water implemented based on IAPWS water property table. The user needs to set the variable “water_phase” either as “Liq” for liquid water or “Vap” for

water vapor. Currently the model does not support the pipe with two phase flow since the two-phase flow is usually unstable. The user also needs to specify the configuration variable “contraction_expansion_at_end”. If there is a contraction at the end of the pipe, the value for the variable should be “contraction”. If there is an expansion at the end, the value should be “expansion”. The value of “None” is used if there is no contraction or expansion at the end of the pipe.

Model inputs (variable name):

- number of pipes (number_of_pipes)
- tube dimensions (length, inner diameter, elevation change) (length, diameter, elevation_change)
- water/steam flow rate and states at inlet (flow_mol, enth_mol, pressure)
- pressure drop correction factor (fcorrection_dp)
- heat duty (usually fixed equal to 0)
- if expansion at the end of pipe is True, user needs to specify area ratio at the end (area_ratio), which is the area after contraction or expansion divided by the cross sectional area of the pipe)

Model Outputs:

- pressure drop (deltaP)
- water/steam flow rate and states at outlet (flow_mol, enth_mol, pressure)

Degrees of Freedom

As mentioned above, the waterpipe model includes rigorous pressure drop, therefore, detailed pipe dimensions are required. Aside from the inlet conditions and tube dimensions, the waterpipe model usually has one degree of freedom, the heat duty, which can be fixed for it to be fully specified.

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	time	Heat transferred from flue gas to tube side fluid
deltaP		time	Pressure drop

Constraints

The main constraints in this model are used to compute the pressure drop. Three types of pressure changes are considered in the pipe model, including the pressure change due to friction along the pipe length, the pressure change due to gravity if there is an elevation change from the inlet to the outlet, and the pressure change due to the contraction or expansion at the end.

Pressure drop:

$$\Delta P = \Delta P_{friction} + \Delta P_{gravity} + \Delta P_{contraction}$$

$$\Delta P_{tube_{friction}} = f(diameter, \rho, V, numberofpipes, length, Re, fcorrection_{dp})$$

$$\Delta P_{gravity} = f(\rho, accelerationgravity, elevationchange)$$

$$\Delta P_{contraction} = f(\rho, V, K_{loss})$$

Reynolds number:

$$Re_{tube} = \frac{tube_{di} V \rho}{\mu}$$

where:

- fcorrection_dp: correction factor if the pipe is not smooth
- diameter: inner diameter (m)
- ρ : density (kg/m³)
- Re: Reynolds number
- V: fluid velocity (m/s)
- Kloss: loss coefficient due to the contraction (function of the area ratio)

Dynamic Model

The dynamic model version of the steam heater model can be constructed by selecting dynamic=True. If dynamic = True, the material accumulation holdups are constructed. While, the metal energy holdup is not considered.

Heat Exchanger With Three Streams

The `HeatExchangerWith3Streams` model consists of a heat exchanger with three inlets, *side_1* represents the hot stream, while *side_2* and *side_3* are cold streams. This model is a simplified generic heat exchanger model with lumped UA (the product of the overall heat transfer coefficient and the heat transfer area).

In a power plant flowsheet this model is used to represent an air preheater unit. This is because modeling the Ljungström type preheater is quite challenging since it involves not only the hot and cold gas streams but also the energy stored in and released from the metal parts.

Degrees of Freedom

Aside from the inlet conditions, a 3 inlet heat exchanger model usually has six degrees of freedom, which must be fixed for it to be fully specified. Things that are frequently fixed are two of:

- `UA_side_2` - lumped overall heat transfer and heat transfer area of side 2
- `UA_side_3` - lumped overall heat transfer and heat transfer area of side 3
- `frac_heatloss` - fraction of heat loss in the system
- `deltaP_side_1` - pressure drop in side 1
- `deltaP_side_2` - pressure drop in side 2
- `deltaP_side_3` - pressure drop in side 3

Model Structure

The `HeatExchangerWith3Streams` model contains three `ControlVolume0DBlock` blocks. The hot side is named *side_1* and two cold sides are named *side_2* and *side_3*. These names are not configurable. The sign convention is that duty is positive for heat flowing from the hot side to the cold side.

The control volumes are configured the same as the `ControlVolume0DBlock` in the *Heater model*. The `HeatExchangerWith3Streams` model contains additional constraints that calculate the amount of heat transferred from the hot side to the cold side.

The `HeatExchangerWith3Streams` has three inlet ports and three outlet ports. By default these are `side_1_inlet`, `side_2_inlet`, `side_3_inlet`, `side_1_outlet`, `side_2_outlet`, `side_3_outlet`.

Variables

Variable	Symbol	Index Sets	Doc
heat_duty	Q	time	Heat transferred (model includes 3 variables, one for each side)
UA	UA	None	lumped Heat transfer area and overall heat transfer coefficient
LMTD	$LMTD$	time	Log Mean Temperature difference, LMTD

Constraints

The default constraints can be overridden by providing *alternative rules* for the heat transfer equation, temperature difference, heat transfer coefficient, shell and tube pressure drop. This section describes the default constraints.

Heat transfer from hot to cold sides:

$$Q_{side_1} * (1 - frac_{heat_loss}) = Q_{side_2} + Q_{side_3}$$

$$Q_{side_2} = UA_{side_2} \Delta T_2$$

$$Q_{side_3} = UA_{side_3} \Delta T_3$$

Temperature difference is:

$$\Delta T = \frac{\Delta T_1 - \Delta T_2}{\log_e \left(\frac{\Delta T_1}{\Delta T_2} \right)}$$

Note: ΔT_2 is a function of hot stream *side 1* and cold stream *side 2*, and ΔT_3 is a function of hot side and cold stream *side 3*.

Class Documentation

class `idaes.power_generation.unit_models.heat_exchanger_3streams.HeatExchangerWith3Streams` (*args, **kwargs)

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

side_1_property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flowsheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

side_1_property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation. }

side_2_property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flowsheet, **PhysicalParameterObject** - a PhysicalParameterBlock object. }

side_2_property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

side_3_property_package Property parameter object used to define property calculations, **default** - useDefault. **Valid values:** { **useDefault** - use default package from parent model or flowsheet, **PhysicalParameterObject** - a PhysicalParameterBlock object.}

side_3_property_package_args A ConfigBlock with arguments to be passed to a property block(s) and used when constructing these, **default** - None. **Valid values:** { see property package for documentation.}

material_balance_type Indicates what type of material balance should be constructed, **default** - MaterialBalanceType.componentPhase. **Valid values:** { **MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - EnergyBalanceType.enthalpyTotal. **Valid values:** { **EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - MomentumBalanceType.pressureTotal.

Valid values: { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase. }

has_heat_transfer Indicates whether terms for heat transfer should be constructed, **default** - False. **Valid values:** { **True** - include heat transfer terms, **False** - exclude heat transfer terms. }

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - False. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms. }

flow_type_side_2 Flag indicating type of flow arrangement to use for heat exchanger, **default** 'counter-current' counter-current flow arrangement

flow_type_side_3 Flag indicating type of flow arrangement to use for heat exchanger (default = 'counter-current' - counter-current flow arrangement

- **initialize** (*dict*) - Process-BlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the "default" argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HeatExchangerWith3Streams)
New instance

```
class idaes.power_generation.unit_models.heat_exchanger_3streams.HeatExchangerWith3StreamsData  
    Standard Heat Exchanger Unit Model  
    Class
```

```
build()
```

Begin building model

```
initialize (state_args_1=None, state_args_2=None, state_args_3=None, outlvl=0, solver='ipopt',  
            optarg={'tol': 1e-06})
```

General Heat Exchanger initialisation routine.

Keyword Arguments

- **state_args_1** – a dict of arguments to be passed to the property package(s) for side 1 of the heat exchanger to provide an initial state for initialization (see documentation of the specific property package) (default = None).
- **state_args_2** – a dict of arguments to be passed to the property package(s) for side 2 of the heat exchanger to provide an initial state for initialization (see documentation of the specific property package) (default = None).
- **state_args_3** – a dict of arguments to be passed to the property package(s) for side 3 of the heat exchanger to provide an initial state for initialization (see documentation of the specific property package) (default = None).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

Feedwater Heater Dynamic (0D)

The FWH0DDynamic model is a 0D feedwater heater model suitable for dynamic modeling. It is intended to be used primarily with the [IAWPS95](#) property package. The feedwater heater is split into three sections. The condensing section is required while the desuperheating and drain cooling sections are optional. There is also an optional mixer for adding a drain stream from another feedwater heater to the condensing section. The figure below shows the layout of the feedwater

heater. All but the condensing section are optional.

Fig. 27: Feedwater Heater

Features of Dynamic Model

The dynamic version of the 0-D feed water heater model is based on the steady-state feed water heater 0D model *FWH0D*. It contains additional variables related to the mass and energy inventories inside the condensing section of the feed water heater on both the tube and the shell sides. The desuperheater and drain cooler sections, if any, are usually treated as steady-state. For the condensing section, the tubes are always filled with condensate or feed water and the total volume of the liquid inside the tube is usually specified as a user input. The shell side, however, is partially filled with saturated liquid water and the water level changes with time and so is the volume of the saturated water. Only the horizontal design of the feed water heater is modeled here and it usually consists of a cylindrical tank with a fraction of its internal volume occupied by the heat transfer tubes

with the remaining gaps between the tubes occupied either by the steam or the saturated water. The shell side of the condenser section is modeled as a horizontal cylinder similar to that in a drum model *Drum1D* with the total volume of the shell side liquid and the submerged tubes calculated from the water level (see the description of the drum model). The volume of the saturated liquid is simply a fraction of the total volume. Therefore, the additional input variables in the dynamic version of the feed water heater model include the inner diameter of the feed water heater cylinder, the length of the condensing section, the fraction of the volume occupied by the shell side liquid (gaps between the tubes), and the volume of the feed water inside the tubes of the condensing section. Those input variables are constant (do not change with time) for a given design. In addition, the water level defined as the distance from the bottom of the cylinder to the top of the water is also included in the model and indexed by time. The dynamic version of the feed water heater model provides the constraint (equation) to calculate the volume of the shell-side liquid as a function of the water level. The mass and energy accumulation terms are handled by the IDAES control volume class based on the volumes of the tube-side and the shell-side liquids. Since the density of the steam on the shell side is much lower than the density of the liquid, the mass and energy accumulations of the steam above the water level are ignored.

Note that the total heat transfer area and overall heat transfer coefficient are required inputs as in the steady-state model for the condensing section. The overall heat transfer coefficient is dominated by the tube-side convective heat transfer coefficient since the shell-side heat transfer coefficient is usually very

high due to the phase change. Based on an empirical correlation (Bird et al, 1960), the Nusselt number on the tube side is proportional to the Reynolds number to the power of 0.8. Therefore, the overall heat transfer coefficient is approximately proportional to the feed water flow rate to the power of 0.8. A flowsheet level constraint can be imposed to account for the effect of feed water flow rate on the overall heat transfer coefficient.

Initial Condition of Dynamic Model

Typical initial condition for the dynamic model is a steady state condition. The user can call *set_initial_condition* function of the model to initialize the variables related to the material and energy accumulation terms for the dynamic model. Note that the water level at the initial time usually should be fixed to ensure the inventories of mass and energy are well defined.

Degrees of Freedom

The `area` and `overall_heat_transfer_coefficient` should be fixed or constraints should be provided to calculate `overall_heat_transfer_coefficient`. In addition, the geometry variables related to the condensing section including `heater_diameter`, `cond_sect_length`, `vol_frac_shell`, and `tube` volume should be fixed. The initial value of `level` should also be fixed.

If the inlets are also fixed except for the inlet steam flow rate (`inlet_1.flow_mol`), the model will have 0 degrees of freedom.

See `FWH0DDynamic` and `FWH0DDynamicData` for full Python class details.

Property Models

Flue Gas Property Package

A flue gas property package has been developed to provide properties of combustion gases and air. The ideal gas property package includes the main components in flue gas: O₂, N₂, NO, CO₂, H₂O, SO₂

Main parameters:

- molecular weight in kg/kg-mol indexed by component list,
- reference pressure & temperature in Pa and Kelvin,
- critical pressure and temperature in Pa and Kelvin indexed by component list,
- gas constant in J/(mol K),
- constants for specific heat capacity in J/(mol K) indexed by component list and parameter A to H,
- vapor pressure coefficients (Antoine Eq.) P in Bar and T in K indexed by component list and parameters A to C,

Source: NIST webbook (last update: 01/08/2020)

The main methods supported are:

- heat capacity in J/(mol K),
- enthalpy in J/mol,
- entropy in J/(mol K),
- volumetric flowrate m³/s,
- viscosity of mixture in kg/(m s),
- thermal conductivity mixture in J / (m K s),
- molar density m³/mol,
- reduced pressure and temperature (unitless),

Flowsheet Models

Supercritical Coal-Fired Power Plant Flowsheet (steady state)

This is an example supercritical pulverized coal (SCPC) power plant. This simulation model consists of a ~595 MW gross coal fired power plant.

The dimensions and operating conditions used for this simulation do not represent any specific coal-fired power plant.

This model is for demonstration and tutorial purposes only. Before looking at the model, it may be useful to look at the process flow diagram (PFD).

SCPC Power Plant (simplified description)

Inputs:

- Throttle valve opening,
- Feed water pump pressure,
- BFW - boiler feed water (from Feed water heaters),
- Coal from pulverizers

Main Assumptions:

Coal flowrate is a function of the plant load, the coal HHV is fixed and heat duty from fire side to water wall and platen superheater are fixed.

Boiler heat exchanger network:

Water Flow: Fresh water -> FWH's -> Economizer -> Water Wall -> Primary SH -> Platen SH -> Finishing Superheater -> HP Turbine -> Reheater -> IP Turbine

Flue Gas Flow:

Fire Ball -> Platen SH -> Finishing SH -> Reheater -> o -> Economizer -> Air Preheater -> Primary SH -^

Steam Flow: Boiler -> HP Turbine -> Reheater -> IP Turbine -> Condenser HP, IP, and LP steam extractions to Feed Water Heaters

Main Models used:

- Mixers: Attenuator, Flue gas mix
- Heater: Platen SH, Fire/Water side (simplified model), Feed Water Heaters, Hot Tank, Condenser

BoilerHeatExchanger: Economizer, Primary SH, Finishing SH, Reheater

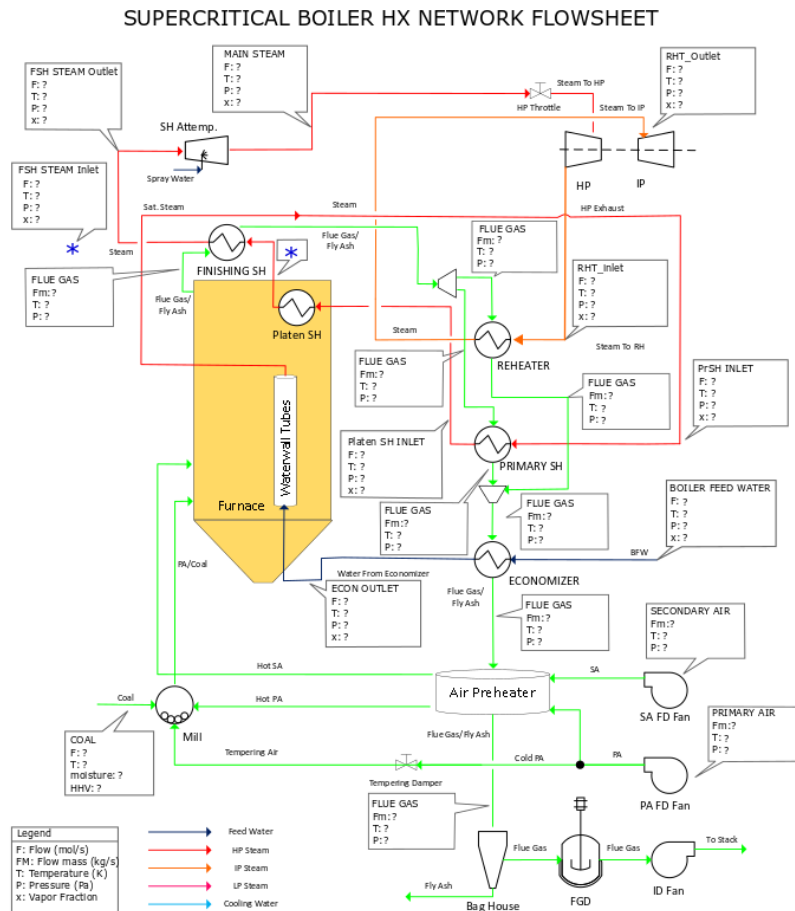
– Shell and tube heat exchanger

- * tube side: Steam (side 1 holdup)
- * shell side: flue gas (side 2 holdup)
- Steam Turbines
- Pumps

Property packages used:

- IAPWS: Water/steam side
- IDEAL GAS: Flue Gas side

Figures Process Flow Diagram:



Subcritical Coal-Fired Power Plant Flowsheet (steady state and dynamic)

Note: This is an example of a subcritical pulverized coal-fired power plant. This simulation model consists of a ~320 MW gross coal fired power plant, the dimensions and operating conditions used for this simulation do not represent any specific power plant. This model is for demonstration and tutorial purposes only.

Introduction

A 320 MW gross subcritical coal-fired power plant is modeled using the unit model library has been developed for demonstration purposes only. This plant simulation does not represent any power plant. This subcritical unit burns Illinois #6 high-volatile bituminous coal. The fuel is identical to the NETL baseline case for a 650 MW unit and its analysis data are listed in Table 1.

Table 1. coal specifications - Proximate Analysis (weight %)

item	As-Received	Dry
Moisture	11.12	0.00
Ash	9.70	10.91
Volatile Matter	34.99	39.37
Fixed Carbon	44.19	49.72
Total	100.00	100.0

Table 2. coal specifications - Ultimate Analysis (weight %)

item	As-Received	Dry
Moisture	11.12	0.00
Carbon	63.75	71.72
Hydrogen	4.50	5.06
Nitrogen	1.25	1.41
Chlorine	0.15	0.17
Sulfur	2.51	2.82
Ash	9.70	10.91
Oxygen	7.02	7.91
Total	100.00	100.0

Table 3. coal specifications - Heating Value

item	As-Received	Dry
Higher Heating Value (HHV), kJ/kg (Btu/lb)	27,113 (11,666)	30,506 (13,126)
Lower Heating Value (LHV), kJ/kg (Btu/lb)	26,151 (11,252)	29,544 (12,712)

The power plant is a generic subcritical unit, where the boiler has 4 levels of wall burners and one level of over-fire airports. There are 18 platen superheaters hanging over the furnace roof serving as the finishing superheater. The platen superheater panels are parallel to the furnace side walls. The boiler has one drum, eight downcomers, backpass superheater, platen superheater, a reheater section (represented with 2 heat exchanger model), a primary superheater, an economizer, and the air preheater (this model is a simplified tri-sector Ljungström type)

The steam cycle equipment includes a multistage steam turbine with single reheat. It has a throttle valve, multiple stages for HP, IP, and LP sections with steam extraction to 3 low-pressure feed water heaters and 2 high-pressure feed water heaters as well as a deaerator and a boiler feed pump turbine. The steam cycle also includes the main and auxiliary condensers, a hotwell tank, a condensate pump, a booster pump and a main pump. Multiple control valves are used to control the water levels of hotwell tank, deaerator tank, and feed water heaters and the spray flow to main steam attemperator. The process flow diagram is shown in Figure 1. The entire process is modeled in two sub-flowsheets, one for the boiler system and the other for the steam cycle system, corresponding to two separate files named “boiler_subfs.py” and “steam_cycle_subfs.py”, respectively.

The main flowsheet contains the two sub-flowsheets in a file named “plant_dyn.py”.

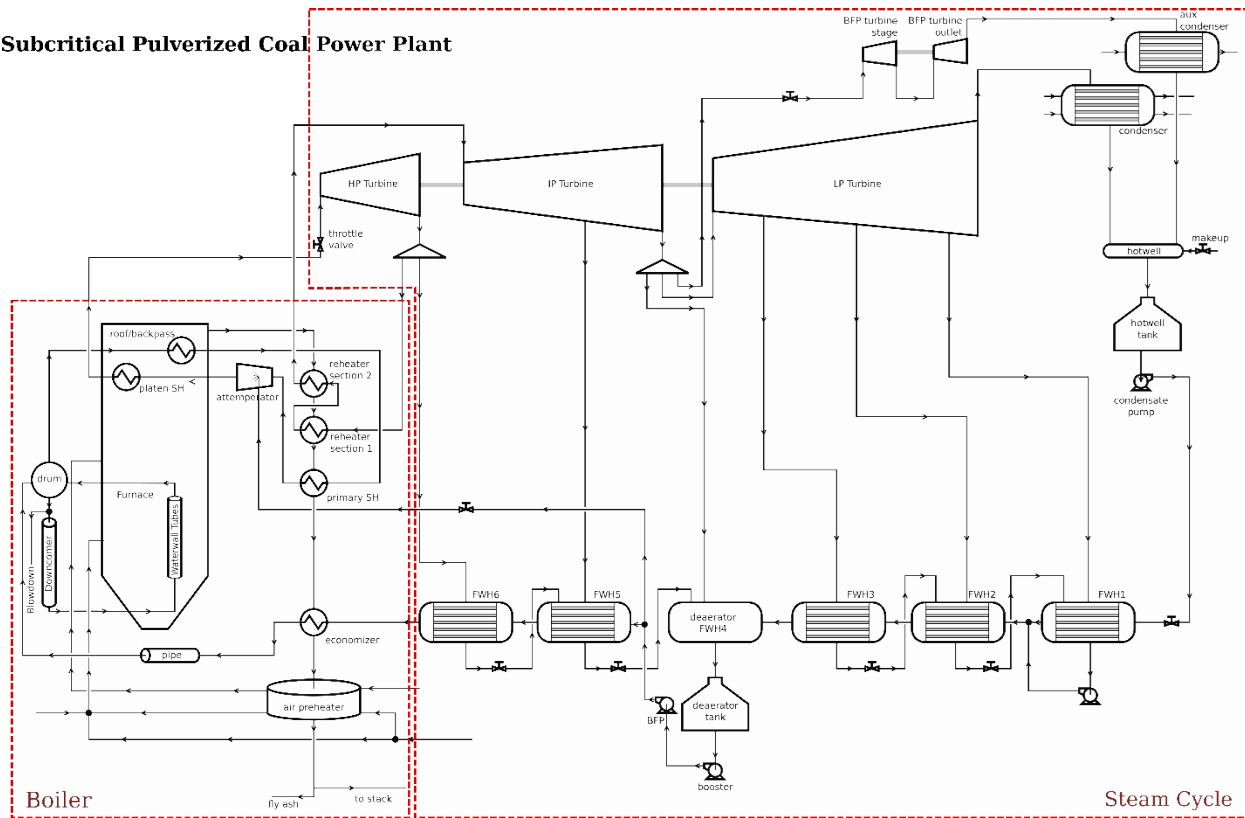
Figure 1: Process Flow Diagram

Property packages used:

- Helmholtz Equation of State (IAPWS95): Water and steam *IAPWS95*
- IDEAL GAS: Air and Flue Gas *Flue-Gas*

It can be seen from the Figure 1 that primary air is first split to two inlet streams, one goes through the primary air sector of the regenerative air preheater where it is heated, and the other,

Subcritical Pulverized Coal Power Plant



also known as tempering air, bypasses the air preheater and is used for primary air temperature control. The two streams are then mixed and connected with the fire side of the boiler. The coal stream is also fed to the fire side of the boiler. While this boiler sub-flowsheet does not contain a specific coal mill model, the partial vaporization of the moisture in the raw coal is modeled in the fire-side boiler model. The secondary air enters the secondary air sector of the air preheater. After being heated in the air preheater, the hot secondary air stream enters boiler's wind-box, from which it enters the furnace either as the secondary air of the burners or as overfire air. The pulverized coal from primary air stream is eventually burned in the boiler by both primary and secondary air to form flue gas that leaves the boiler with small amount of unburned fuel in fly ash. The hot flue gas then goes through the boiler backpass consisting of multiple convective heat exchangers including first the hot reheater, then the cold reheater, the primary superheater, and the economizer. Finally, the flue gas enters the air preheater to heat the cold primary air and secondary air before entering the downstream equipment which is not modeled. The feed water from steam cycle system enters the economizer to absorb the heat transferred from the flue gas and leaves the economizer at a temperature considerably below its saturation temperature. The subcooled water then goes to the boiler drum through water pipes where it mixes with the saturated water separated from the water/steam mixture from the boiler waterwall. The mixed water stream splits to two streams. A small amount of water leaves the system as a blowdown water to prevent buildup

of slag and the main portion of the water stream goes through eight downcomers to enter the bottom of waterwall tubes. The vertical waterwall is modeled by multiple waterwall section models in series. The subcooled water from the downcomers is heated by the combustion products inside the boiler and part of the liquid are vaporized, forming a liquid-vapor 2-phase mixture and eventually enters the drum to complete a circuit for natural circulation of the feed water, in which the density difference between the liquid in the downcomers and the 2-phase mixture in the waterwall tubes drives the circulating flow.

The saturated steam from drum goes to the roof superheater before entering the primary superheater. Note that the enclosure wall tubes for the backpass as a part of the superheaters is not included in the flowsheet model. The steam leaving the primary superheater is mixed with spray water from boiler main feed pump in an attemperator. Finally, the steam from the attemperator enters the platen superheater where the steam is heated to main steam temperature before entering the main turbine. The cold reheat steam from the HP outlet is first heated in the cold reheater and then heated in the hot reheater before entering the IP stages of the turbine. There is no attemperation for the reheat steam.

The steam cycle, depicted in Figure 1, accepts steam from the boiler and uses it to generate electricity. Specifically, HP steam from the boiler passes through a throttle valve prior to entering the HP turbine, after which some is extracted to supply heat to feedwater heater (FWH) 6 while the remaining steam is sent back to the boiler to be reheated. The reheated steam then enters the IP turbine section where some is extracted to supply heat to FWH 5. Additional steam is extracted for the boiler feed pump turbine (BFPT) and deaerator between the IP and LP turbine sections at the IP/LP crossover, while the remaining steam enters the LP turbine. In the LP section, steam is extracted for FWHs 1, 2, and 3. Steam leaving the LP turbine is condensed in the main condenser, while steam leaving the BFPT is condensed in the auxiliary condenser. The condensate from the condenser hotwell is pumped through the LP FWHs 1, 2, and 3, where it is heated by steam extracted from the LP turbine. After passing through the deaerator, the feedwater enters a booster pump prior to entering the main boiler feed pump and HP FWHs 5 and 6.

Dynamic Flowsheet

The dynamic model considers the mass and energy inventories in large vessels in the system including drum, deaerator, feed water heaters, and condenser hotwell. Meanwhile, inventories in downcomers and multiple water-wall zones are also considered. To keep the problem tractable, the inventory in the 2D boiler heat exchanger model for the reheaters, the primary superheater, and the economizer are not considered. However, the internal energy held by the tube metal of those heat exchangers are considered. Some unit models are treated as the steady-state models on the dynamic flowsheet. For example, the boiler fire-side model is assumed as steady-state since the flue gas density is low and so is the residence time (2-3 seconds). All turbine stages, condensers and pumps are modeled as pseudo-steady-state. The entire process is pressure driven, indicating the flow rates of air, flue gas, water, and steam are related to the pressures in the system. Table 4 lists all unit operation models on the boiler sub-flowsheet including their names, descriptions, unit model library names, and dynamic/steady-state flag. Table 5 lists all unit operations on the steam cycle sub-flowsheet.

Table 4. List of unit models on the boiler system sub-flowsheet

Unit Name	Description	Unit Library Name	Dynamic
aBoiler	Boiler fire-side surrogate	<i>BoilerFireside</i>	False
aDrum	1D boiler drum	<i>Drum1D</i>	True
blowdown_split	Splitter for blowdown	HelmSplitter	False
aDowncomer	Downcomer	<i>Downcomer</i>	True
Waterwalls	12 waterwall zones	<i>WaterwallSection</i>	True
aRoof	Roof superheater	<i>SteamHeater</i>	False
aPlaten	Platen superheater	<i>SteamHeater</i>	False
aRH1	2D Cold reheater	HeatExchangerCrossFlow2D_Header <i>HX2D</i>	True *
aRH2	2D Hot reheater	HeatExchangerCrossFlow2D_Header <i>HX2D</i>	True *
aPSH	2D Primary superheater	HeatExchangerCrossFlow2D_Header <i>HX2D</i>	True *
aECON	2D Economizer	HeatExchangerCrossFlow2D_Header <i>HX2D</i>	True *
aPipe	Pipes from eco. to drum	<i>WaterPipe</i>	False
Mixer_PA	Mixer of hot PA and TA	Mixer	False
Attemp	Attemperator	HelmMixer	False
aAPH	Air preheater	<i>HeatExchangerWith3Streams</i>	False

- The heat held by tube metal is modeled as dynamic while fluids are modeled as steady-state

Table 5. List of unit models on the

steam cycle system sub-flowsheet

Unit Name	Description	Unit Library Name	Dynamic
turb	Multistage turbine	<i>HelmTurbineMultistage</i>	False
bfp_turb_valve	BFPT regulating valve	<i>HelmValve</i>	False
bfp_turb	Front stage of BFPT	<i>HelmTurbineStage</i>	False
bfp_turb_os	Outlet stage of BFPT	<i>HelmTurbineOutletStage</i>	False
condenser	Main condenser	HelmNtuCondenser	False
aux_condenser	Auxiliary condenser	HelmNtuCondenser	False
condenser_hotwell	Mixer of 3 water streams	HelmMixer	False
makeup_valve	Makeup water valve	<i>HelmValve</i>	False
hotwell_tank	Hotwell tank	<i>WaterTank</i>	Dynamic
cond_pump	Condensate pump	HelmIsentropicCompressor	False
cond_valve	Condensate Valve	HelmValve	False
fwh1	Feed water heater 1	<i>FWH0D</i>	Dynamic
fwh1_drain_pump	Drain pump after FWH 1	HelmIsentropicCompressor	False
fwh1_drain_return	Mixer of drain and condensate	HelmMixer	False
fwh2	Feed water heater 2	<i>FWH0D</i>	Dynamic
fwh2_valve	Drain valve for FWH 2	<i>HelmValve</i>	False
fwh3	Feed water heater 3	<i>FWH0D</i>	Dynamic
Fwh3_valve	Drain valve for FWH 3	<i>HelmValve</i>	False
fwh4_deair	Mixer for deaerator	HelmMixer	False
da_tank	Deserator water tank	<i>WaterTank</i>	Dynamic
booster	Booster pump	HelmIsentropicCompressor	False
bfp	Main boiler feed pump	HelmIsentropicCompressor	False
split_attemp	Splitter for spray water	HelmSplitter	False
spray_valve	Control valve for water spray	<i>HelmValve</i>	False
Fwh5	Feed water heater 5	<i>FWH0D</i>	Dynamic
Fwh5_valve	Drain valve for FWH 5	<i>HelmValve</i>	False
Fwh6	Feed water heater 6	<i>FWH0D</i>	Dynamic
Fwh6_valve	Drain valve for FWH 6	<i>HelmValve</i>	False

- Dynamic flag is true for condensing section only

There are several control valves to regulate the water and steam flows in the steam cycle system including the throttle valve to control the power output, BFPT valve to control feed water pump speed and, therefore, the feed water flow, makeup water valve to control condenser hotwell tank level, condensate valve to control deaerator tank level, and water spray valve to control the main steam temperature. There are also drain valves between the drain outlet of a feed water heater and the drain inlet of its cascading downstream feed water heater. They are used to control the water level inside the condensing section of the feed water heater.

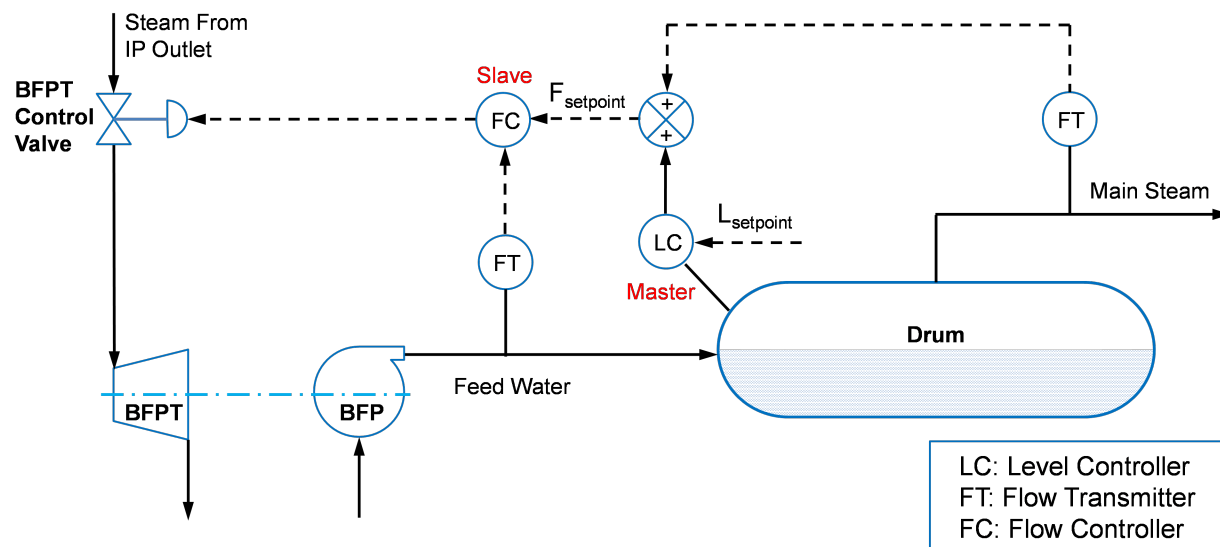
Controllers are included in the dynamic

flowsheet. There are several regulatory level controllers, all of which are either PID controllers, including those to control the levels of FWH2, FWH3, FWH5, FWH6, deaerator tank, and hotwell tank. The corresponding control valves involved in those level controllers are drain valves of FWH2, FWH3, FWH5, FWH6, the condensate valve, and the makeup water valve, respectively.

The main steam temperature is controlled by the attemperator with the spray water from the main feed pump outlet. The spray valve between the attemperator and the feed pump outlet is controlled by a PID controller that is also configured with default valve opening bounded between 0.05 and 1. This configuration limits the spray flow to certain values if the main steam temperature is way below or above its setpoint.

The drum level is controlled by a 3-element controller as shown in Figure 2. It is implemented with two cascading PI control loops. The master controller of the drum level cascading control is used to provide the setpoint of the slave controller. The slave controller controls the feed water flow rate based on the setpoint provided by the master controller. The three measured elements include drum level, main steam flow and feed water flow. Note that the main steam flow rate is usually controlled by a throttle valve such that the required power output is met. The feed water flow rate should be the same as the main steam flow rate in a steady-state condition. If the drum level is deviated from its setpoint, the feed water flow should be adjusted to compensate the level deviation. For example, when the drum level is too low, the feed water flow rate should be increased. Therefore, the setpoint of the feed water flow rate for the slave controller is equal to the sum of the measured main steam flow rate and the output of the master controller, which is the adjustment calculated by the master controller due to the deviation of the drum level from its setpoint. In case there is a drum water blowdown flow, the amount of blowdown flow should also be added. In the modeled power plant, the flow rate of feed water is controlled by the governing valve of the boiler feed pump turbine (BFPT), which controls the speed of the BFPT and hence the speed of the boiler feed pump (BFP). The BFPT uses the steam from the IP turbine outlet to generate the mechanical work needed for the BFP pump.

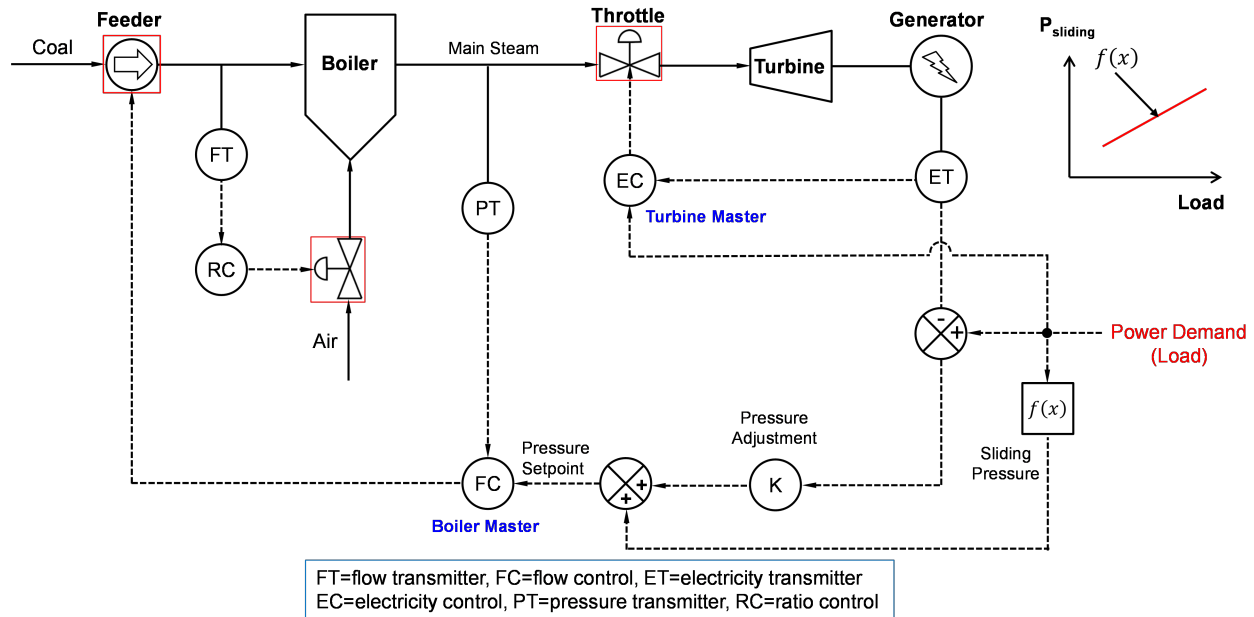
Figure 2: Drum Level Control



The supervisory level control for the power generation is implemented as a typical coordinated control as shown in Figure 3. It involves a turbine master that controls the throttle valve opening to meet the power demand and a boiler master that controls the coal feed rate and air flow rates. In the dynamic flow-sheet model, both the turbine master and the boiler master are implemented as PI controllers. The turbine master simply controls the power output to meet the power demand by adjusting the throttle valve opening. The boiler master controls the coal feed rate and air flow rates to maintain the desired main steam pressure. The setpoint for the main steam pressure is calculated as the sum of two parts. The first part is the desired steady-state sliding pressure as a function of load (sliding pressure curve as shown in the figure). In the current dynamic plant model, the sliding pressure is implemented as a linear function of power demand. The second part is the pressure adjustment term calculated based on the deviation of electrical power output from the demand multiplied by a gain factor. The second part represents the coordination between the turbine master and boiler master. Note that in the current dynamic flowsheet model, the control for the air flows is not implemented with detailed PID controllers for FD and ID fans and their dampers. The primary and secondary air flow rates are actually specified as constraints such that the mole fraction of O₂ in flue gas is set to be a predefined function of coal flow rate, which is related to the load. The primary air flow rate is specified as a constraint that specifies the primary air to coal flow ratio as a function of coal feed rate (mill curve). In other words, the primary and secondary air flow rates is controlled in proportion to the coal feed rate as in a typical ratio control loop.

Figure 3: Coordinated Control

Table 6 lists the PI and PID controllers



on the dynamic flowsheet. Some controllers are declared on the steam cycle sub-flowsheet while others are declared on the main flowsheet. The table also lists the type of the controller, whether it is bounded for output, and whether it belongs to the steam cycle or main flowsheet.

Table 6. Controllers in the dynamic flowsheet model

Unit Name	Description	Type	Bounded	Flowsheet
fwh2_ctrl	Level controller for FWH 2	PI	No	Steam cycle
Fwh3_ctrl	Level controller for FWH 3	PI	No	Steam cycle
Fwh5_ctrl	Level controller for FWH 5	PI	No	Steam cycle
Fwh6_ctrl	Level controller for FWH 6	PI	No	Steam cycle
da_ctrl	Level controller for deaerator tank	PI	No	Steam cycle
makeup_ctrl	Level controller for hotwell tank	PI	Yes	Steam cycle
spray	Main steam temperature controller	PID	Yes	Steam Cycle
drum_master_ctrl	Master controller for drum level	PI	No	Main
drum_slave_ctrl	Slave controller for drum level	PI	No	Main
turbine_master_ctrl	Turbine master controller	PI	No	Main
boiler_master_ctrl	Boiler master controller	PI	No	Main

Steady-state power plant example:

A steady state version of the power plant flowsheet is constructed by calling the `m_ss = main_steady_state()` method line 1824 in the `subcritical_power_plant.py` file. This function will build a steady state version of the power plant in Figure 1. This power plant model consists of two subflowsheets, the boiler subsystem (`m_ss.fs_main.fs_blr`) and the steam cycle subsystem (`m_ss.fs_main.fs_stc`). These two subsystems are connected through arcs at the flowsheet level. A custom initialization procedure has been developed, in which we initialize each subflowsheet at the time at a given load. After initializing the subflowsheet the example solves the entire power plant model for a given load (degrees of freedom = 0).

Main Fixed Variables:

- power demand
(`m_ss.fs_main.power_output.fix(320)`)
- main steam temperature
(`m.fs_main.fs_stc.temperature_main_steam.fix(810)` in Kelvin)
- water level (drum, deareator, condenser, feedwater heaters)
- equipment geometry/dimension
- fuel composition and HHV (on dry basis)

Main un-
fixed vari-
ables cal-
culated by
the model:
* Coal
flowrate
(`m.fs_main.fs_blr.aBoiler.flowrate_coal_raw`
is un-
fixed and
calculated
to match
the power
demand)
* wa-
ter/steam

```

flowrates
*
operator
flowrate
(m.fs_main.fs_stc.spray_valve.valve_opening.unfix())
free to keep main steam
at 810 K * Boiler feed-
water pump pressure (m.fs_main.fs_stc.bfp.outlet.pressure.unfix()) * throttle valve opening
(m.fs_main.fs_stc.turb.throttle_valve[1].valve_opening.unfix()) * primary air and secondary air flowrates (con-
strained by primary air to coal ratio and O2 mol fraction in the flue gas)

```

Dynamic power plant example:

The dynamic simulation version of the power plant examples is built when the user calls the `m_dyn = main_dynamic()` method in line 1820 in the `subcritical_power_plant.py` file. The user should note that this method takes a long time to solve (~60 min). This method builds and runs a subcritical coal-fired power plant dynamic simulation. The demonstration example prepared for this simulation consists of 5%/min ramping down from full load to 50% load, holding for 30 minutes and then ramping up to 100% load and holding for 20 minutes.

This method first creates a steady state version of the power plant, initializes the steady state model, and then uses this steady state model for initializing the dynamic model. Two dynamic flowsheets are constructed here, the main difference is that they have different time steps in the discretization domain. Dynamic flowsheet 1 uses a step size of 30 seconds and dynamic flowsheet 2 uses a time step of 60 seconds. This is useful to speed up the overall simulation time and to reduce the final number of variables and constraints. Note that the dynamic flowsheet 1 is used when the load is changing to capture the dynamic transient conditions of the plant change (i.e., while plant is ramping). While the dynamic flowsheet 2 is used when process is near steady state transient conditions. To simulate the dynamic case, this example implements a ramp function for the power demand and fixes the set point to match the power demand (see code below).

```
for_
↳t in m.fs_main.config.time:
    power_demand_
↳= input_profile(t0+t, x0)
    m.fs_main.turbine_
↳master_ctrl.setpoint[t].
↳value = power_demand
```

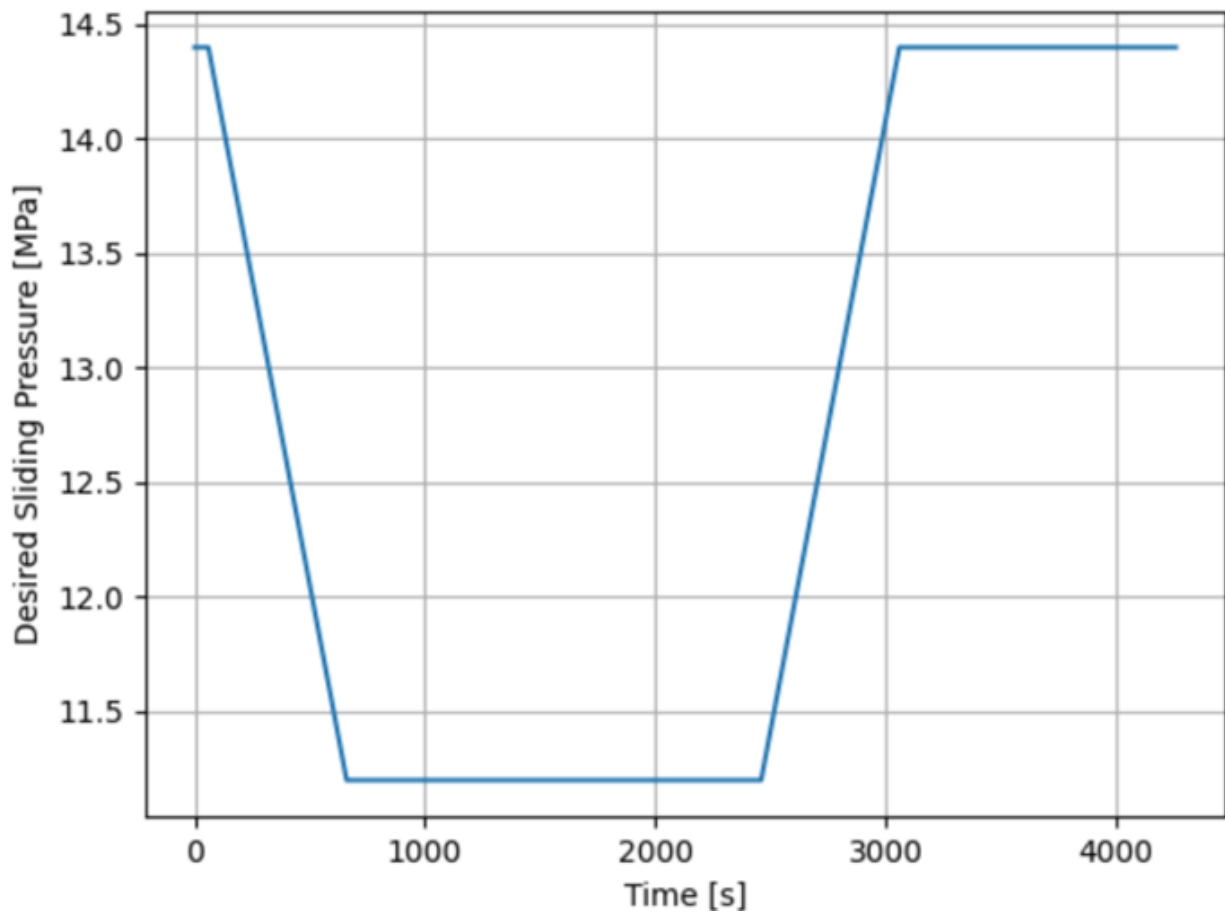
Solving the dynamic model:

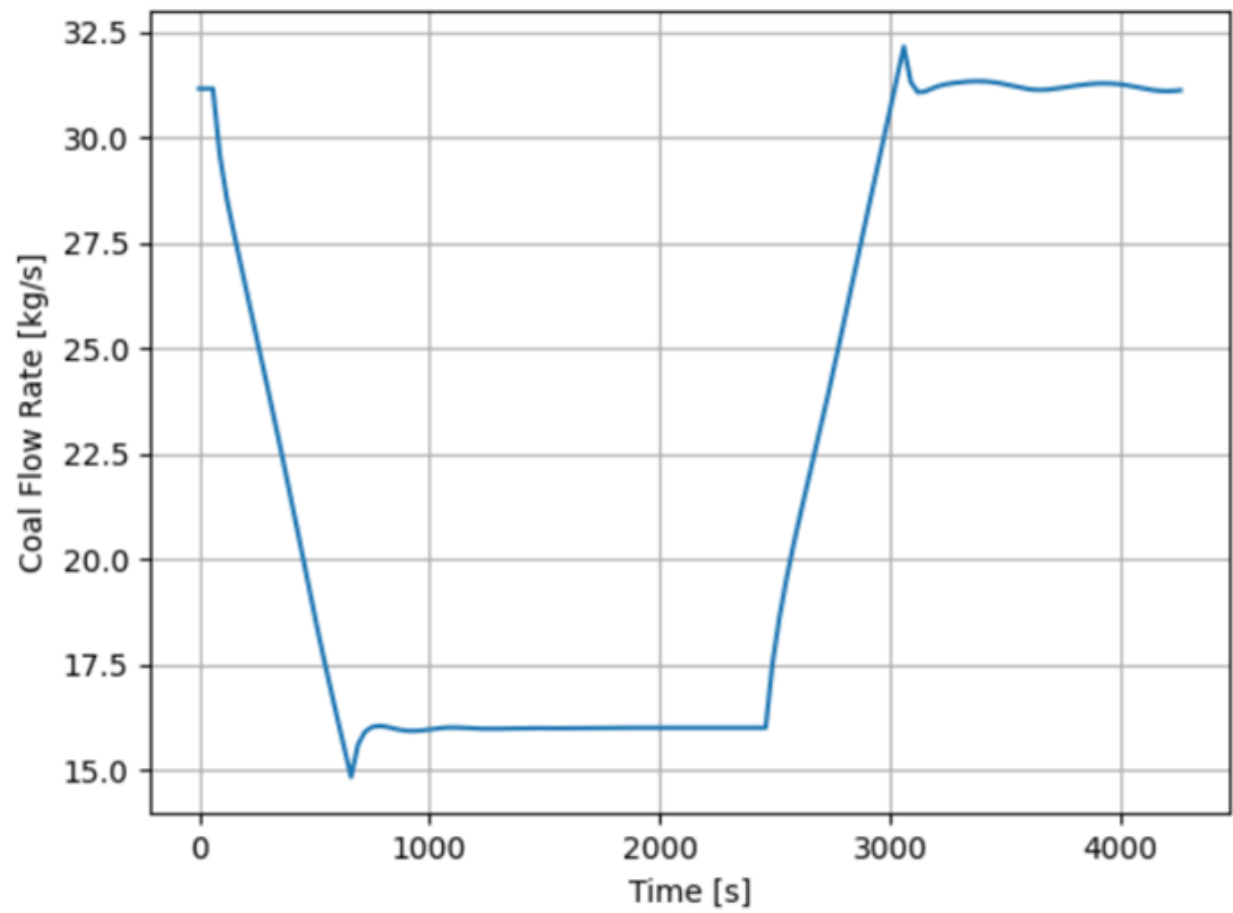
At this point we have a very large mathematical model, therefore, to exploit the temporal distribution of the model. The team implemented a rolling horizon approach (also known as receding horizon or moving time window), in which the full space model is divided into subproblems with 2 time periods each, then we solve the subproblem and use the solution of the previous subproblem to connect with the next time window (each time window consists of a dynamic model with 2 time periods). Thus, the dynamic model with 2 time steps is solved based on the disturbance of load demand specified by the user (power demand function described above). If the time duration for the simulation is longer than the period of the 2 time steps, the results of the solved dynamic model at the end of the second time step will be copied as the initial condition for the simulation of the next

2 time steps. The results to be copied include the errors for the integral and derivative parts of individual controllers. In case the error term for the integral part of a controller is too large (windup error), the user has an option to reset the windup error. If the time step size is changed, the user needs to choose a different dynamic model to copy to (dynamic flowsheet 1 or dynamic flowsheet 2). After that the time window is rolled to the second one and the simulation for the second time period can then be solved. This process is repeated for multiple time periods until the entire duration for the dynamic simulation is solved.

Note that during each rolling time window simulation, the results at individual time step for the main performance variables and equipment health are saved. Those results are plotted after the last simulation and written to a text file for review and further processing. Figure 4 shows an example of dynamic model simulation result, the load demand (ramp function) and coal feed rate for as functions of time.

Figure 4: Transient results for a load changing dynamic simulation





Power Plant Costing Library

Contents

- *Power Plant Costing Library*
- *Introduction*
- * *Costing sub-blocks*
- * *Capital Cost Stages*
- * *Dollar year scaling*
- *Power Plant Costing Module*
- * *Example*
- *Supercritical CO₂ Costing Module*
- *Other Costing Modules*
- * *Air Separation Unit*
- *Utility Functions*
- * *Initialize Costing*
- * *Total Flowsheet Cost Constraint*
- * *Display Total Flowsheet Cost*
- * *Display Individual Costs*
- * *Checking Bounds*
- *References*

Introduction

Note: The power plant costing method is available for most of the unit operations in power plants (Boiler, Feed Water Heaters, Compressor, Turbine, Condenser, etc.).

A capital cost methodology is developed in this module, both bare and erected cost and total plant cost are calculated based on costing correlations. The Power Plant Costing Library contains two main costing functions *get_PP_costing* and *get_SCO₂_unit_cost*. The first function (*get_PP_costing*) can be called to include cost correlations for equipment typically used in simulation of 7 technologies:

1. Supercritical pulverized coal plants (SCPC),

2. Subcritical pulverized coal plants,
3. Two-stage IGCC,
4. Single-stage IGCC,
5. Single-stage dry-feed IGCC,
6. natural gas air-fired plant (NGCC),
7. Advanced ultra-supercritical PC (AUSC).

Similarly, *get_sCO2_unit_cost* can be called to include cost correlations for equipment in supercritical CO2 power cycle plants.

Details are given for each method later in this documentation, however, there are many similarities between methods as discribed below:

Costing sub-blocks

In general, when *get_PP_costing* or *get_sCO2_unit_cost* is called on an instance of a unit model, a new sub-block is created on that unit named *costing* (i.e. *flowsheet.unit.costing*). All variables and constraints related to costing will be constructed within this new block (see detailed documentation for each unit for details on these variables and constraints).

Capital Cost Stages

There are multiple stages of capital cost, the lowest stage is the equipment cost which only includes the cost of manufacturing the equipment. The next stage is the bare erected cost (BEC) which includes the equipment cost and the cost of material and labor for installation. The final stage is the total plant cost (TPC) which includes the BEC plus the engineering fee, process contingency, and project contingency, all of which are typically estimated as a percentage of BEC.

$$\text{bare_erected_cost} = \text{equipment_cost} * (1 + \text{material_cost} + \text{labor_cost})$$

$$\text{total_plant_cost} = \text{bare_erected_cost} * (1 + \text{eng_fee} + \text{process_contingency} + \text{project_contingency})$$

Note: The equations above assume the additional costs (eng_fee or process and project contingencies) are given as percentages of BEC and TPC.

All costing methods calculate the bare erected and total plant costs. The sCO₂ library is currently the only one that includes an equipment cost.

Dollar year scaling

The value of money decreases over time due to inflation and missed investment opportunity. Thus, all costs must be normalized to the same dollar year to be compared on a consistent basis. This is done using a CE index and the following formula:

$$bare_erected_cost = bare_erected_cost_{base_year} * (CE_index / CE_index_{base_year})$$

In the costing functions this equation is built into the constraint for the lowest level capital cost in the selected method.

Table 1. Base years of costing modules

Module	Base Year
Power Plant Costing	2018
sCO ₂ Costing	2017
ASU	2011

The first time a ‘get costing’ function is called for a unit operation within a flowsheet, an additional *costing* block is created on the flowsheet object (i.e. *flowsheet.costing*) in order to hold any global parameters relating to costing. The most common of these parameters is the CE index parameter. The CE index will be set to the base year of the method called.

Note: The global parameters are created when the first instance of *get_costing* is called and use the values provided there for initialization. Subsequent *get_costing* calls use the existing parameters, and do not change the initialized values. i.e. any “year” argument provided to a *get_costing* call after the first will be ignored.

To manually set the dollar year the user must call `m.fs.get_costing(year=2019)` before any calls to a ‘get costing’ function are made.

Power Plant Costing Module

A default costing module has been developed based on the capital cost scaling methodology from NETL's Bituminous Baseline Report Rev 4 [1]. It provides costing correlations for common variants of pulverized coal (PC), integrated gassification combined cycle (IGCC), and natural gas combined cycle (NGCC) power generation technologies. Users should refer to reference [2] for details of the costing correlations, however, a summary is provided below.

The module breaks down the cost of a power plant into separate accounts for each system within the plant. The accounts are scaled based on a process parameter that determines the size of the equipment needed. The cost of the account is computed based on the scaled parameter, reference parameter, reference cost, and scaling exponent determined by NETL in [1]. This equation is similar to a six tenth factor approach, however, the exponents have been trained using several vendor quotes.

$$scaled_cost = reference_cost * (\frac{scaled_param}{reference_param})^\alpha$$

where:

- `scaled_cost` - the cost of the system in Million dollars
- `reference_cost` - the cost of the reference system in thousands of dollars
- `scaled_param` - the value of the system's process parameter
- `reference_param` - the value of the reference system's process parameter
- `alpha` - scaling exponent

Note: The capital cost scaling equation can be applied to any capital cost stage. In the power plant costing library it is applied to the bare erected cost, while in the sCO₂ library it is applied to the equipment cost.

The Power Plant costing method has five arguments, `self`, `cost_accounts`, `scaled_param`, `units`, and `tech`.

- `self` : an existing unit model or Pyomo Block
- `cost_accounts` : A list of accounts or a string containing the name of a pre-named account. If the input is a list all accounts must share the same process parameter. Pre-named accounts are listed below.
- `scaled_param` : The Pyomo Variable representing the accounts' scaled parameter
- `tech` : The technology to cost, different technologies have different accounts.

1. Supercritical PC,
2. Subcritical PC,
3. Two-stage, slurry-feed IGCC
4. Single-stage, slurry-feed IGCC
5. Single-stage, dry-feed IGCC,
6. Natural Gas Combined Cycle (NGCC),
7. Advanced Ultrasupercritical PC

- `units` : The user must pass a string with the units the `scaled_param` is in. It serves as a check to make sure the costing method is being used correctly.

Many accounts scale using the same process parameter. For convenience the user is allowed to enter accounts as a list instead of having to cost each account individually. If the accounts in the list do not use the same process parameter an error will be raised.

It is recognized that many users will be unfamiliar with the accounts in the Bituminous Baseline. For this reason the `cost_accounts` argument will also accept a string with the name of a pre-named account. Pre-named accounts aggregate the relevant accounts for certain systems. The pre-named accounts for each technology can be found in the tables below.

Table 2. Pre-named Accounts for PC technologies

Pre-named Account	Accounts Included	Process Parameter	Units
Coal Handling	1.1, 1.2, 1.3, 1.4, 1.9a	Coal Feed Rate	lb/hr
Sorbent Handling	1.5, 1.6, 1.7, 1.8, 1.9b	Limestone Feed Rate	lb/hr
Coal Feed	2.1, 2.2, 2.9a	Coal Feed Rate	lb/hr
Sorbent Feed	2.5, 2.6, 2.9b	Limestone Feed Rate	lb/hr
Feedwater System	3.1, 3.3	HP BFW Flow Rate	lb/hr
PC Boiler	4.9	HP BFW Flow Rate	lb/hr
Steam Turbine	8.1	Steam Turbine Power	kW
Condenser	8.3	Condenser Duty	MMBtu/hr
Cooling Tower	9.1	Cooling Tower Duty	MMBtu/hr
Circulating Water System	9.2, 9.3, 9.4, 9.6, 9.7	Circulating Water Flow Rate	gpm
Ash Handling	10.6, 10.7, 10.9	Total Ash Flow	lb/hr

Table 3. Pre-named Accounts for IGCC technologies

Pre-named Account	Accounts Included	Process Parameter	Units
Coal Handling	1.1, 1.2, 1.3, 1.4, 1.9	Coal Feed Rate	lb/hr
Coal Feed	2.1, 2.2, 2.9	Coal Feed Rate	lb/hr
Feedwater System	3.1, 3.3	HP BFW Flow Rate	lb/hr
Gasifier	4.1	Coal Feed Rate	lb/hr
Syngas Cooler	4.2	Syngas Cooler Duty	MMBtu/hr
ASU	4.3a	Oxygen Production	tpd
ASU Oxidant Compression	4.3b	Main Air Compressor Power	kW
Combustion Turbine	6.1, 6.3	Syngas Flowrate	lb/hr
Syngas Expander	6.2	Syngas Flowrate	lb/hr
HRSG	7.1, 7.2	HRSG Duty	MMBtu/hr
Steam Turbine	8.1	Steam Turbine Power	MW
Condenser	8.3	Condenser Duty	MMBtu/hr
Cooling Tower	9.1	Cooling Tower Duty	MMBtu/hr
Circulating Water System	9.2, 9.3, 9.4, 9.6, 9.7	Circulating Water Flow Rate	gpm
Slag Handling	10.1, 10.2, 10.3, 10.6, 10.7, 10.8, 10.9	Slag Production	lb/hr

Table 4. Pre-named Accounts for NGCC technologies

Pre-named Account	Accounts Included	Process Parameter	Units
Feedwater System	3.1, 3.3	HP BFW Flow Rate	lb/hr
Combustion Turbine	6.1, 6.3	Fuel Gas Flow	lb/hr
HRSG	7.1, 7.2	HRSG Duty	MMBtu/hr
Steam Turbine	8.1	Steam Turbine Power	kW
Condenser	8.3	Condenser Duty	MMBtu/hr
Cooling Tower	9.1	Cooling Tower Duty	MMBtu/hr
Circulating Water System	9.2, 9.3, 9.4, 9.6, 9.7	Circulating Water Flow Rate	gpm

The library has a 7th technology of AUSC. These operate at higher temperatures than traditional PC plants. The library contains modified correlation for the PC boiler, steam turbine, and steam

pipings to reflect the use of high temperature materials.

Table 5. Pre-named Accounts for AUSE technologies

Pre-named Account	Accounts Included	Process Parameter	Units
PC Boiler	4.9	HP BFW Flow Rate	lb/hr
Steam Turbine	8.1	Steam Turbine Power	kW
Steam Piping	8.4	HP BFW Flow Rate	lb/hr

A call to `get_PP_costing` creates two variables and two constraints for each account in the list. The variables are `bare_erected_cost` and `total_plant_cost`. Both variables are indexed by the account number in string format. The function makes a new block called `self.costing` where all variables and parameters associated with costing are stored.

Note: The `bare_erected_cost` and `total_plant_cost` are in Million dollars.

Example

Below is a simple example of how to add cost correlations to a flowsheet including a heat exchanger using the default IDAES costing module.

```
from pyomo.environ_
↳ import (ConcreteModel,
↳ SolverFactory)
from idaes_
↳ core import FlowsheetBlock
from idaes_
↳ generic_models.unit_models_
↳ heat_exchanger import \
    (HeatExchanger,
↳ HeatExchangerFlowPattern)
from idaes.generic_models_
↳ properties import iapws95
from idaes.power_
↳ generation.costing.power_
↳ plant_costing import \
    (get_PP_costing,
↳ initialize_costing,
↳ display_total_plant_costs,
↳ display_flowsheet_cost)

m = ConcreteModel()
m.fs_
↳ = FlowsheetBlock(default=
↳ {"dynamic": False})
```

(continues on next page)

(continued from previous page)

```

m.fs.properties = iapws95.
↳ Iapws95ParameterBlock()

m.fs.unit_
↳ = HeatExchanger(default={
    ↳
    ↳ "shell": {"property_
↳ package": m.fs.properties},
    ↳
    ↳ "tube": {"property_
↳ package": m.fs.properties},
    ↳ "flow_pattern":_
↳ HeatExchangerFlowPattern.
↳ countercurrent})
# set inputs
m.fs.unit.shell_inlet.flow_
↳ mol[0].fix(100) # mol/s
m.fs.unit.shell_inlet.enth_
↳ mol[0].fix(3500) # j/s
m.fs.unit.
↳ shell_inlet.pressure[0].
↳ fix(101325) # Pa

m.fs.unit.tube_
↳ inlet.flow_mol[0].fix(100)
m.fs.unit.tube_
↳ inlet.enth_mol[0].fix(4000)
m.fs.unit.tube_inlet.
↳ pressure[0].fix(101325.0)

m.fs.
↳ unit.area.fix(1000) # m2
m.fs.unit.overall_
↳ heat_transfer_coefficient.
↳ fix(100) # W/m2K

m.fs.unit.initialize()

m.fs.
↳ unit.duty_MMBtu = pyo.Var(
    m.fs.time,
    initialize=1e5,
    doc="Condenser_
↳ duty in MMBtu/hr")

@m.fs.
↳ unit.Constraint(m.fs.time)
def duty_conversion(b, t):
    conv_fact = 3.412/1e6
    return b.duty_MMBtu[t]_
↳ == conv_fact*b.heat_duty[t]

get_PP_costing(m.fs.unit,
↳ 'Condenser', m.fs.unit.
↳ duty_MMBtu, 'MMBtu/hr', 1)
# initialize_
↳ costing equations

```

(continues on next page)

(continued from previous page)

```

initialize_costing(fs)

opt = SolverFactory('ipopt')
opt.options = {'tol'
↳ ': 1e-6, 'max_iter': 50}
results_
↳ = opt.solve(m, tee=True)

display_total_plant_costs(fs)
display_flowsheet_cost(fs)

```

Supercritical CO2 Costing Module

The sCO₂ costing function, besides including the capital cost and engineering of the equipment, it can cost penalty due to the high temperature and pressure equipment required for sCO₂ PC plants. The function has five arguments, self, equipment, scaled_param, temp_C, and n_equip.

- self : an existing unit model or Pyomo Block
- equipment : The type of equipment to be costed, see table 6
- scaled_param : The Pyomo Variable representing the component's scaled parameter
- temp_C : The Pyomo Variable representing the hottest temperature of the piece of equipment being costed. Some pieces of equipment do not have a temperature associated with them, so the default argument is None.
- n_equip : The number of pieces of equipment to be costed. The function will evenly divide the scaled parameter between the number passed.

The equipment cost is calculated using the following two equations. A temperature correction factor account for advanced materials needed at high temperatures.

$$equipment_cost = reference_cost * (scaled_parameter)^\alpha * temperature_factor$$

$$temperature_factor = 1 + c * (T - T_{bp}) + d * (T - T_{bp})^2 : if T \geq T_{bp}$$

$$(if T > 550, otherwise temperature_factor = 1)$$

$$T_{bp} = 550C$$

The bare erected and total plant costs are calculated as shown in the introduction. There are currently no estimates for the total plant cost components, so bare erected cost will be the same as total plant cost for now.

Five variables and constraints are created within the costing block. Three are for the equipment, bare erected, and total plant costs. One is for the temperature correction factor. The last one is for the scaled parameter divided by `n_equip`.

Table 6. sCO₂ Costing Library Components

Component	Scaling Parameter	Units
Coal-fired heaters	Q	MW_{th}
Natural gas-fired heaters	Q	MW_{th}
Recuperators	UA	W/K
Direct air coolers	UA	W/K
Radial turbines	W_{sh}	MW_{sh}
Axial turbines	W_{sh}	MW_{sh}
IG centrifugal compressors	W_{sh}	MW_{sh}
Barrel type compressors	V_{in}	m^3/s
Gearboxes	W_e	MW_{sh}
Generators	W_e	MW_e
Explosion proof motors	W_e	MW_e
Synchronous motors	W_e	MW_e
Open drip-proof motors	W_e	MW_e

Other Costing Modules

Air Separation Unit

The ASU costing function calculates total plant cost in the exact same way as the `get_PP_costing` function. `get_ASU_cost` takes two arguments: `self`, and `scaled_param`.

- `self` - a Pyomo Block or unit model
- `scaled_param` - The scaled parameter. For the ASU it is the oxygen flowrate in units of tons per day.

Utility Functions

Initialize Costing

The `costing_initialization` function will initialize all the variable within every costing block in the flowsheet. It takes one argument, the flowsheet object. It should be called after all the calls to ‘get costing’ functions are completed.

The function iterates through the flow-sheet looking for costing blocks and calculates variables from constraints.

Total Flowsheet Cost Constraint

For optimization, a constraint summing all the total plant costs is required. Calling `build_flowsheet_cost_constraint(m)` creates a variable named `m.fs.flowsheet_cost` and builds the required constraint at the flowsheet level.

Note: The costing libraries can be used for simulation or optimization. For simulation, costing constraints can be built and solved after the flowsheet has been solved. For optimization, the costing constraints will need to be solved with the flowsheet.

Display Total Flowsheet Cost

Calling `display_flowsheet_cost(m)` will print the value of `m.fs.flowsheet_cost`.

Display Individual Costs

There are three functions for displaying individual costs.

- `display_total_plant_costs(fs)`
- `display_bare_erected_costs(fs)`
- `display_equipment_costs(fs)`

Each one prints out a list of the costed blocks and the cost level of the function chosen. The functions should be called after solving the model.

Checking Bounds

Currently, only the sCO₂ module has support for checking bounds.

All costing methods have a range, outside of which the correlations become inaccurate. Calling `check_sCO2_costing_bounds(fs)` will display which components are within the proper range and which are outside it. It should be called after the model is solved.

References

1. DOE/NETL-2015/1723 Cost and Performance Baseline for Fossil Energy Plants Volume 1a: Bituminous Coal (PC) and Natural Gas to Electricity Revision 3 and 4
2. DOE/NETL-341/013113 Quality Guidelines for Energy System Studies Capital Cost Scaling Methodology
3. NETL_PUB_21490 Techno-economic Evaluation of Utility-Scale Power Plants Based on the Indirect sCO₂ Brayton Cycle. Charles White, David Gray, John Plunkett, Walter Shelton, Nathan Weiland, Travis Shultz. September 25, 2017
4. sCO₂ Power Cycle Component Cost Correlations from DOE Data Spanning Multiple Scales and Applications. Nathan Weiland, Blake Lance, Sandeep Pidaparti. Proceedings of ASME Turbo Expo 2019: Turbomachinery Technical Conference and Exposition GT2019. June 17-21, 2019, Phoenix, Arizona, USA

Proportional-Integral-Derivative (PID) Controller

The IDAES framework contains a basic PID control implementation, this section describes the dynamic power plant PID controller model.

Introduction

The PID controller model represents a PID controller in a feedback control loop of a process plant. Depending on the user specified configuration, this model can be configured as a proportional only (P), proportional and integral (PI), proportional and derivative (PD), or proportional, integral and derivative (PID) controller. When declaring a controller model, the user needs to specify the model type through the configuration option “type”, the process variable to be controlled $y(t)$ through the configuration option “pv”, and the manipulated variable $u(t)$ through the configuration option “mv”. The “pv” and “mv” variables can be any of the time-indexed variable on a dynamic flowsheet. The setpoint of the process variable $r(t)$ is a variable declared inside the model named as “setpoint”, which is usually fixed or specified as a function of other process variables. The variable or expression for the error $e(t)$ is defined as the setpoint minus the process variable to be controlled.

$$e(t) = r(t) - y(t)$$

Since the proportional part exists in any type of controllers, the gain for the proportional part $K_p(t)$ named as “gain_p” is always included as a variable in the model. Note that the gain is declared as a time-indexed variable since it could be changed from time to time if a gain scheduling approach is applied. If the model contains the integral part (for a PI or PID type controller), the user needs to specify the gain for the integral part $K_i(t)$ named as “gain_i”. A variable named “integral_of_error” is also declared inside the model for the integral error $e_i(t)$ defined as

$$e_i(t) = \int_0^t e(t') dt'$$

Since Pyomo.DAE does not provide a direct method to calculate the integral of a variable for the discretized equations, the PID controller model declares the integral term as a regular variable and a constraint that sets the derivative of the integral term with respect to time as the error term $e(t)$

$$\frac{de_i(t)}{dt} = e(t)$$

If the model contains the derivative part (for a PD or PID type controller), the user needs to specify the gain for the derivative part $K_d(t)$ named as “gain_d”. A derivative variable named “derivative_of_error” is also declared inside the model for the derivative error $e_d(t)$ defined as

$$e_d(t) = \frac{de(t)}{dt}$$

Pyomo.DAE provides a method to calculate the derivative error using its “DerivativeVar” declaration for the discretized equations.

For a general PID controller, the deviation of the manipulated variable from its

steady-state value $u'(t)$ can be expressed as

$$u'(t) = K_p e(t) + K_i e_i(t) + K_d e_d(t)$$

Note that the terms for the integral and derivative part on the right side of the equation could be zero depending on the controller type. The actual output for the manipulated variable $u(t)$ is calculated as the sum of the deviation term $u'(t)$ and the reference value also known as steady-state bias u_{ref}

$$u(t) = u'(t) + u_{ref}$$

Note that the steady-state bias is reference value that is not time-indexed in the current model. To account for an actuator saturation condition, the calculated manipulated variable $u(t)$ can optionally be clamped within a range between its lower and upper bounds. For example, a control valve cannot close to less than 0% or open to more than 100%. To declare this option, the configuration option “bounded_output” is set to True. With this option turned on, the user needs to set the value for two mutable parameters. Parameter “mv_lb” is for the lower bound and parameter “mv_ub” is for the upper bound. If they are not set by the user, the default values of 0.05 and 1 for the lower and upper bounds will be used as defaults, respectively. Different clamping function has been tried to make the output of the manipulated variable $u(t)$ within the lower and upper bounds. Since Pyomo model requires all functions to be smooth and differentiable, current PID controller model uses a sigmoid function as the clamping function. If the lower and upper bounds of u are u_l and u_u , respectively, the sigmoid function is defined as

$$f(u) = u_l + \frac{u_u - u_l}{1 + \exp\left[-\frac{4(u - \frac{u_l + u_u}{2})}{u_u - u_l}\right]}$$

where u is the calculated manipulated variable before the clamping function is applied and $f(u)$ is the actual output value for the manipulated variable. This function has the following properties:

$$\begin{aligned} f(-\infty) &= u_l \\ f(\infty) &= u_u \\ f\left(\frac{u_l + u_u}{2}\right) &= \frac{u_l + u_u}{2} \\ \frac{df}{du}\left(\frac{u_l + u_u}{2}\right) &= 1 \end{aligned}$$

Certainly, it is also smooth and differentiable. One disadvantage of the function is that when all error terms are zero ($e(t)=e_i(t)=e_d(t)=0$), the final output for the manipulated variable is not steady-state bias unless the bias is the same as the average of the lower and upper bound values.

$$f(u) \neq U_{ref}$$

For a PI controller, if $e(t)=0$, to make $f(u)=u_{ref}$, the integral error $e_i(t)$ should be set to a non-zero value, which can be calculated as

$$e_{(i,o)}(t) = \frac{1}{K_i} \left[\frac{u_l + u_u}{2} - U_{ref} - \frac{u_u - u_l}{4} \ln\left(\frac{u_u - u_l}{u_{ref} - u_l} - 1\right) \right]$$

The model provides an expression named “integral_of_error_ref” for the above equation. There is a similar expression named “integral_of_error_mv” to calculate the required integral error for a given $f(u)$ value when $e(t)=0$. While current clamping function is quite robust in solving the dynamic system with PID controllers, a better clamping function is still under development. If a manipulated variable is unlikely to reach saturation, it is recommended to disable the clamping option. It needs to be mentioned that the integral error in an PID controller could causes the wind-up issue if it gets larger and larger, especially for a slow process. To reset the wind-up at certain time, the dynamic simulation can be performed in multiple discretized time periods and the integral error as a variable can be reset to zero or other small values after solving one time period and before solving a subsequent time period. The PID controller model should be declared only for a dynamic flowsheet. It needs to be mentioned that when a dynamic flowsheet is discretized in a time domain, the calculation for the manipulated variable at the initial time is skipped. The user should provide the initial condition for the manipulated variable. Also note that the current implementation of the PID controller model ignore the measurement delay for the process variable. The current model simply provides continuous equations for the controller model, which is solved by the discretization through Pyomo.DAE. This is different from the actual PID installed at a plant where the controller calculates the current maneuver based on the measured process variable from the previous time step.

Gas Solid Contactors Model Library

This specialized IDAES application library contains a suite of generic advanced models that are applicable to gas-solid processes. The axially discretized models are one dimensional, with two phases (gas and solid).

Gas Solid Contactors Flowsheets

Contents

ss_BFB_methane_combustion

Gas Solid Contactors Unit Models

Contents

Bubbling Fluidized Bed Reactor

The IDAES Bubbling Fluidized Bed Reactor (BFBR) model represents a unit operation where two material streams, a solid phase and a gas phase, pass through a linear vessel while undergoing chemical reaction(s). The BFBR model is represented as a 1-D axially discretized model with two phases (gas and solid), and two regions (bubble and emulsion). The model captures the gas-solid interaction between both phases and regions through reaction, mass and heat transfer.

Assumptions:

- Cloud-wake region effects are negligible and are not modelled.
- Gas emulsion is at minimum fluidization conditions.
- Gas feeds into emulsion region before the excess enters into the bubble region.
- Gas and solids are well mixed in the radial direction but vary axially.

Requirements:

- Property package contains temperature and pressure variables.
- Property package contains minimum fluidization velocity and voidage parameters.

The BFBR model equations are derived from:

- A. Lee, D.C. Miller. A one-dimensional (1-D) three-region model for a bubbling fluidized-bed adsorber, Ind. Eng. Chem. Res. 52 (2013) 469–484.

Degrees of Freedom

BFBRs generally have at least 3 (or more) degrees of freedom, consisting of design and operating variables. The design variables of reactor length, diameter and number of orifices in the distributor are typically the minimum variables to be fixed.

Model Structure

The core BFBR unit model consists of two inlet ports (named `gas_inlet` and `solid_inlet`), two outlet ports (named `gas_outlet` and `solid_outlet`), and three `ControlVolume1DBlock` Blocks (named `bubble_region`, `gas_emulsion_region` and `solid_emulsion_region`).

Construction Arguments

The IDAES BFBR model has construction arguments specific to the whole unit and to the individual regions.

Arguments that are applicable to the BFBR unit as a whole are:

- `finite_elements` - sets the number of finite elements to use when discretizing the spatial domains (default = 10).
- `length_domain_set` - sets the list of point to use to initialize a new ContinuousSet (default = [0.0, 1.0]).
- `transformation_method` - sets the discretization method to use by the Pyomo TransformationFactory to transform the spatial domain (default = `dae.finite_difference`):
 - `dae.finite_difference` - finite difference method.
 - `dae.collocation` - orthogonal collocation method.
- `transformation_scheme` - sets the scheme to use when transforming a domain. Selected schemes should be compatible with the `transformation_method` chosen (default = None):
 - None - defaults to “BACKWARD” for finite difference transformation method and to “LAGRANGE-RADAU” for collocation transformation method
 - BACKWARD - use a finite difference transformation method.
 - FORWARD - use a finite difference transformation method.
 - LAGRANGE-RADAU - use a collocation transformation method.
- `collocation_points` - sets the number of collocation points to use when discretizing the spatial domains (default = 3, collocation methods only).
- `flow_type` - indicates the flow arrangement within the unit to be modeled. Options are:
 - ‘co-current’ - (default) gas and solid streams both flow in the same direction (from $x=0$ to $x=1$)

- 'counter-current' - gas and solid streams flow in opposite directions (gas from $x=0$ to $x=1$ and solid from $x=1$ to $x=0$).
- `material_balance_type` - indicates what type of energy balance should be constructed (default = `MaterialBalanceType.componentTotal`).
- `MaterialBalanceType.componentTotal` - use total component balances.
- `MaterialBalanceType.total` - use total material balance.
- `energy_balance_type` - indicates what type of energy balance should be constructed (default = `EnergyBalanceType.enthalpyTotal`).
- `EnergyBalanceType.none` - excludes energy balances.
- `EnergyBalanceType.enthalpyTotal` - single enthalpy balance for material.
- `momentum_balance_type` - indicates what type of momentum balance should be constructed (default = `MomentumBalanceType.pressureTotal`).
- `MomentumBalanceType.none` - exclude momentum balances.
- `MomentumBalanceType.pressureTotal` - single pressure balance for material.
- `has_pressure_change` - indicates whether terms for pressure change should be constructed (default = `True`).
- `True` - include pressure change terms.
- `False` - exclude pressure change terms.

Arguments that are applicable to the gas phase:

- `property_package` - property package to use when constructing bubble region Property Blocks (default = 'use_parent_value'). This is provided as a Physical Parameter Block by the Flowsheet when creating the model. If a value is not provided, the ControlVolume Block will try to use the default property package if one is defined.
- `property_package_args` - set of arguments to be passed to the bubble region

Property Blocks when they are created (default = 'use_parent_value').

- `reaction_package` - reaction package to use when constructing bubble region Reaction Blocks (default = None). This is provided as a Reaction Parameter Block by the Flowsheet when creating the model. If a value is not provided, the ControlVolume Block will try to use the default property package if one is defined.
- `reaction_package_args` - set of arguments to be passed to the bubble region Reaction Blocks when they are created (default = None).
- `has_equilibrium_reactions` - sets flag to indicate if terms of equilibrium controlled reactions should be constructed (default = False).

Arguments that are applicable to the solid phase:

- `property_package` - property package to use when constructing bubble region Property Blocks (default = 'use_parent_value'). This is provided as a Physical Parameter Block by the Flowsheet when creating the model. If a value is not provided, the ControlVolume Block will try to use the default property package if one is defined.
- `property_package_args` - set of arguments to be passed to the bubble region Property Blocks when they are created (default = 'use_parent_value').
- `reaction_package` - reaction package to use when constructing bubble region Reaction Blocks (default = None). This is provided as a Reaction Parameter Block by the Flowsheet when creating the model. If a value is not provided, the ControlVolume Block will try to use the default property package if one is defined.
- `reaction_package_args` - set of arguments to be passed to the bubble region Reaction Blocks when they are created (default = None).
- `has_equilibrium_reactions` - sets flag to indicate if terms of equilibrium con-

trolled reactions should be constructed (default = False).

Additionally, BFBR units have the following construction arguments which are passed to all the ControlVolume1DBlock Blocks and are always specified to their default values.

Argument	Default Value
dynamic	useDefault
has_holdup	useDefault

Constraints

Geometric Constraints

Area of orifice:

$$A_{or} = \frac{1}{n_{or}}$$

Bed cross-sectional area:

$$A_{bed} = \pi \frac{D_{bed}^2}{4}$$

Area of bubble region:

$$A_{b,t,x} = \delta_{t,x} A_{bed}$$

Area of gas emulsion region:

$$A_{ge,t,x} = \delta_{e,t,x} \varepsilon_{e,t,x} A_{bed}$$

Area of solid emulsion region:

$$A_{se,t,x} = \delta_{e,t,x} (1 - \varepsilon_{e,t,x}) A_{bed}$$

Length of bubble region:

$$L_b = L_{bed}$$

Length of gas emulsion region:

$$L_{ge} = L_{bed}$$

Length of solid emulsion region:

$$L_{se} = L_{bed}$$

Hydrodynamic Constraints

Emulsion region volume fraction:

$$\delta_{e,t,x} = 1 - \delta_{t,x}$$

Average cross-sectional voidage:

$$\varepsilon_{t,x} = 1 - (1 - \varepsilon_{e,t,x})(1 - \delta_{t,x})$$

Emulsion region voidage:

$$\varepsilon_{e,t,x} = \varepsilon_{mf,se}$$

Bubble growth coefficient:

$$\gamma_{t,x} = \frac{0.0256}{v_{mf,se}} \left(\frac{D_{bed}}{g} \right)^{0.5}$$

Maximum bubble diameter:

$$d_{bm,t,x}^5 g = 2.59^5 ([v_{g,t,x} - v_{ge,t,x}] A_{bed})^2$$

Bubble diameter (gas inlet, $x = 0$):

$$d_{b,t,x} = 1.38 g^{-0.2} ([v_{g,t,x} - v_{ge,t,x}] A_{or})^{0.4}$$

Bubble diameter ($x > 0$):

$$\frac{dd_{b,t,x}}{dx} = \frac{0.3}{D_{bed}} L_{bed} (d_{bm,t,x} - d_{b,t,x} - \gamma_{t,x} (D_{bed} d_{b,t,x})^{0.5})$$

Bubble rise velocity:

$$v_{br,t,x}^2 = 0.711^2 g d_{b,t,x}$$

Bubble velocity:

$$v_{b,t,x} = v_{g,t,x} - v_{mf,se} + v_{br,t,x}$$

Average gas density:

$$C_{g,t,x} = \frac{F_{mol,b,t,x} C_{b,total,t,x} + F_{mol,ge,t,x} C_{ge,total,t,x}}{F_{mol,b,t,x} + F_{mol,ge,t,x}}$$

Superficial gas velocity:

$$v_{g,t,x} = \frac{F_{mol,b,t,x} + F_{mol,ge,t,x}}{A_{bed} C_{g,t,x}}$$

Bubble volume fraction:

$$\delta_{t,x} v_{b,t,x} = v_{ge,t,x} \delta_{e,t,x} - v_{g,t,x}$$

Gas emulsion pressure drop:

if 'has_pressure_change' is 'True':

$$\Delta P_{ge,t,x} = -g(1 - \varepsilon_{e,t,x}) \rho_{mass,se,t,x}$$

elif 'has_pressure_change' is 'False':

$$P_{ge,t,x} = P_{g,t,inlet}$$

Mass Transfer Constraints

Bubble to emulsion gas mass transfer coefficient:

$$K_{be,t,x,j}d_{b,t,x}^{1.25} = 5.94v_{mf,se}d_{b,t,x}^{0.25} + 5.85D_{vap,ge,t,x,j}g^{0.25}$$

Bulk gas mass transfer:

if $C_{ge,total,t,x} > C_{b,total,t,x}$:

$$K_{gbulk,t,x,j} = 6K_d\delta_{t,x}A_{bed}(C_{ge,total,t,x} - C_{b,total,t,x})d_{b,t,x}y_{ge,t,x,j}$$

else:

$$K_{gbulk,t,x,j} = 6K_d\delta_{t,x}A_{bed}(C_{ge,total,t,x} - C_{b,total,t,x})d_{b,t,x}y_{b,t,x,j}$$

Heat Transfer Constraints

Bubble to emulsion gas heat transfer coefficient:

$$H_{be,t,x,j}d_{b,t,x}^{1.25} = 4.5v_{mf,se}c_{p_vap,b,t,x}C_{b,total,t,x}d_{b,t,x}^{0.25} + 5.85(k_{vap,b,t,x}C_{b,total,t,x}c_{p_vap,b,t,x})^{0.5}g^{0.25}$$

Convective heat transfer coefficient:

$$h_{tc,t,x}d_{p,se} = 0.03k_{vap,e,t,x}\left(v_{ge,t,x}d_{p,se}\frac{C_{ge,total,t,x}}{\mu_{vap,ge,t,x}}\right)^{1.3}$$

Emulsion region gas-solids convective heat transfer:

$$h_{t_gs,t,x}d_{p,se} = 6\delta_{e,t,x}(1 - \varepsilon_{e,t,x})h_{tc,t,x}(T_{ge,t,x} - T_{se,t,x})$$

Bulk gas heat transfer:

if $C_{ge,total,t,x} > C_{b,total,t,x}$:

$$H_{gbulk,t,x} = K_d\delta_{t,x}A_{bed}(C_{ge,total,t,x} - C_{b,total,t,x})d_{b,t,x}H_{ge,t,x}$$

else:

$$H_{gbulk,t,x} = K_d\delta_{t,x}A_{bed}(C_{ge,total,t,x} - C_{b,total,t,x})d_{b,t,x}H_{b,t,x}$$

Mass and heat transfer terms in control volumes

Bubble mass transfer '(p=vap)':

$$M_{tr,b,t,x,p,j} = K_{gbulk,t,x,j} - A_{b,t,x}K_{be,t,x,j}(C_{b,total,t,x} - C_{ge,total,t,x})$$

Gas emulsion mass transfer '(p=vap)':

$$M_{tr,ge,t,x,p,j} = -K_{gbulk,t,x,j} + A_{b,t,x}K_{be,t,x,j}(C_{b,total,t,x} - C_{ge,total,t,x}) + r_{hetero,ge,t,x,j}$$

if 'energy_balance_type' is not 'EnergyBalanceType.none':

Bubble heat transfer:

$$H_{tr,b,t,x} = H_{gbulk,t,x} - A_{b,t,x} H_{be,t,x,j} (T_{b,t,x} - T_{ge,t,x})$$

Gas emulsion heat transfer:

$$H_{tr,ge,t,x} = -H_{gbulk,t,x} + A_{b,t,x} H_{be,t,x,j} (T_{b,t,x} - T_{ge,t,x}) - h_{t_{gs},t,x} A_{bed}$$

Solid emulsion heat transfer:

$$H_{tr,se,t,x} = h_{t_{gs},t,x} A_{bed}$$

Reaction constraints

if 'homogeneous reaction package' is
not 'None':

Bubble rate reaction extent:

$$r_{ext,b,t,x,r} = A_{b,t,x} r_{b,t,x,r}$$

Gas emulsion rate reaction extent:

$$r_{ext,ge,t,x,r} = A_{ge,t,x} r_{ge,t,x,r}$$

if 'heterogeneous reaction package' is
not 'None':

Solid emulsion rate reaction extent:

$$r_{ext,se,t,x,r} = A_{se,t,x} r_{se,t,x,r}$$

Gas emulsion heterogeneous rate reaction extent:

$$r_{hetero,ge,t,x,j} = A_{se,t,x} \sum_r^{reactions} s_{se,j,r} r_{se,t,x,r}$$

Flowrate constraints

Bubble gas flowrate:

$$F_{mol,b,t,x} = A_{bed} \delta_{t,x} v_{b,t,x} C_{b,total,t,x}$$

Emulsion gas flowrate:

$$F_{mol,ge,t,x} = A_{bed} v_{ge,t,x} C_{ge,total,t,x}$$

Inlet boundary conditions

if 'has_pressure_change' is 'True':

Gas emulsion pressure at inlet:

$$P_{ge,t,0} = P_{g,t,inlet} - \Delta P_{or}$$

Total gas balance at inlet:

$$F_{mol,b,t,0} + F_{mol,ge,t,0} = F_{mol,g,t,inlet}$$

Solid particle porosity at inlet:

$$\phi_{se,t,0} = \phi_{t,inlet}$$

Gas emulsion velocity at inlet:

$$v_{ge,t,0} = v_{mf,se}$$

Bubble mole fraction at inlet:

$$y_{b,t,0,j} = y_{g,t,inlet,j}$$

Gas emulsion mole fraction at inlet:

$$y_{ge,t,0,j} = y_{g,t,inlet,j}$$

Solid emulsion mass flow at inlet:

if 'flow_type' is 'co_current' $x = 0$ else
if 'flow_type' is 'counter_current' $x = 1$:

$$F_{mass,se,t,x} = F_{mass,s,t,inlet}$$

Solid emulsion mass fraction at inlet:

if 'flow_type' is 'co_current' $x = 0$ else
if 'flow_type' is 'counter_current' $x = 1$:

$$x_{se,t,x} = x_{s,t,inlet}$$

if 'energy_balance_type' is not 'EnergyBalanceType.none':

Gas inlet energy balance:

$$H_{b,t,0} + H_{ge,t,0} = H_{g,t,inlet}$$

Gas emulsion temperature at inlet:

$$T_{ge,t,0} = T_{g,t,inlet}$$

elif 'energy_balance_type' is 'EnergyBalanceType.none':

Isothermal bubble region:

$$T_{b,t,x} = T_{g,t,inlet}$$

Isothermal gas emulsion region:

$$T_{ge,t,x} = T_{g,t,inlet}$$

Isothermal solid emulsion region:

$$T_{se,t,x} = T_{s,t,inlet}$$

if 'flow_type' is 'co_current' $x = 0$ else
 if 'flow_type' is 'counter_current' $x = 1$:

Solid inlet energy balance:

$$H_{se,t,x} = H_{s,t,inlet}$$

Outlet boundary conditions

Gas emulsion pressure at outlet:

$$P_{g,t,outlet} = P_{ge,t,1}$$

Total gas balance at outlet:

$$F_{mol,g,t,outlet} = F_{mol,b,t,1} + F_{mol,ge,t,1}$$

Solid outlet material balance:

if 'flow_type' is 'co_current' $x = 1$ else
 if 'flow_type' is 'counter_current' $x = 0$:

$$F_{mass,s,t,outlet} = F_{mass,se,t,x}$$

Solid particle porosity at outlet:

$$\phi_{t,outlet} = \phi_{se,t,1}$$

if 'energy_balance_type' is not 'EnergyBalanceType.none':

Gas outlet energy balance:

$$H_{g,t,outlet} = H_{b,t,1} + H_{ge,t,1}$$

Solid outlet energy balance:

if 'flow_type' is 'co_current' $x = 1$ else
 if 'flow_type' is 'counter_current' $x = 0$:

$$H_{s,t,outlet} = H_{se,t,x}$$

elif 'energy_balance_type' is 'EnergyBalanceType.none':

Gas outlet energy balance:

$$T_{g,t,outlet} = T_{ge,t,1}$$

Solid outlet energy balance:

if 'flow_type' is 'co_current' $x = 1$ else
 if 'flow_type' is 'counter_current' $x = 0$:

$$T_{s,t,outlet} = T_{se,t,x}$$

Variables

List of variables in the BFBR model:

Variable	Name	Notes
L_{bed}	bed_height	Bed height
D_{bed}	bed_diameter	Reactor diameter
A_{bed}	bed_area	Reactor cross-sectional area
A_{or}	area_orifice	Distributor plate area per orifice
n_{or}	number_orifice	Number of distributor plate orifices per area
$\delta_{t,x}$	delta	Volume fraction occupied by bubble region
$\delta_{e,t,x}$	delta_e	Volume fraction occupied by emulsion region
$\varepsilon_{t,x}$	voidage_average	Cross-sectional average voidage
$\varepsilon_{e,t,x}$	voidage_emulsion	Emulsion region voidage fraction
$\phi_{se,t,x}$	solid_emulsion_region.particle_porosity	Particle porosity of solid
$\gamma_{t,x}$	bubble_growth_coeff	Bubble growth coefficient
$d_{bm,t,x}$	bubble_diameter_max	Maximum theoretical bubble diameter
$d_{b,t,x}$	bubble_diameter	Average bubble diameter
$v_{g,t,x}$	velocity_superficial_gas	Gas superficial velocity
$v_{ge,t,x}$	velocity_emulsion_gas	Emulsion region superficial gas velocity
$v_{br,t,x}$	velocity_bubble_rise	Bubble rise velocity
$v_{b,t,x}$	velocity_bubble	Average bubble diameter
$K_{be,t,x,j}$	Kbe	Bubble to emulsion gas mass transfer coefficient
$K_{gbulk,t,x,j}$	Kgbulk_c	Gas phase component bulk transfer rate
$H_{be,t,x,j}$	Hbe	Bubble to emulsion gas heat transfer coefficient
$c_{p,vap,b,t,x}$	cp_mol	Mixture mole heat capacity
$h_{tc,t,x}$	htc_conv	Gas to solid convective heat transfer coefficient
$\mu_{vap,ge,t,x}$	visc_d	Mixture dynamic viscosity
$h_{t-gs,t,x}$	ht_conv	Gas to solid convective enthalpy transfer
$H_{gbulk,t,x}$	Hgbulk	Bulk gas heat transfer between bubble and emulsion
$r_{hetero,ge,t,x,j}$	gas_emulsion_hetero_rxn	Gas emulsion heterogeneous rate reaction generation
L_b	bubble_region.length	
L_{ge}	gas_emulsion_region.length	
L_{se}	solid_emulsion_region.length	
$A_{b,t,x}$	bubble_region.area	
$A_{ge,t,x}$	gas_emulsion_region.area	
$A_{se,t,x}$	solid_emulsion_region.area	
$\Delta P_{ge,t,x}$	gas_emulsion_region.deltaP	pressure drop across gas emulsion region
$\rho_{mass,se,t,x}$	solid_emulsion_region.properties.dens_mass_particle	solid particle mass density
$D_{vap,ge,t,x,j}$	gas_emulsion_region.properties.diffusion_comp	gas component diffusion in gas emulsion region
$C_{b,total,t,x}$	bubble_region.properties.dens_mol_vap	gas mole density in the bubble region
$C_{g,t,x}$	average_gas_density	average gas density
$C_{ge,total,t,x}$	gas_emulsion_region.properties.dens_mol_vap	gas mole density in the emulsion region
$M_{tr,b,t,x,p,j}$	bubble_region.mass_transfer_term	
$M_{tr,ge,t,x,p,j}$	gas_emulsion_region.mass_transfer_term	
$M_{tr,se,t,x,p,j}$	solid_emulsion_region.mass_transfer_term	
$r_{ext,b,t,x,r}$	bubble_region.rate_reaction_extent	
$r_{ext,ge,t,x,r}$	gas_emulsion_region.rate_reaction_extent	
$r_{ext,se,t,x,r}$	solid_emulsion_region.rate_reaction_extent	
$r_{b,t,x,r}$	bubble_region.reactions.reaction_rate	
$r_{ge,t,x,r}$	gas_emulsion_region.reactions.reaction_rate	

continues on next page

Table 9 – continued from previous page

Variable	Name	Notes
$r_{se,t,x,r}$	solid_emulsion_region.reactions.reaction_rate	
$k_{vap,b,t,x}$	bubble_region.properties.therm_cond	bubble region thermal conductivity
$k_{vap,e,t,x}$	gas_emulsion_region.properties.therm_cond	gas emulsion region thermal conductivity
$T_{b,t,x}$	bubble_region.properties.temperature	
$T_{ge,t,x}$	gas_emulsion_region.properties.temperature	
$T_{se,t,x}$	solid_emulsion_region.properties.temperature	
$H_{tr,b,t,x}$	bubble_region.heat	bubble region heat transfer term
$H_{tr,ge,t,x}$	gas_emulsion_region.heat	gas emulsion region heat transfer term
$H_{tr,se,t,x}$	solid_emulsion_region.heat	solid emulsion region heat transfer term
$F_{mol,b,t,x}$	bubble_region.properties.flow_mol	
$F_{mol,ge,t,x}$	gas_emulsion_region.properties.flow_mol	
$y_{b,t,x,j}$	bubble_region.properties.mole_frac	
$y_{ge,t,x,j}$	gas_emulsion_region.properties.mole_frac	
$x_{se,t,x,j}$	solid_emulsion_region.properties.mass_frac	
$P_{ge,t,x}$	gas_emulsion_region.properties.pressure	
$F_{mol,g,t,inlet}$	gas_inlet.flow_mol	
$y_{g,t,inlet,j}$	gas_inlet.mole_frac	
$P_{g,t,inlet}$	gas_inlet.pressure	
$T_{g,t,inlet}$	gas_inlet.temperature	
$H_{g,t,inlet}$	gas_inlet.enthalpy	
$F_{mass,s,t,inlet}$	solid_inlet.flow_mass	
$\phi_{t,inlet}$	solid_inlet.particle_porosity	
$x_{s,t,inlet}$	solid_inlet.mass_frac	
$T_{s,t,inlet}$	solid_inlet.temperature	
$H_{s,t,inlet}$	solid_inlet.enthalpy	
$F_{mass,se,t,x}$	solid_emulsion_region.properties.flow_mass	
$H_{b,t,x}$	bubble_region.properties.enthalpy	
$H_{ge,t,x}$	gas_emulsion_region.properties.enthalpy	
$H_{se,t,x}$	solid_emulsion_region.properties.enthalpy	
$F_{mol,g,t,outlet}$	gas_outlet.flow_mol	
$y_{g,t,outlet,j}$	gas_outlet.mole_frac	
$P_{g,t,outlet}$	gas_outlet.pressure	
$T_{g,t,outlet}$	gas_outlet.temperature	
$H_{g,t,outlet}$	gas_outlet.enthalpy	
$F_{mass,s,t,outlet}$	solid_outlet.flow_mass	
$\phi_{t,outlet}$	solid_outlet.particle_porosity	
$x_{s,t,outlet}$	solid_outlet.mass_frac	
$T_{s,t,outlet}$	solid_outlet.temperature	
$H_{s,t,outlet}$	solid_outlet.mass_enthalpy	
$v_{mf,se}$	solid_emulsion_region.properties.velocity_mf	velocity at minimum fluidization
$\varepsilon_{mf,se}$	solid_emulsion_region.properties.voidage_mf	voidage at minimum fluidization
K_d	Kd	bulk gas permeation coefficient
$d_{p,se}$	solid_emulsion_region.properties._params.particle_dia	
ΔP_{or}	deltaP_orifice	Pressure drop across orifice

Parameters

List of parameters in the BFBR model:

Parameter	Name	Notes
$s_{se,j,r}$	rate_reaction_stoichiometry	Reference to solid_emulsion_region.reactions.rate_reaction_stoichiometry

Subscripts

List of subscripts in the BFBR model:

Subscript	Name
b	bubble region
e	emulsion region
g	gas phase
ge	gas_emulsion
j	component
p	phase
s	solid phase
se	solid_emulsion
r	reaction
t	time
x	length

Initialization

The initialization method for this model will save the current state of the model before commencing initialization and reloads it afterwards. The state of the model will be the same after initialization, only the initial guesses for unfixed variables will be changed.

The model allows for the passing of a dictionary of values of the state variables of the gas and solid phases that can be used as initial guesses for the state variables throughout the time and spatial domains of the model. This is optional but recommended. A typical guess could be values of the gas and solid inlet port variables at time $t=0$.

The model initialization proceeds through a sequential hierarchical method where the model equations are deactivated at the start of the initialization routine, and the complexity of the model is built up through activation and

solution of various sub-model blocks and equations at each initialization step. At each step the model variables are updated to better guesses obtained from the model solution at that step.

The initialization routine proceeds as follows:

- Step 1. Initialize the thermo-physical and transport properties model blocks
- Step 2. Initialize geometric constraints
- Step 3. Initialize the hydrodynamic properties
- Step 4a. Initialize pressure drop and mass balances without reactions
- Step 4b. Initialize mass balances with reactions
- Step 5. Initialize energy balances

BFBR Class

```
class idaes.gas_solid_contactors.unit_models.bubbling_fluidized_bed.BubblingFluidizedBed(*args,  
**kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

finite_elements Number of finite elements to use when discretizing length domain (default=20)

length_domain_set length_domain_set - (optional) list of point to use to initialize a new ContinuousSet if length_domain is not provided (default = [0.0, 1.0]).

transformation_method Method to use to transform domain. Must be a method recognised by the Pyomo TransformationFactory, **default** - "dae.finite_difference". **Valid values:** { "dae.finite_difference" - Use a finite difference transformation scheme, "dae.collocation" - use a collocation transformation scheme }

transformation_scheme Scheme to use when transforming domain. See Pyomo documentation for supported schemes, **default** - None. **Valid values:** { **None** - defaults to "BACKWARD" for finite difference transformation method, and to "LAGRANGE-RADAU" for collocation transformation method, "BACKWARD" - Use a finite difference transformation method, "FORWARD" - use a finite difference transformation method, "LAGRANGE-RADAU" - use a collocation transformation method }

collocation_points Number of collocation points to use per finite element when discretizing length domain (default=3)

flow_type Flow configuration of Bubbling Fluidized Bed **default** - "co_current". **Valid values:** { "co_current" - gas flows from 0 to 1, solid flows from 0 to 1, "counter_current" - gas flows from 0 to 1, solid flows from 1 to 0. }

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.componentTotal. **Valid values:** { **MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **MaterialBalanceType.componentTotal** - use total component balances, **Mate-**

rialBalanceType.elementTotal - use total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - **EnergyBalanceType.enthalpyTotal**. **Valid values:** { **EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - **MomentumBalanceType.none**. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - **False**. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

gas_phase_config gas phase config arguments

dynamic Indicates whether this model will be dynamic or not, **default** = **useDefault**. **Valid values:** { **useDefault** - get flag from parent (default = **False**), **True** - set as a dynamic model, **False** - set as a steady-state model.}

has_holdup Indicates whether holdup terms should be constructed or not. Must be **True** if **dynamic** = **True**,

default - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms. }

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object. }

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation. }

solid_phase_config solid phase config arguments

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default =

False), **True** - construct holdup terms,
False - do not construct holdup terms}

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object.}

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

- **initialize** (*dict*) - ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the "default" argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (BubblingFluidizedBed) New instance

BFBRData Class

class `idaes.gas_solid_contactors.unit_models.bubbling_fluidized_bed.BubblingFluidizedBedData` (co
Standard Bubbling Fluidized Bed Unit
Model Class

build()
Begin building model :param None:

Returns None

initialize (*gas_phase_state_args*={}, *solid_phase_state_args*={}, *outlvl*=0, *solver*='ipopt', *optarg*={'tol': 1e-06})
Initialisation routine for Bubbling Fluidized Bed unit

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine * 0 = no output (default) * 1 = return solver state for each step in routine * 2 = return solver state for each step in subroutines * 3 = include solver output information (tee=True)
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

results_plot()
Plot method for common bubbling fluidized bed variables

Variables plotted: Tge : temperature of gas in the emulsion region Tgb : temperature of gas in the bubble region Tse : temperature of solid in the emulsion region Ge : flowrate of gas in the emulsion region Gb : flowrate of gas in the bubble region cet : total concentration of gas in the emulsion region cbt : total concentration of gas in the bubble region y_b : mole fraction of gas components in the bubble region x_e : mass fraction of solid components in the emulsion region

Moving Bed Reactor

The IDAES Moving Bed Reactor (MBR) model represents a unit operation where two material streams – a solid phase and a gas phase – pass through a linear reactor vessel while undergoing chemical reaction(s). The two streams have opposite flow directions (counter-flow). The MBR mathematical model is a 1-D rigorous first-principles model consisting of a set of differential equations obtained by applying the mass, energy (for each phase) and momentum balance equations.

Assumptions:

- The radial concentration and temperature gradients are assumed to be negligible.
- The reactor is assumed to be adiabatic.
- The solid phase is assumed to be moving at a constant velocity determined by the solids feed rate to the reactor.

Requirements:

- Property package contains temperature and pressure variables.
- Property package contains minimum fluidization velocity.

The MBR model is based on:

- A. Ostace, A. Lee, C.O. Okoli, A.P. Burgard, D.C. Miller, D. Bhattacharyya, Mathematical modeling of a moving-bed reactor for chemical looping combustion of methane, in: M.R. Eden, M. Ierapetritou, G.P. Towler (Eds.), 13th Int. Symp. Process Syst. Eng. (PSE 2018), Computer-Aided Chemical Engineering 2018, pp. 325–330, San Diego, CA.

Degrees of Freedom

MBRs generally have at least 2 (or more) degrees of freedom, consisting of design and operating variables. The design variables of reactor length and diameter are typically the minimum variables to be fixed.

Model Structure

The core MBR unit model consists of two `ControlVolume1DBlock` Blocks (named `gas_phase` and `solid_phase`), each with one Inlet Port (named `gas_inlet` and `solid_inlet`) and one Outlet Port (named `gas_outlet` and `solid_outlet`).

Constraints

In the following, the subscripts g and s refer to the gas and solid phases, respectively. In addition to the constraints written by the `control_volume` Block, MBR units write the following Constraints:

Geometry Constraints

Area of the reactor bed:

$$A_{bed} = \pi \left(\frac{D_{bed}}{2} \right)^2$$

Area of the gas domain:

$$A_{g,t,x} = \varepsilon A_{bed}$$

Area of the solid domain:

$$A_{s,t,x} = (1 - \varepsilon) A_{bed}$$

Length of the gas domain:

$$L_g = L_{bed}$$

Length of the solid domain:

$$L_s = L_{bed}$$

Hydrodynamic Constraints

Superficial velocity of the gas:

$$u_{g,t,x} = \frac{F_{mol,g,t,x}}{A_{bed}\rho_{mol,g,t,x}}$$

Superficial velocity of the solids:

$$u_{s,t} = \frac{F_{mass,s,t,inlet}}{A_{bed}\rho_{mass,s,t,inlet}}$$

Pressure drop:

The constraints written by the MBR model to compute the pressure drop (if *has_pressure_change* is ‘True’) in the reactor depend upon the construction arguments chosen:

If *pressure_drop_type* is *simple_correlation*:

$$-\frac{dP_{g,t,x}}{dx} = 0.2 (\rho_{mass,s,t,x} - \rho_{mass,g,t,x}) u_{g,t,x}$$

If *pressure_drop_type* is *er-gun_correlation*:

$$-\frac{dP_{g,t,x}}{dx} = \frac{150\mu_{g,t,x}(1-\varepsilon)^2(u_{g,t,x} + u_{s,t})}{\varepsilon^3 d_p^2} + \frac{1.75(1-\varepsilon)\rho_{mass,g,t,x}(u_{g,t,x} + u_{s,t})^2}{\varepsilon^3 d_p}$$

Reaction Constraints

Gas phase reaction extent:

If *gas_phase_config.reaction_package* is not ‘None’:

$$\xi_{g,t,x,r} = r_{g,t,x,r} A_{g,t,x}$$

Solid phase reaction extent:

If *solid_phase_config.reaction_package* is not ‘None’:

$$\xi_{s,t,x,r} = r_{s,t,x,r} A_{s,t,x}$$

Gas phase heterogeneous rate generation/consumption:

$$M_{g,t,x,p,j} = A_{s,t,x} \sum_r^{reactions} \nu_{s,j,r} r_{s,t,x,r}$$

Dimensionless numbers, mass and heat transfer coefficients

Particle Reynolds number:

$$Re_{p,t,x} = \frac{u_{g,t,x} \rho_{mass,g,t,x}}{\mu_{g,t,x} d_p}$$

Prandtl number:

$$Pr_{t,x} = \frac{c_{p,t,x} \mu_{g,t,x}}{k_{g,t,x}}$$

Particle Nusselt number:

$$Nu_{p,t,x} = 2 + 1.1 Pr_{t,x}^{1/3} |Re_{p,t,x}|^{0.6}$$

Particle to fluid heat transfer coefficient

$$h_{gs,t,x} d_p = Nu_{p,t,x} k_{g,t,x}$$

If *energy_balance_type* not *EnergyBalanceType.none*:

Gas phase - gas to solid heat transfer:

$$H_{g,t,x} = -\frac{6}{d_p} h_{gs,t,x} (T_{g,t,x} - T_{s,t,x}) A_{s,t,x}$$

Solid phase - gas to solid heat transfer:

$$H_{s,t,x} = \frac{6}{d_p} h_{gs,t,x} (T_{g,t,x} - T_{s,t,x}) A_{s,t,x}$$

List of Variables

Variable	Description	Reference to
A_{bed}	Reactor bed cross-sectional area	<code>bed_area</code>
$A_{g,t,x}$	Gas phase area (interstitial cross-sectional area)	<code>gas_phase.area</code>
$A_{s,t,x}$	Solid phase area	<code>solid_phase.area</code>
$c_{p,t,x}$	Gas phase heat capacity (constant P)	<code>gas_phase.properties.cp_mass</code>
D_{bed}	Reactor bed diameter	<code>bed_diameter</code>
$F_{mass,s,t,inlet}$	Total mass flow rate of solids, at inlet ($x = 1$)	<code>solid_phase.properties.flow_mass</code>
$F_{mol,g,t,x}$	Total molar flow rate of gas	<code>gas_phase.properties.flow_mol</code>
$H_{g,t,x}$	Gas to solid heat transfer term, gas phase	<code>gas_phase.heat</code>
$H_{s,t,x}$	Gas to solid heat transfer term, solid phase	<code>solid_phase.heat</code>
$h_{gs,t,x}$	Gas-solid heat transfer coefficient	<code>gas_solid_htc</code>
$k_{g,t,x}$	Gas thermal conductivity	<code>gas_phase.properties.therm_cond</code>
L_{bed}	Reactor bed height	<code>bed_height</code>
L_g	Gas domain length	<code>gas_phase.length</code>
L_s	Solid domain length	<code>solid_phase.length</code>
$M_{g,t,x,p,j}$	Rate generation/consumption term, gas phase	<code>gas_phase.mass_transfer_term</code>
$Nu_{p,t,x}$	Particle Nusselt number	<code>Nu_particle</code>

continues on next page

Table 10 – continued from previous page

Variable	Description	Reference to
$dP_{g,t,x}$	Total pressure derivative w.r.t. x (axial position)	gas_phase.deltaP
$Pr_{t,x}$	Prandtl number	Pr
$r_{g,t,x,r}$	Gas phase reaction rate	gas_phase.reactions.reaction_rate
$r_{s,t,x,r}$	Solid phase reaction rate	solid_phase.reactions.reaction_rate
$Re_{p,t,x}$	Particle Reynolds number	Re_particle
$T_{g,t,x}$	Gas phase temperature	gas_phase.properties.temperature
$T_{s,t,x}$	Solid phase temperature	solid_phase.properties.temperature
$u_{g,t,x}$	Superficial velocity of the gas	velocity_superficial_gas
$u_{s,t}$	Superficial velocity of the solids	velocity_superficial_solid
<i>Greek letters</i>		
ε	Reactor bed voidage	bed_voidage
$\mu_{g,t,x}$	Dynamic viscosity of gas mixture	gas_phase.properties.visc_d
$\xi_{g,t,x,r}$	Gas phase reaction extent	gas_phase.rate_reaction_extent
$\xi_{s,t,x,r}$	Solid phase reaction extent	solid_phase.rate_reaction_extent
$\rho_{mass,g,t,inlet}$	Density of gas mixture	gas_phase.properties.dens_mass
$\rho_{mass,s,t,inlet}$	Density of solid particles	solid_phase.properties.dens_mass_particle
$\rho_{mol,g,t,x}$	Molar density of the gas	gas_phase.properties.dens_mole

List of Parameters

Parameter	Description	Reference to
d_p	Solid particle diameter	solid_phase.properties._params.particle_dia
$\nu_{s,j,r}$	Stoichiometric coefficients	solid_phase.reactions.rate_reaction_stoichiometry

Initialization

The initialization method for this model will save the current state of the model before commencing initialization and reloads it afterwards. The state of the model will be the same after initialization, only the initial guesses for unfixed variables will be changed.

The model allows for the passing of a dictionary of values of the state variables of the gas and solid phases that can be used as initial guesses for the state variables throughout the time and spatial domains of the model. This is optional but recommended. A typical guess could be values of the gas and solid inlet port variables at time $t = 0$.

The model initialization proceeds through a sequential hierarchical method where the model equations are

deactivated at the start of the initialization routine, and the complexity of the model is built up through activation and solution of various sub-model blocks and equations at each initialization step. At each step the model variables are updated to better guesses obtained from the model solution at that step.

The initialization routine proceeds in as follows:

- Step 1: Initialize the thermo-physical and transport properties model blocks.
- Step 2: Initialize the hydrodynamic properties.
- Step 3a: Initialize mass balances without reactions and pressure drop.
- Step 3b: Initialize mass balances with reactions and without pressure drop.
- Step 3c: Initialize mass balances with reactions and pressure drop.
- Step 4: Initialize energy balances.

MBR Class

```
class idaes.gas_solid_contactors.unit_models.moving_bed.MBR(*args, **kws)
```

Parameters

- **rule** (*function*) – A rule function or None. Default rule calls build().
- **concrete** (*bool*) – If True, make this a toplevel model. **Default** - False.
- **ctype** (*str*) – Pyomo ctype of the block. **Default** - “Block”
- **default** (*dict*) – Default Process-BlockData config

Keys

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { useDefault - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model. }

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True,

default - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

finite_elements Number of finite elements to use when discretizing length domain (default=20)

length_domain_set length_domain_set - (optional) list of point to use to initialize a new ContinuousSet if length_domain is not provided (default = [0.0, 1.0])

transformation_method Method to use to transform domain. Must be a method recognised by the Pyomo TransformationFactory, **default** - "dae.finite_difference". **Valid values:** { "dae.finite_difference" - Use a finite difference transformation method, "dae.collocation" - use a collocation transformation method }

transformation_scheme Scheme to use when transforming domain. See Pyomo documentation for supported schemes, **default** - None. **Valid values:** { **None** - defaults to "BACKWARD" for finite difference transformation method, and to "LAGRANGE-RADAU" for collocation transformation method, "BACKWARD" - Use a finite difference transformation method, "FORWARD" - use a finite difference transformation method, "LAGRANGE-RADAU" - use a collocation transformation method }

collocation_points Number of collocation points to use per finite element when discretizing length domain (default=3)

flow_type Flow configuration of Moving Bed - counter_current: gas side flows from 0 to 1 solid side flows from 1 to 0

material_balance_type Indicates what type of mass balance should be constructed, **default** - MaterialBalanceType.componentTotal. **Valid values:** { **MaterialBalanceType.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use phase component balances, **Ma-**

terialBalanceType.componentTotal - use total component balances, **MaterialBalanceType.elementTotal** - use total element balances, **MaterialBalanceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be constructed, **default** - **EnergyBalanceType.enthalpyTotal**. **Valid values:** { **EnergyBalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBalanceType.energyTotal** - single energy balance for material, **EnergyBalanceType.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should be constructed, **default** - **MomentumBalanceType.pressureTotal**. **Valid values:** { **MomentumBalanceType.none** - exclude momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalanceType.pressurePhase** - pressure balances for each phase, **MomentumBalanceType.momentumTotal** - single momentum balance for material, **MomentumBalanceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be constructed, **default** - **False**. **Valid values:** { **True** - include pressure change terms, **False** - exclude pressure change terms.}

pressure_drop_type Indicates what type of pressure drop correlation should be used, **default** - **“simple_correlation”**. **Valid values:** { **“simple_correlation”** - Use a simplified pressure drop correlation, **“ergun_correlation”** - Use the ergun equation.}

gas_phase_config gas phase config arguments

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { useDefault - get flag from parent (default = False), **True** - set as a dynamic model, **False** - set as a steady-state model.}

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { useDefault - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object.}

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation.}

solid_phase_config solid phase config arguments

dynamic Indicates whether this model will be dynamic or not, **default** = useDefault. **Valid values:** { useDefault

- get flag from parent (default = False),
True - set as a dynamic model, **False** -
 set as a steady-state model.}

has_holdup Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - False. **Valid values:** { **useDefault** - get flag from parent (default = False), **True** - construct holdup terms, **False** - do not construct holdup terms }

has_equilibrium_reactions Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - True. **Valid values:** { **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms. }

property_package Property parameter object used to define property calculations (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a ParameterBlock object

property_package_args A dict of arguments to be passed to the PropertyBlockData and used when constructing these (default = 'use_parent_value') - 'use_parent_value' - get package from parent (default = None) - a dict (see property package for documentation)

reaction_package Reaction parameter object used to define reaction calculations, **default** - None. **Valid values:** { **None** - no reaction package, **ReactionParameterBlock** - a ReactionParameterBlock object. }

reaction_package_args A ConfigBlock with arguments to be passed to a reaction block(s) and used when constructing these, **default** - None. **Valid values:** { see reaction package for documentation. }

- **initialize** (*dict*) - ProcessBlockData config for individual elements. Keys are BlockData indexes and values are dictionaries described under the "default" argument above.
- **idx_map** (*function*) - Function to take the index of a BlockData element and return the index in the initialize dict

from which to read arguments. This can be provided to override the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (MBR) New instance

MBRData Class

class `idaes.gas_solid_contactors.unit_models.moving_bed.MBRData` (*component*)
Standard Moving Bed Unit Model Class.

build ()
Begin building model (pre-DAE transformation).

Parameters None –

Returns None

initialize (*gas_phase_state_args*={}, *solid_phase_state_args*={}, *outlvl*=0, *solver*='ipopt', *optarg*={'tol': 1e-06})
Initialisation routine for MB unit (default solver ipopt).

Keyword Arguments

- **state_args** – a dict of arguments to be passed to the property package(s) to provide an initial state for initialization (see documentation of the specific property package) (default = {}).
- **outlvl** – sets output level of initialization routine
- **optarg** – solver options dictionary object (default={'tol': 1e-6})
- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

results_plot ()
Plot method for common moving bed variables

Variables plotted: Tg : Temperature in gas phase Ts : Temperature in solid phase vg : Superficial gas velocity P : Pressure in gas phase Ftotal : Total molar flowrate of gas Mtotal : Total mass flowrate of solid Cg : Concentration of gas components in the gas phase y_frac : Mole fraction of gas components in the gas phase x_frac : Mass fraction of solid components in the solid phase

Gas Solid Contactors Property Models

Contents

`methane_iron_OC_reduction`

Contents

4.6 Related Packages

In addition to the IDAES PSE framework, several related software packages have been developed and/or extended to support capabilities for IDAES.

4.6.1 Pecos: Data Quality Control and Fault Detection

Before using plant data in process models, quality control and fault detection analysis is recommended to identify potential data issues (e.g., missing or corrupt data) and data points that are not suitable for the intended analysis (e.g., abnormal plant behavior). The following documentation describes methods to run data quality control and fault detection analysis using Pecos.

Pecos is an open-source Python package designed to monitor performance of time series data, subject to a series of quality control tests. The software includes methods to run quality control tests defined by the user and generate reports which include test results and graphics. Results from the quality control analysis can be used to extract “clean data” which removes data points that failed quality control inspection. Pecos was originally developed for the U.S. Department of Energy in 2016 to monitor solar photovoltaic systems and has been used for a wide range of applications since. The software was updated for IDAES to facilitate near real-time analysis using continuous data streams.

More information on Pecos can be found in the online documentation at

<https://pecos.readthedocs.io>. The software repository is located at <https://github.com/sandialabs/pecos>.

The following functionality is available in Pecos:

- Check data for missing, non-monotonic, and duplicate time stamps
- Check for missing data
- Check for corrupt data
- Check for data that are outside the expected range
- Check for stagnant data and/or abrupt changes in the data using the difference between max and min values within a rolling window
- Check for outliers using normalized data within a rolling window

The analysis generates the following information:

- Cleaned data (data that failed a test are removed)
- Boolean mask (indicates which data points failed a test)
- Summary of the quality control test results (includes the variable name, start and end time for each failure, and an error message)

The test results summary and accompanying graphics can be included in HTML or LATEX reports generated using Pecos.

Pecos supports both static and streaming analysis along with custom quality control functions:

- Static analysis operates on the entire data set to determine if all data points are normal or anomalous. While this can include operations like moving window statistics, the quality control tests operate on the entire data set at once.
- Streaming analysis loops through each data point using a quality control test that relies on information from “clean data” in a moving window. If a data point is determined to be anomalous, it

is not included in the window for subsequent analysis.

- The user can define custom quality control functions used to determine if data is anomalous and return custom meta-data from the analysis.

Data points that do not pass quality control inspection should be removed or replaced by various means (interpolation, data from a duplicate sensor, values from a model) before using the data for further analysis. Data replacement strategies are generally defined on a case-by-case basis. If large sections of the data failed quality control tests, the data might not be suitable for use.

The raw data, results from the quality control analysis, and the analysis files used to run Pecos can be stored in the Data Management Framework (DMF) to ensure reproducible results.

Installation

To install Pecos using pip:

```
pip install pecos
```

To install Pecos using git:

```
git clone https://  
→github.com/sandialabs/pecos  
cd pecos  
python setup.py install
```

Examples

IDAES examples that use Pecos are listed on the [examples online documentation page](#). Pecos also includes several general examples, located at <https://github.com/sandialabs/pecos/tree/master/examples>.

4.7 License

Institute for the Design of Advanced Energy Systems Process Systems Engineering Framework (IDAES PSE Framework) Copyright (c) 2019, by the software owners: The Regents of the University of California, through Lawrence Berkeley National Laboratory, National Technology & Engineering Solutions of Sandia, LLC, Carnegie Mellon University, West Virginia University Research Corporation, et al. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the Institute for the Design of Advanced Energy Systems (IDAES), University of California, Lawrence Berkeley National Laboratory, National Technology & Engineering Solutions of Sandia, LLC, Sandia National Laboratories, Carnegie Mellon University, West Virginia University Research Corporation, U.S. Dept. of Energy, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN

NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code (“Enhancements”) to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant Lawrence Berkeley National Laboratory the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.

4.8 Copyright

Institute for the Design of Advanced Energy Systems Process Systems Engineering Framework (IDAES PSE Framework) was produced under the DOE Institute for the Design of Advanced Energy Systems (IDAES), and is copyright (c) 2018-2019 by the software owners: The Regents of the University of California, through Lawrence Berkeley National Laboratory, National Technology & Engineering Solutions of Sandia, LLC, Carnegie Mellon University, West Virginia University Research

Corporation, et al. All rights reserved.

NOTICE. This Software was developed under funding from the U.S. Department of Energy and the U.S. Government consequently retains certain rights. As such, the U.S. Government has been granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, distribute copies to the public, prepare derivative works, and perform publicly and display publicly, and to permit other to do so. Copyright (C) 2018-2019 IDAES - All Rights Reserved

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

i

idaes.apps.caprese.nmpc, 206
 idaes.apps.matopt.materials.canvas, 201
 idaes.apps.matopt.materials.design, 201
 idaes.apps.matopt.materials.lattices.lattice, 201
 idaes.apps.matopt.opt.mat_modeling, 203
 idaes.core.components, 358
 idaes.core.control_volume0d, 306
 idaes.core.control_volumeld, 320
 idaes.core.flowsheet_model, 302
 idaes.core.phases, 361
 idaes.core.process_base, 299
 idaes.core.process_block, 297
 idaes.core.property_base, 340
 idaes.core.reaction_base, 349
 idaes.core.unit_model, 354
 idaes.core.util.dyn_utils, 363
 idaes.core.util.homotopy, 370
 idaes.core.util.initialization, 371
 idaes.core.util.misc, 438
 idaes.core.util.model_serializer, 376
 idaes.core.util.model_statistics, 392
 idaes.core.util.scaling, 407
 idaes.core.util.tables, 415
 idaes.core.util.unit_costing, 424
 idaes.gas_solid_contactors.unit_models.bubblingFluidizedBed, 759
 idaes.gas_solid_contactors.unit_models.movingBed, 770
 idaes.generic_models.properties.activity_coeff_models.activity_coeff_prop_pack, 454
 idaes.generic_models.properties.cubic_eos, 446
 idaes.generic_models.properties.helmholtz_helmholtz, 457
 idaes.generic_models.properties.iapws95, 458
 idaes.generic_models.properties.swco2, 465
 idaes.generic_models.unit_models.cstr, 487
 idaes.generic_models.unit_models.equilibrium_reactor, 491
 idaes.generic_models.unit_models.feed, 495
 idaes.generic_models.unit_models.feed_flash, 499
 idaes.generic_models.unit_models.flash, 503
 idaes.generic_models.unit_models.gibbs_reactor, 508
 idaes.generic_models.unit_models.heat_exchanger_1D, 528
 idaes.generic_models.unit_models.heater, 511
 idaes.generic_models.unit_models.mixer, 538
 idaes.generic_models.unit_models.plug_flow_reactor, 545
 idaes.generic_models.unit_models.pressure_changer, 551
 idaes.generic_models.unit_models.product, 556
 idaes.generic_models.unit_models.separator, 561
 idaes.generic_models.unit_models.statejunction, 568
 idaes.generic_models.unit_models.stoichiometric_reactor, 571
 idaes.generic_models.unit_models.translator, 575
 idaes.generic_models.unit_models.valve, 578
 idaes.power_generation.unit_models.feedwater_heater, 595
 idaes.power_generation.unit_models.helm.turbine, 601
 idaes.power_generation.unit_models.helm.turbine_in, 606
 idaes.power_generation.unit_models.helm.turbine_mu, 625
 idaes.power_generation.unit_models.helm.turbine_out, 613

`idaes.power_generation.unit_models.helm.turbine_stage,`
 [620](#)
`idaes.power_generation.unit_models.helm.valve_steam,`
 [635](#)
`idaes.surrogate.pysmo.kriging,` [187](#)
`idaes.surrogate.pysmo.polynomial_regression,`
 [173](#)
`idaes.surrogate.pysmo.radial_basis_function,`
 [180](#)

Symbols

<code>_GeneralVarLikeExpressionData</code> (class in <code>idaes.core.util.misc</code>), 439	<code>--create</code>
<code>__init__()</code> (<code>idaes.surrogate.pysmo.kriging.KrigingModel</code> method), 187	<code>dmf-init</code> command line option, 99
<code>__init__()</code> (<code>idaes.surrogate.pysmo.polynomial_regression.PolynomialRegression</code> method), 176	<code>idaes-bin-directory</code> command line option, 135
<code>__init__()</code> (<code>idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunctions</code> method), 182	<code>idaes-data-directory</code> command line option, 137
<code>__init__()</code> (<code>idaes.surrogate.pysmo.sampling.CVTSampling</code> method), 197	<code>idaes-lib-directory</code> command line option, 140
<code>__init__()</code> (<code>idaes.surrogate.pysmo.sampling.HaltonSampling</code> method), 193	<code>--created value</code>
<code>__init__()</code> (<code>idaes.surrogate.pysmo.sampling.HammersleySampling</code> method), 195	<code>dmf-find</code> command line option, 96
<code>__init__()</code> (<code>idaes.surrogate.pysmo.sampling.LatinHypercubeSampling</code> method), 190	<code>--derived resource</code>
<code>__init__()</code> (<code>idaes.surrogate.pysmo.sampling.UniformSampling</code> method), 192	<code>dmf-register</code> command line option, 103
<code>-I</code>	<code>--desc</code>
<code>idaes-get-examples</code> command line option, 137	<code>dmf-init</code> command line option, 99
<code>-N</code>	<code>idaes-bin-directory</code> command line option, 135
<code>idaes-get-examples</code> command line option, 138	<code>idaes-data-directory</code> command line option, 137
<code>-S, --sort</code>	<code>idaes-lib-directory</code> command line option, 140
<code>dmf-ls</code> command line option, 101	<code>--file value</code>
<code>-U</code>	<code>dmf-find</code> command line option, 96
<code>idaes-get-examples</code> command line option, 138	<code>--help</code>
<code>-V</code>	command line option, 142
<code>idaes-get-examples</code> command line option, 138	<code>idaes-bin-directory</code> command line option, 135
<code>--by value</code>	<code>idaes-copyright</code> command line option, 136
<code>dmf-find</code> command line option, 96	<code>idaes-data-directory</code> command line option, 137
<code>--color</code>	<code>idaes-get-examples</code> command line option, 137
<code>dmf-ls</code> command line option, 101	<code>idaes-get-extensions</code> command line option, 139
<code>dmf-related</code> command line option, 107	<code>idaes-lib-directory</code> command line option, 140
<code>dmf-status</code> command line option, 111	<code>--is-subject</code>
<code>--contained resource</code>	<code>dmf-register</code> command line option, 103
<code>dmf-register</code> command line option, 103	<code>--list, --no-list</code>

dmf-rm command line option, [109](#)
--list-releases
 idaes-get-examples command line
 option, [137](#)
--modified value
 dmf-find command line option, [96](#)
--multiple
 dmf-info command line option, [97](#)
 dmf-rm command line option, [109](#)
--name
 dmf-init command line option, [99](#)
--name value
 dmf-find command line option, [96](#)
--no-color
 dmf-ls command line option, [101](#)
 dmf-related command line option, [107](#)
 dmf-status command line option, [111](#)
--no-copy
 dmf-register command line option,
 [103](#)
--no-download
 idaes-get-examples command line
 option, [138](#)
--no-install
 idaes-get-examples command line
 option, [137](#)
--no-prefix
 dmf-ls command line option, [101](#)
--no-unicode
 dmf-related command line option, [107](#)
--no-unique
 dmf-register command line option,
 [103](#)
--output value
 dmf-find command line option, [96](#)
--prev resource
 dmf-register command line option,
 [103](#)
--quiet
 command line option, [142](#)
 dmf command line option, [93](#)
--strict
 dmf-register command line option,
 [103](#)
--type value
 dmf-find command line option, [96](#)
--unicode
 dmf-related command line option, [107](#)
--unstable
 idaes-get-examples command line
 option, [138](#)
--url
 idaes-get-extensions command line
 option, [139](#)

--used resource
 dmf-register command line option,
 [103](#)
--verbose
 command line option, [142](#)
 dmf command line option, [93](#)
--version
 dmf-register command line option,
 [103](#)
--version TEXT
 idaes-get-examples command line
 option, [138](#)
-a, --all
 dmf-status command line option, [111](#)
-d, --dir TEXT
 idaes-get-examples command line
 option, [137](#)
-d, --direction
 dmf-related command line option, [107](#)
-f, --format value
 dmf-info command line option, [97](#)
-l
 idaes-get-examples command line
 option, [137](#)
-q
 command line option, [142](#)
 dmf command line option, [93](#)
-r, --reverse
 dmf-ls command line option, [101](#)
-s, --show
 dmf-ls command line option, [101](#)
-s, --show info
 dmf-status command line option, [111](#)
-t, --type
 dmf-register command line option,
 [103](#)
-v
 command line option, [142](#)
 dmf command line option, [93](#)
-y, --yes
 dmf-rm command line option, [109](#)

A

activated_block_component_generator()
 (in module *idaes.core.util.model_statistics*),
 [392](#)
activated_blocks_set() (in module
 idaes.core.util.model_statistics), [392](#)
activated_constraints_generator() (in
 module *idaes.core.util.model_statistics*), [393](#)
activated_constraints_set() (in module
 idaes.core.util.model_statistics), [393](#)
activated_equalities_generator() (in mod-
 ule *idaes.core.util.model_statistics*), [393](#)

activated_equalities_set() (in module *idaes.core.util.model_statistics*), 393
 activated_inequalities_generator() (in module *idaes.core.util.model_statistics*), 393
 activated_inequalities_set() (in module *idaes.core.util.model_statistics*), 393
 activated_objectives_generator() (in module *idaes.core.util.model_statistics*), 394
 activated_objectives_set() (in module *idaes.core.util.model_statistics*), 394
 active_variables_in_deactivated_blocks_set() (in module *idaes.core.util.model_statistics*), 394
 ActivityCoeffParameterBlock (class in *idaes.generic_models.properties.activity_coeff_model*), 454
 ActivityCoeffStateBlock (class in *idaes.generic_models.properties.activity_coeff_model*), 455
 ActivityCoeffStateBlockData (class in *idaes.generic_models.properties.activity_coeff_model*), 456
 add() (*idaes.core.util.misc.IndexedVarLikeExpression* method), 439
 add() (*idaes.core.util.misc.SimpleVarLikeExpression* method), 439
 add_adiabatic() (*idaes.generic_models.unit_models.pressure_changer.PressureChangerData* method), 554
 add_energy_mixing_equations() (*idaes.generic_models.unit_models.mixer.MixerData* method), 541
 add_energy_splitting_constraints() (*idaes.generic_models.unit_models.separator.SeparatorData* method), 565
 add_geometry() (*idaes.core.control_volume0d.ControlVolume0DBlockData* method), 308
 add_geometry() (*idaes.core.control_volume1d.ControlVolume1DBlockData* method), 323
 add_inlet_port() (*idaes.core.unit_model.UnitModelBlockData* method), 354
 add_inlet_port_objects() (*idaes.generic_models.unit_models.separator.SeparatorData* method), 565
 add_inlet_state_blocks() (*idaes.generic_models.unit_models.mixer.MixerData* method), 541
 add_isentropic() (*idaes.generic_models.unit_models.pressure_changer.PressureChangerData* method), 554
 add_isothermal() (*idaes.generic_models.unit_models.pressure_changer.PressureChangerData* method), 554
 add_material_mixing_equations() (*idaes.generic_models.unit_models.mixer.MixerData* method), 541
 add_material_splitting_constraints() (*idaes.generic_models.unit_models.mixer.MixerData* method), 541
 add_mixed_state_block() (*idaes.generic_models.unit_models.mixer.MixerData* method), 541
 add_mixed_state_block() (*idaes.generic_models.unit_models.separator.SeparatorData* method), 565
 add_momentum_splitting_constraints() (*idaes.generic_models.unit_models.separator.SeparatorData* method), 565
 add_outlet_port() (*idaes.core.unit_model.UnitModelBlockData* method), 355
 add_outlet_port_objects() (*idaes.generic_models.unit_models.separator.SeparatorData* method), 565
 add_outlet_state_blocks() (*idaes.generic_models.unit_models.separator.SeparatorData* method), 565
 add_phase_component_balances() (*idaes.core.control_volume0d.ControlVolume0DBlockData* method), 308
 add_phase_energy_balances() (*idaes.core.control_volume1d.ControlVolume1DBlockData* method), 323
 add_phase_enthalpy_balances() (*idaes.core.control_volume0d.ControlVolume0DBlockData* method), 308
 add_phase_enthalpy_balances() (*idaes.core.control_volume1d.ControlVolume1DBlockData* method), 324
 add_phase_momentum_balances() (*idaes.core.control_volume0d.ControlVolume0DBlockData* method), 309
 add_phase_momentum_balances() (*idaes.core.control_volume1d.ControlVolume1DBlockData* method), 324
 add_phase_pressure_balances() (*idaes.core.control_volume0d.ControlVolume0DBlockData* method), 309
 add_phase_pressure_balances() (*idaes.core.control_volume1d.ControlVolume1DBlockData* method), 324
 add_port() (*idaes.core.unit_model.UnitModelBlockData* method), 355
 add_port_objects() (*idaes.generic_models.unit_models.mixer.MixerData* method), 541

`method`), 541
`add_pressure_equality_equations()`
 (`idaes.generic_models.unit_models.mixer.MixerData`
`method`), 541
`add_pressure_minimization_equations()`
 (`idaes.generic_models.unit_models.mixer.MixerData`
`method`), 541
`add_pump()` (`idaes.generic_models.unit_models.pressure_changer.PressureChangerData`
`method`), 554
`add_reaction_blocks()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 309
`add_reaction_blocks()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 324
`add_split_fractions()`
 (`idaes.generic_models.unit_models.separator.SeparatorData`
`method`), 565
`add_state_blocks()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 309
`add_state_blocks()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 325
`add_state_material_balances()`
 (`idaes.core.unit_model.UnitModelBlockData`
`method`), 356
`add_total_component_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 309
`add_total_component_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 325
`add_total_element_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 310
`add_total_element_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 326
`add_total_energy_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 311
`add_total_energy_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 327
`add_total_enthalpy_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 311
`add_total_enthalpy_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 327
`add_total_material_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 312
`add_total_material_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 327
`add_total_momentum_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 312
`add_total_momentum_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 328
`add_total_pressure_balances()`
 (`idaes.core.control_volume0d.ControlVolume0DBlockData`
`method`), 312
`add_total_pressure_balances()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 328
`alamo`
`alamopy`
`alamo`, 158, 169
`alamopy`
`alamo`, 158, 169
`apply_transformation()`
 (`idaes.core.control_volume1d.ControlVolume1DBlockData`
`method`), 328
`arcs_to_stream_dict()` (in module
`idaes.core.util.tables`), 415
`atoms` (`idaes.apps.matopt.opt.mat_modeling.MaterialDescriptor`
`attribute`), 202
`atoms` (`idaes.apps.matopt.opt.mat_modeling.MatOptModel`
`attribute`), 203

B

`base_scaled_var_generator()` (in module
`idaes.core.util.scaling`), 413
`base_class_module()`
 (`idaes.core.process_block.ProcessBlock` class
`method`), 299
`base_class_name()`
 (`idaes.core.process_block.ProcessBlock` class
`method`), 299
`binary` (`idaes.apps.matopt.opt.mat_modeling.MaterialDescriptor`
`attribute`), 203
`BoilerFireside`
`idaes.power_generation.unit_models.boiler_fireside`
`BoilerHeatExchanger`
`idaes.power_generation.unit_models.boiler_heat_exchanger`
`BoilerHeatExchanger` (class in
`idaes.power_generation.unit_models.boiler_heat_exchanger`),
`BoilerHeatExchangerData` (class in
`idaes.power_generation.unit_models.boiler_heat_exchanger`),
`BoilerHeatExchangerData` 652

`bound()` (`idaes.core.util.model_serializer.StoreSpec` class method), 383
`bounds` (`idaes.apps.matopt.opt.mat_modeling.MaterialDescription` attribute), 203
`BubblingFluidizedBed` (class in `idaes.gas_solid_contactors.unit_models.bubbling_fluidized_bed`), 759
`BubblingFluidizedBedData` (class in `idaes.gas_solid_contactors.unit_models.bubbling_fluidized_bed`), 764
`build()` (`idaes.core.control_volume0d.ControlVolume0DBlockData` method), 312
`build()` (`idaes.core.control_volume1d.ControlVolume1DBlockData` method), 328
`build()` (`idaes.core.flowsheet_model.FlowsheetBlockData` method), 302
`build()` (`idaes.core.process_base.ProcessBlockData` method), 299
`build()` (`idaes.core.property_base.PhysicalParameterBlockData` method), 340
`build()` (`idaes.core.property_base.StateBlockData` method), 343
`build()` (`idaes.core.reaction_base.ReactionBlockDataBase` method), 351
`build()` (`idaes.core.reaction_base.ReactionParameterBlockData` method), 349
`build()` (`idaes.core.unit_model.UnitModelBlockData` method), 356
`build()` (`idaes.gas_solid_contactors.unit_models.bubbling_fluidized_bed` method), 764
`build()` (`idaes.gas_solid_contactors.unit_models.moving_bed` method), 775
`build()` (`idaes.generic_models.control.pid_controller.PIDBlockData` method), 590
`build()` (`idaes.generic_models.properties.activity_coeff_models.activity_coeff_models_activity_coeff_state_block_data` method), 456
`build()` (`idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicEquationOfStateBlockData` method), 446
`build()` (`idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicEquationOfStateBlockData` method), 447
`build()` (`idaes.generic_models.properties.iapws95.Iapws95ParameterBlockData` method), 464
`build()` (`idaes.generic_models.properties.iapws95.Iapws95StateBlockData` method), 461
`build()` (`idaes.generic_models.properties.interrogator.properties_interrogator_data` method), 481
`build()` (`idaes.generic_models.properties.interrogator.reactions_interrogator_data` method), 484
`build()` (`idaes.generic_models.properties.swco2.SWCO2ParameterBlockData` method), 470
`build()` (`idaes.generic_models.properties.swco2.SWCO2StateBlockData` method), 468
`build()` (`idaes.generic_models.unit_models.cstr.CSTRData` method), 490
`build()` (`idaes.generic_models.unit_models.equilibrium_reactor.EquilibriumReactorData` method), 494
`build()` (`idaes.generic_models.unit_models.feed.FeedData` method), 497
`build()` (`idaes.generic_models.unit_models.feed_flash.FeedFlashData` method), 500
`build()` (`idaes.generic_models.unit_models.flash.FlashData` method), 506
`build()` (`idaes.generic_models.unit_models.gibbs_reactor.GibbsReactorData` method), 510
`build()` (`idaes.generic_models.unit_models.heat_exchanger.HeatExchangerData` method), 522
`build()` (`idaes.generic_models.unit_models.heat_exchanger_1D.HeatExchanger1DData` method), 534
`build()` (`idaes.generic_models.unit_models.heater.HeaterData` method), 515
`build()` (`idaes.generic_models.unit_models.mixer.MixerData` method), 541
`build()` (`idaes.generic_models.unit_models.plug_flow_reactor.PFRData` method), 548
`build()` (`idaes.generic_models.unit_models.pressure_changer.PressureChangerData` method), 554
`build()` (`idaes.generic_models.unit_models.product.ProductData` method), 557
`build()` (`idaes.generic_models.unit_models.separator.SeparatorData` method), 566
`build()` (`idaes.generic_models.unit_models.statejunction.StateJunctionData` method), 569
`build()` (`idaes.generation.unit_models.bubbling_fluidized_bed` method), 574
`build()` (`idaes.generation.unit_models.moving_bed` method), 576
`build()` (`idaes.generic_models.unit_models.translator.TranslatorData` method), 576
`build()` (`idaes.generic_models.unit_models.valve.ValveData` method), 586
`build()` (`idaes.generic_models.properties.activity_coeff_models.activity_coeff_models_activity_coeff_state_block_data` method), 652
`build()` (`idaes.generation.unit_models.boiler_heat_exchanger_2D` method), 693
`build()` (`idaes.generation.unit_models.feedwater_heater_0D.FWHeaterData` method), 601
`build()` (`idaes.generation.unit_models.heat_exchanger_3streams` method), 710
`build()` (`idaes.generation.unit_models.helm.turbine.HelmIsentropicTurbineData` method), 605
`build()` (`idaes.generation.unit_models.helm.turbine_inlet.HelmTurbineInletData` method), 612
`build()` (`idaes.generation.unit_models.helm.turbine_multistage.HelmTurbineMultistageData` method), 633
`build()` (`idaes.generation.unit_models.helm.turbine_outlet.HelmTurbineOutletData` method), 619
`build()` (`idaes.generation.unit_models.helm.turbine_stage.HelmTurbineStageData` method), 624
`build()` (`idaes.power_generation.unit_models.helm.valve_steam.HelmValveSteamData` method), 640

`build_reaction_block()` (*idaes.core.reaction_base.ReactionParameterBlock* method), 349
`build_state_block()` (*idaes.core.property_base.PhysicalParameterBlock* method), 340
`build_txy_diagrams()` (in module *idaes.core.util.phase_equilibria*), 374
C
`calculate_bubble_point_pressure()` (*idaes.core.property_base.StateBlockData* method), 343
`calculate_bubble_point_temperature()` (*idaes.core.property_base.StateBlockData* method), 343
`calculate_dew_point_pressure()` (*idaes.core.property_base.StateBlockData* method), 343
`calculate_dew_point_temperature()` (*idaes.core.property_base.StateBlockData* method), 343
`calculate_scaling_factors()` (*idaes.generic_models.unit_models.valve.ValveData* method), 586
`canv` (*idaes.apps.matopt.opt.mat_modeling.MaterialDescriptor* attribute), 202
`canv` (*idaes.apps.matopt.opt.mat_modeling.MatOptModel* attribute), 203
`Canvas` (class in *idaes.apps.matopt.materials.canvas*), 201
command line option
 `--help`, 142
 `--quiet`, 142
 `--verbose`, 142
 `-q`, 142
 `-v`, 142
`Component` (class in *idaes.core.components*), 358
`confDs` (*idaes.apps.matopt.opt.mat_modeling.MaterialDescriptor* attribute), 203
`confDs` (*idaes.apps.matopt.opt.mat_modeling.MatOptModel* attribute), 203
`confint_regression()` (*idaes.surrogate.pysmo.polynomial_regression.PolynomialRegression* method), 177
`constraint_autoscale_large_jac()` (in module *idaes.core.util.scaling*), 412
`constraint_scaling_transform()` (in module *idaes.core.util.scaling*), 410
`constraint_scaling_transform_undo()` (in module *idaes.core.util.scaling*), 410
`ControlVolume0DBlock` (class in *idaes.core.control_volume0d*), 306
`ControlVolume0DBlockData` (class in *idaes.core.control_volume0d*), 307
`ControlVolume1DBlock` (class in *idaes.core.control_volume1d*), 320
`ControlVolume1DBlockData` (class in *idaes.core.control_volume1d*), 323
`copy_non_time_indexed_values()` (in module *idaes.core.util.dyn_utils*), 363
`copy_values_at_time()` (in module *idaes.core.util.dyn_utils*), 363
`CRADA`, 255
`create_inlet_list()` (*idaes.generic_models.unit_models.mixer.MixerData* method), 542
`create_outlet_list()` (*idaes.generic_models.unit_models.separator.SeparatorData* method), 566
`create_stream_table_dataframe()` (in module *idaes.core.util.tables*), 416
`CSTR` (class in *idaes.generic_models.unit_models.cstr*), 487
`CSTRData` (class in *idaes.generic_models.unit_models.cstr*), 490
`CubicParameterData` (class in *idaes.generic_models.properties.cubic_eos.cubic_prop_pack*), 446
`CubicStateBlock` (class in *idaes.generic_models.properties.cubic_eos.cubic_prop_pack*), 446
`CubicStateBlockData` (class in *idaes.generic_models.properties.cubic_eos.cubic_prop_pack*), 447
`CVTSampling` (class in *idaes.surrogate.pysmo.sampling*), 197
D
`Data Management (IDAES-DMF)`, 14
`data_scaling()` (*idaes.surrogate.pysmo.polynomial_regression.FeatureScaling* static method), 174
`data_scaling_minmax()` (*idaes.surrogate.pysmo.radial_basis_function.FeatureScaling* static method), 180
`data_unscaling()` (*idaes.surrogate.pysmo.polynomial_regression.FeatureScaling* static method), 174
`data_unscaling_minmax()` (*idaes.surrogate.pysmo.radial_basis_function.FeatureScaling* static method), 181
`deactivate_constraints_unindexed_by()` (in module *idaes.core.util.dyn_utils*), 363
`deactivate_model_at()` (in module *idaes.core.util.dyn_utils*), 364
`deactivated_blocks_set()` (in module *idaes.core.util.model_statistics*), 394

`deactivated_constraints_generator()` (in module `idaes.core.util.model_statistics`), 394
`deactivated_constraints_set()` (in module `idaes.core.util.model_statistics`), 394
`deactivated_equalities_generator()` (in module `idaes.core.util.model_statistics`), 395
`deactivated_equalities_set()` (in module `idaes.core.util.model_statistics`), 395
`deactivated_inequalities_generator()` (in module `idaes.core.util.model_statistics`), 395
`deactivated_inequalities_set()` (in module `idaes.core.util.model_statistics`), 395
`deactivated_objectives_generator()` (in module `idaes.core.util.model_statistics`), 395
`deactivated_objectives_set()` (in module `idaes.core.util.model_statistics`), 395
`declare_process_block_class()` (in module `idaes.core.process_block`), 297
`define_display_vars()` (`idaes.core.property_base.StateBlockData` method), 344
`define_display_vars()` (`idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicStateBlockData` method), 447
`define_metadata()` (`idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicEquationData` class method), 446
`define_metadata()` (`idaes.generic_models.properties.interrogator.properties_interrogator.PropertyInterrogatorData` class method), 481
`define_metadata()` (`idaes.generic_models.properties.interrogator.reactions_interrogator.ReactionInterrogatorData` class method), 484
`define_port_members()` (`idaes.core.property_base.StateBlockData` method), 344
`define_state_vars()` (`idaes.core.property_base.StateBlockData` method), 344
`define_state_vars()` (`idaes.generic_models.properties.activity_coeff_models.activity_coeff_prop_pack.ActivityCoeffStateBlockData` method), 456
`define_state_vars()` (`idaes.generic_models.properties.cubic_eos.cubic_prop_pack.CubicStateBlockData` method), 447
`degrees_of_freedom()` (in module `idaes.core.util.model_statistics`), 390
`delta_temperature_amtd_callback()` (in module `idaes.generic_models.unit_models.heat_exchanger`), 524
`delta_temperature_lmtd_callback()` (in module `idaes.generic_models.unit_models.heat_exchanger`), 524
`delta_temperature_underwood_callback()` (in module `idaes.generic_models.unit_models.heat_exchanger`), 524
`derivative_variables_set()` (in module `idaes.core.util.model_statistics`), 396
Design (class in `idaes.apps.matopt.materials.design`), 201
DMF
 `dmf`, 89
 `dmf`
 DMF, 89
 Help, 91
 `dmf` command line option
 `--quiet`, 93
 `--verbose`, 93
 `-q`, 93
 `-v`, 93
 `dmf-find` command line option
 `--by` value, 96
 `--created` value, 96
 `--file` value, 96
 `--modified` value, 96
 `--name` value, 96
 `--type` value, 96
 `dmf-info` command line option
 `-f`, `--format` value, 97
 identifier, 97
 `--create`, 99
 `--desc`, 99
 path, 99
 `dmf-ls` command line option
 `-S`, `--sort`, 101
 `--color`, 101
 `--no-color`, 101
 `--no-prefix`, 101
 `-r`, `--reverse`, 101
 `-s`, `--show`, 101
 `--contained` resource, 103
 `--derived` resource, 103
 `--no-copy`, 103
 `--no-unique`, 103
 `--prev` resource, 103
 `--strict`, 103
 used resource, 103
 `--version`, 103
 `-t`, `--type`, 103
 related command line option
 `--color`, 107
 `--no-color`, 107


```

--no-unicode, 107
--unicode, 107
-d, --direction, 107
dmf-rm command line option
--list, --no-list, 109
--multiple, 109
-y, --yes, 109
identifier, 109
dmf-status command line option
--color, 111
--no-color, 111
-a, --all, 111
-s, --show info, 111
Downcomer
    idaes.power_generation.unit_models.downcomer, 665
Drum
    idaes.power_generation.unit_models.drum, 661
Drum1D
    idaes.power_generation.unit_models.drum1D, 694
E
EquilibriumReactor (class in flowsheet() (idaes.core.process_base.ProcessBlockData
    idaes.generic_models.unit_models.equilibrium_reactor), method), 300
    FlowsheetBlock
EquilibriumReactorData (class in idaes.core.flowsheet_model, 300
    idaes.generic_models.unit_models.equilibrium_reactor), 494
    FlowsheetBlock (class in idaes.core.flowsheet_model), 304
expressions_set() (in module idaes.core.util.model_statistics), 396
F
FeatureScaling (class in idaes.surrogate.pysmo.polynomial_regression), 174
FeatureScaling (class in idaes.surrogate.pysmo.radial_basis_function), 180
Feed (class in idaes.generic_models.unit_models.feed), 496
FeedData (class in idaes.generic_models.unit_models.feed), 497
FeedFlash (class in idaes.generic_models.unit_models.feed_flash), 499
FeedFlashData (class in idaes.generic_models.unit_models.feed_flash), 500
find_comp_in_block() (in module idaes.core.util.dyn_utils), 364
find_comp_in_block_at_time() (in module idaes.core.util.dyn_utils), 365
fix_initial_conditions()
    (idaes.core.process_base.ProcessBlockData
    method), 300
fix_state_vars() (in module idaes.core.util.initialization), 371
fix_vars_unindexed_by() (in module idaes.core.util.dyn_utils), 365
fixed_unused_variables_set() (in module idaes.core.util.model_statistics), 396
fixed_variables_generator() (in module idaes.core.util.model_statistics), 396
fixed_variables_in_activated_equalities_set()
    (in module idaes.core.util.model_statistics), 396
fixed_variables_only_in_inequalities()
    (in module idaes.core.util.model_statistics), 397
fixed_variables_set() (in module idaes.core.util.model_statistics), 397
Flash (class in idaes.generic_models.unit_models.flash), 503
FlashData (class in idaes.generic_models.unit_models.flash), 506
FlowsheetBlockData
    idaes.core.flowsheet_model, 300
    FlowsheetBlock (class in idaes.core.flowsheet_model), 304
FlowsheetBlockData (class in idaes.core.flowsheet_model), 302
FlueGasParameterBlock
    idaes.power_generation.properties.IdealProp_Flu
    714
FlueGasParameterData
    idaes.power_generation.properties.IdealProp_Flu
    714
FlueGasStateBlock
    idaes.power_generation.properties.IdealProp_Flu
    714
FlueGasStateBlockData
    idaes.power_generation.properties.IdealProp_Flu
    714
from_json() (in module idaes.core.util.model_serializer), 380
FWHOD
    idaes.power_generation.unit_models.feedwater_he
    592
FWHODDynamic
    idaes.power_generation.unit_models.feedwater_he
    711

```

FWHCondensing0D (idaes.core.property_base.StateBlockData
 idaes.power_generation.unit_models.feedwater_heater_0D, 344D,
 595 get_enthalpy_flow_terms()
 FWHCondensing0D (class in (idaes.generic_models.properties.activity_coeff_models.activity_c
 idaes.power_generation.unit_models.feedwater_heater_0D)method), 457
 596 get_enthalpy_flow_terms()
 FWHCondensing0DData (class in (idaes.generic_models.properties.cubic_eos.cubic_prop_pack.Cu
 idaes.power_generation.unit_models.feedwater_heater_0D)method), 447
 601 get_feature_vector()
 (idaes.surrogate.pysmo.kriging.KrigingModel
 method), 188
G
 generate_expression() get_feature_vector()
 (idaes.surrogate.pysmo.kriging.KrigingModel (idaes.surrogate.pysmo.polynomial_regression.PolynomialRegres
 method), 188 method), 177
 generate_expression() get_feature_vector()
 (idaes.surrogate.pysmo.polynomial_regression.PolynomialRegression (idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunc
 method), 177 method), 183
 generate_expression() get_fixed_dict() (in module
 (idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunction (idaes.core.util.dyn_utils), 366
 method), 183 get_implicit_index_of_set() (in module
 generate_table() (in module idaes.core.util.dyn_utils), 366
 idaes.core.util.tables), 417 get_index_of_set() (in module
 get_activity_dict() (in module idaes.core.util.dyn_utils), 366
 idaes.core.util.dyn_utils), 365 get_location_of_coordinate_set() (in mod-
 get_class_attr_list() ule idaes.core.util.dyn_utils), 367
 (idaes.core.util.model_serializer.StoreSpec get_material_density_terms()
 method), 383 (idaes.core.property_base.StateBlockData
 get_component() (idaes.core.property_base.PhysicalParameterBlock)method), 344
 method), 340 get_material_density_terms()
 get_constraint_transform_applied_scaling_factor (idaes.generic_models.properties.activity_coeff_models.activity_c
 (in module idaes.core.util.scaling), 410 method), 457
 get_costing() (idaes.core.flowsheet_model.FlowsheetBlockData get_material_density_terms()
 method), 303 (idaes.generic_models.properties.cubic_eos.cubic_prop_pack.Cu
 method), 447
 get_data_class_attr_list() get_material_diffusion_terms()
 (idaes.core.util.model_serializer.StoreSpec (idaes.core.property_base.StateBlockData
 method), 383 method), 344
 get_default_scaling() get_material_flow_basis()
 (idaes.core.property_base.PhysicalParameterBlock)method), 344
 method), 340 (idaes.core.property_base.StateBlockData
 get_derivatives_at() (in module method), 344
 idaes.core.util.dyn_utils), 366 get_material_flow_basis()
 get_energy_density_terms() (idaes.generic_models.properties.activity_coeff_models.activity_c
 (idaes.core.property_base.StateBlockData method), 457
 method), 344 get_material_flow_basis()
 get_energy_density_terms() (idaes.generic_models.properties.cubic_eos.cubic_prop_pack.Cu
 (idaes.generic_models.properties.activity_coeff_models.activity_cmethod), 447
 method), 457 get_material_flow_terms()
 get_energy_density_terms() (idaes.core.property_base.StateBlockData
 (idaes.generic_models.properties.cubic_eos.cubic_prop_pack.Cu)method), 344
 method), 447 get_material_flow_terms()
 get_energy_diffusion_terms() (idaes.generic_models.properties.activity_coeff_models.activity_c
 (idaes.core.property_base.StateBlockData method), 457
 method), 344 get_material_flow_terms()
 get_enthalpy_flow_terms() (idaes.generic_models.properties.cubic_eos.cubic_prop_pack.Cu

method), 447
 get_mixed_state_block() (idaes.generic_models.unit_models.mixer.MixerData method), 542
 get_mixed_state_block() (idaes.generic_models.unit_models.separator.SeparatorData method), 566
 get_phase() (idaes.core.property_base.PhysicalParameterBlock method), 341
 get_phase_component_set() (idaes.core.property_base.PhysicalParameterBlock method), 341
 get_reaction_rate_basis() (idaes.core.reaction_base.ReactionBlockDataBase method), 351
 get_scaling_factor() (in module idaes.core.util.scaling), 409
 GibbsReactor (class in HelmholtzStateBlock idaes.generic_models.unit_models.gibbs_reactor), 508
 GibbsReactorData (class in HelmIsentropicTurbine idaes.generic_models.unit_models.gibbs_reactor), 510
 Graphical User Interfaces, 14
H
 HaltonSampling (class in HelmholtzStateBlock idaes.surrogate.pysmo.sampling), 193
 HammersleySampling (class in HelmholtzStateBlock idaes.surrogate.pysmo.sampling), 195
 Heater (idaes.generic_models.unit_models.heater, 511
 Heater (class in idaes.generic_models.unit_models.heater), 512
 HeaterData (class in HelmholtzStateBlock idaes.generic_models.unit_models.heater), 515
 HeatExchanger (idaes.generic_models.unit_models.heat_exchanger, 515
 HeatExchanger (class in HelmholtzStateBlock idaes.generic_models.unit_models.heat_exchanger), 518
 HeatExchanger1D (class in HelmholtzStateBlock idaes.generic_models.unit_models.heat_exchanger_1D), 528
 HeatExchanger1DData (class in HelmholtzStateBlock idaes.generic_models.unit_models.heat_exchanger_1D), 534
 HeatExchangerCrossFlow2D_Header (idaes.power_generation.unit_models.boiler_heat_exchanger_2D), 678
 HeatExchangerCrossFlow2D_Header (class in HelmholtzStateBlock idaes.power_generation.unit_models.boiler_heat_exchanger_2D), 689
 HeatExchangerCrossFlow2D_HeaderData (class in idaes.power_generation.unit_models.boiler_heat_exchanger_2D), 693
 HeatExchangerData (class in idaes.generic_models.unit_models.heat_exchanger), 522
 HeatExchangerWith3Streams (idaes.power_generation.unit_models.heat_exchanger_3streams), 706
 HeatExchangerWith3Streams (class in idaes.power_generation.unit_models.heat_exchanger_3streams), 708
 HeatExchangerWith3StreamsData (class in idaes.power_generation.unit_models.heat_exchanger_3streams), 710
 HelmholtzStateBlock (class in HelmholtzStateBlock idaes.generic_models.properties.helmholtz.helmholtz_state_block), 457
 HelmIsentropicTurbine (idaes.power_generation.unit_models.helm.turbine), 601
 HelmIsentropicTurbine (class in idaes.power_generation.unit_models.helm.turbine), 602
 HelmIsentropicTurbineData (class in idaes.power_generation.unit_models.helm.turbine), 605
 HelmPhaseSeparator (idaes.power_generation.unit_models.helm.phase_separator), 660
 HelmTurbineInletStage (idaes.power_generation.unit_models.helm.turbine_inlet), 610
 HelmTurbineInletStageData (class in idaes.power_generation.unit_models.helm.turbine_inlet), 612
 HelmTurbineMultistage (idaes.power_generation.unit_models.helm.turbine_multistage), 625
 HelmTurbineMultistage (class in idaes.power_generation.unit_models.helm.turbine_multistage), 631
 HelmTurbineMultistageData (class in idaes.power_generation.unit_models.helm.turbine_multistage), 633
 HelmTurbineOutletStage (idaes.power_generation.unit_models.helm.turbine_outlet), 613
 HelmTurbineOutletStage (class in idaes.power_generation.unit_models.helm.turbine_outlet), 613

[idaes.power_generation.unit_models.helm.turbine_outlet](#), module, 201
[617](#)
[HelmTurbineOutletStageData](#) (class in module, 201
[idaes.power_generation.unit_models.helm.turbine_outlet](#)), apps.matopt.materials.lattices.lattice
[619](#) module, 201
[HelmTurbineStage](#) [idaes.apps.matopt.opt.mat_modeling](#)
[idaes.power_generation.unit_models.helm.turbine_outlet](#) module, 201
[620](#) [idaes.core.components](#)
[HelmTurbineStage](#) (class in module, 358
[idaes.power_generation.unit_models.helm.turbine_outlet](#)), core.control_volume0d
[622](#) module, 306
[HelmTurbineStageData](#) (class in [idaes.core.control_volumeld](#)
[idaes.power_generation.unit_models.helm.turbine_outlet](#)), module, 320
[624](#) [idaes.core.flowsheet_model](#)
[HelmValve](#) [FlowsheetBlock](#), 300
[idaes.power_generation.unit_models.helm.turbine_outlet](#) [FlowsheetBlockData](#), 300
[635](#) module, 302
[HelmValve](#) (class in [idaes.core.phases](#)
[idaes.power_generation.unit_models.helm.valve_steam](#)), module, 361
[637](#) [idaes.core.process_base](#)
[HelmValveData](#) (class in module, 299
[idaes.power_generation.unit_models.helm.valve_steam](#)), core.process_block
[639](#) module, 297
[Help](#) [idaes.core.property_base](#)
[dmf](#), 91 module, 340
[Home](#) [idaes.core.reaction_base](#)
[idaes](#), 1 module, 349
[homotopy\(\)](#) (in module [idaes.core.util.homotopy](#)), 370 [idaes.core.unit_model](#)
[htpx\(\)](#) (in module [idaes.generic_models.properties.iapws95](#)), module, 354
[460](#) [idaes.core.util.dyn_utils](#)
[htpx\(\)](#) (in module [idaes.generic_models.properties.swco2](#)), module, 363
[466](#) [idaes.core.util.homotopy](#)
[I](#) module, 370
[Iapws95ParameterBlock](#) (class in module, 371
[idaes.generic_models.properties.iapws95](#)), [idaes.core.util.misc](#)
[462](#) module, 438
[Iapws95ParameterBlockData](#) (class in [idaes.core.util.model_serializer](#)
[idaes.generic_models.properties.iapws95](#)), module, 376
[464](#) [idaes.core.util.model_statistics](#)
[Iapws95StateBlock](#) module, 392
[idaes.generic_models.properties.iapws95](#) [idaes.core.util.scaling](#)
[458](#) module, 407
[Iapws95StateBlock](#) (class in [idaes.core.util.tables](#)
[idaes.generic_models.properties.iapws95](#)), module, 415
[460](#) [idaes.core.util.unit_costing](#)
[Iapws95StateBlockData](#) (class in module, 424
[idaes.generic_models.properties.iapws95](#)), [idaes.gas_solid_contactors.unit_models.bubbling_flow](#)
[461](#) module, 759
[idaes](#) [idaes.gas_solid_contactors.unit_models.moving_bed](#)
[Home](#), 1 module, 770
[idaes.apps.caprese.nmpc](#) [idaes.generic_models.properties.activity_coeff_model](#)
[module](#), 206 module, 454
[idaes.apps.matopt.materials.canvas](#) [idaes.generic_models.properties.cubic_eos.cubic_pro](#)

module, [446](#)
 idaes.generic_models.properties.helmholtz.helmholtzStateBlock, [457](#)
 module, [457](#)
 idaes.generic_models.properties.iapws95.Iapws95StateBlock, [458](#)
 module, [458](#)
 idaes.generic_models.properties.swco2.SWC02StateBlock, [465](#)
 idaes.generic_models.unit_models.cstr module, [487](#)
 idaes.generic_models.unit_models.equilibrium module, [491](#)
 idaes.generic_models.unit_models.feed module, [495](#)
 idaes.generic_models.unit_models.feed_flash module, [499](#)
 idaes.generic_models.unit_models.flash module, [503](#)
 idaes.generic_models.unit_models.gibbs_reducer module, [508](#)
 idaes.generic_models.unit_models.heat_exchanger.HeatExchanger, [515](#)
 idaes.generic_models.unit_models.heat_exchanger module, [528](#)
 idaes.generic_models.unit_models.heater.Heater, [511](#)
 module, [511](#)
 idaes.generic_models.unit_models.mixer module, [538](#)
 idaes.generic_models.unit_models.plug_flow_reactor module, [545](#)
 idaes.generic_models.unit_models.pressure_changer module, [551](#)
 idaes.generic_models.unit_models.product module, [556](#)
 idaes.generic_models.unit_models.separator module, [561](#)
 idaes.generic_models.unit_models.statejunction module, [568](#)
 idaes.generic_models.unit_models.stoichiometric_reactor module, [571](#)
 idaes.generic_models.unit_models.translator module, [575](#)
 idaes.generic_models.unit_models.valve module, [578](#)
 Valve, [578](#)
 idaes.power_generation.control.pid_controller.Proportional-Integral-Derivative (PID) Controller, [742](#)
 idaes.power_generation.properties.IdealProperties module, [713](#)
 FlueGasParameterBlock, [714](#)
 FlueGasParameterData, [714](#)
 FlueGasStateBlock, [714](#)
 FlueGasStateBlockData, [714](#)
 idaes.power_generation.unit_models.boiler_fireside.BoilerFireside, [671](#)
 idaes.power_generation.unit_models.boiler_heat_exchanger.BoilerHeatExchanger, [640](#)
 idaes.power_generation.unit_models.boiler_heat_exchanger.HeatExchangerCrossFlow2D_Header, [678](#)
 idaes.power_generation.unit_models.downcomer.Downcomer, [665](#)
 idaes.power_generation.unit_models.drum.Drum, [661](#)
 idaes.power_generation.unit_models.drum1D.Drum1D, [694](#)
 idaes.power_generation.unit_models.feedwater_heater.FWH0D, [592](#)
 FWHCondensing0D, [595](#)
 module, [595](#)
 idaes.power_generation.unit_models.feedwater_heater.FWH0DDynamic, [711](#)
 idaes.power_generation.unit_models.heat_exchanger_1.HeatExchangerWith3Streams, [706](#)
 idaes.power_generation.unit_models.helm.phase_separator.HelmPhaseSeparator, [660](#)
 idaes.power_generation.unit_models.helm.turbine module, [601](#)
 idaes.power_generation.unit_models.helm.turbine_inlet.HelmTurbineInletStage, [606](#)
 module, [606](#)
 idaes.power_generation.unit_models.helm.turbine_multistage.HelmTurbineMultistage, [625](#)
 module, [625](#)
 idaes.power_generation.unit_models.helm.turbine_outlet.HelmTurbineOutletStage, [613](#)
 module, [613](#)
 idaes.power_generation.unit_models.helm.turbine_stage.HelmIsentropicTurbine, [601](#)
 HelmTurbineStage, [620](#)
 module, [620](#)
 idaes.power_generation.unit_models.helm.valve_steam.HelmValve, [635](#)
 module, [635](#)
 idaes.power_generation.unit_models.steamheater.SteamHeater, [667](#)
 idaes.power_generation.unit_models.waterpipe.WaterPipe, [703](#)
 idaes.power_generation.unit_models.waterwall.WaterwallSection, [653](#)
 idaes.surrogate.pysmo.kriging module, [187](#)
 idaes.surrogate.pysmo.polynomial_regression module, [187](#)
 idaes.surrogate.pysmo.radial_basis_function module, [180](#)

```

idaes.unit_models.heat_exchanger
    Proportional-Integral-Derivative
        (PID) Controller, 587
IDAES-AI, 14
IDAES-CMF, 14
IDAES-Core, 13
IDAES-Design, 14
IDAES-Enterprise, 14
IDAES-IP, 13
IDAES-Materials, 14
IDAES-Operations, 14
IDAES-UQ, 14
idaes-bin-directory command line
    option
    --create, 135
    --exists, 135
    --help, 135
idaes-copyright command line option
    --help, 136
idaes-data-directory command line
    option
    --create, 137
    --exists, 137
    --help, 137
idaes-get-examples command line option
    -I, 137
    -N, 138
    -U, 138
    -V, 138
    --help, 137
    --list-releases, 137
    --no-download, 138
    --no-install, 137
    --unstable, 138
    --version TEXT, 138
    -d, --dir TEXT, 137
    -l, 137
idaes-get-extensions command line
    option
    --help, 139
    --url, 139
idaes-lib-directory command line
    option
    --create, 140
    --exists, 140
    --help, 140
identifier
    dmf-info command line option, 97
    dmf-rm command line option, 109
IndexedVarLikeExpression (class in
    idaes.core.util.misc), 439
init_isentropic()
    (idaes.generic_models.unit_models.pressure_changer.PressureChangerData
    method), 554
initialize() (idaes.core.control_volume0d.ControlVolume0DBlockData
    method), 313
initialize() (idaes.core.control_volume1d.ControlVolume1DBlockData
    method), 328
initialize() (idaes.core.property_base.StateBlock
    method), 345
initialize() (idaes.core.reaction_base.ReactionBlockBase
    method), 352
initialize() (idaes.core.unit_model.UnitModelBlockData
    method), 356
initialize() (idaes.gas_solid_contactors.unit_models.bubbling_fluidi
    method), 764
initialize() (idaes.gas_solid_contactors.unit_models.moving_bed.ML
    method), 775
initialize() (idaes.generic_models.unit_models.feed.FeedData
    method), 497
initialize() (idaes.generic_models.unit_models.heat_exchanger.Heat
    method), 522
initialize() (idaes.generic_models.unit_models.heat_exchanger_1D.
    method), 534
initialize() (idaes.generic_models.unit_models.mixer.MixerData
    method), 542
initialize() (idaes.generic_models.unit_models.pressure_changer.Pre
    method), 555
initialize() (idaes.generic_models.unit_models.product.ProductData
    method), 557
initialize() (idaes.generic_models.unit_models.separator.SeparatorL
    method), 566
initialize() (idaes.generic_models.unit_models.statejunction.StateJun
    method), 569
initialize() (idaes.generic_models.unit_models.translator.Translator
    method), 576
initialize() (idaes.generic_models.unit_models.valve.ValveData
    method), 586
initialize() (idaes.power_generation.unit_models.boiler_heat_excha
    method), 652
initialize() (idaes.power_generation.unit_models.boiler_heat_excha
    method), 694
initialize() (idaes.power_generation.unit_models.feedwater_heater_
    method), 601
initialize() (idaes.power_generation.unit_models.heat_exchanger_3.
    method), 711
initialize() (idaes.power_generation.unit_models.helm.turbine.Helm
    method), 605
initialize() (idaes.power_generation.unit_models.helm.turbine_inlet
    method), 612
initialize() (idaes.power_generation.unit_models.helm.turbine_mult
    method), 633
initialize() (idaes.power_generation.unit_models.helm.turbine_outle
    method), 619
initialize() (idaes.power_generation.unit_models.helm.turbine_staga
    method), 624
initialize() (idaes.power_generation.unit_models.helm.valve_steam.
    method), 640

```

[initialize_by_time_element\(\)](#) (in module [MaterialDescriptor](#) (class in [idaes.core.util.initialization](#)), 372 [idaes.apps.matopt.opt.mat_modeling](#)), 202
[integer](#) ([idaes.apps.matopt.opt.mat_modeling.MaterialDescriptorModel](#) (class in [idaes.apps.matopt.opt.mat_modeling](#)), 203
[attribute](#)), 203
[is_flowsheet\(\)](#) ([idaes.core.flowsheet_model.FlowsheetBlockData](#) (class in [idaes.apps.matopt.opt.mat_modeling.MatOptModel](#)
[method](#)), 303 [method](#)), 203
[is_property_constructed\(\)](#) [MBR](#) (class in [idaes.gas_solid_contactors.unit_models.moving_bed](#)),
[\(idaes.core.property_base.StateBlockData](#) 770
[method](#)), 344 [MBRData](#) (class in [idaes.gas_solid_contactors.unit_models.moving_bed](#)),
[is_property_constructed\(\)](#) 775
[\(idaes.core.reaction_base.ReactionBlockDataBase](#) [min_scaling_factor\(\)](#) (in module
[method](#)), 351 [idaes.core.util.scaling](#)), 414
[isfixed\(\)](#) ([idaes.core.util.model_serializer.StoreSpec](#) [minimize\(\)](#) ([idaes.apps.matopt.opt.mat_modeling.MatOptModel](#)
[class method](#)), 384 [method](#)), 204
[Mixer](#) (class in [idaes.generic_models.unit_models.mixer](#)),
538
[K](#)
[KrigingModel](#) (class in [idaes.surrogate.pysmo.kriging](#)), 187 [MixerData](#) (class in
[idaes.generic_models.unit_models.mixer](#)),
540
[L](#)
[large_residuals_set\(\)](#) (in module [idaes.core.util.model_statistics](#)), 397 [model_check\(\)](#) ([idaes.core.control_volume0d.ControlVolume0DBlockData](#)
[method](#)), 313
[LatinHypercubeSampling](#) (class in [idaes.surrogate.pysmo.sampling](#)), 190 [model_check\(\)](#) ([idaes.core.control_volume1d.ControlVolume1DBlockData](#)
[method](#)), 329
[Lattice](#) (class in [idaes.apps.matopt.materials.lattices.lattice](#)), [model_check\(\)](#) ([idaes.core.flowsheet_model.FlowsheetBlockData](#)
[method](#)), 303
201 [model_check\(\)](#) ([idaes.core.unit_model.UnitModelBlockData](#)
[method](#)), 357
[list_models_requiring_property\(\)](#) [model_check\(\)](#) ([idaes.generic_models.properties.activity_coeff_models](#)
[\(idaes.generic_models.properties.interrogator.properties_interrogator.PropertyInterrogatorData](#)
[method](#)), 482 [method](#)), 457
[list_models_requiring_property\(\)](#) [model_check\(\)](#) ([idaes.generic_models.properties.cubic_eos.cubic_prop](#)
[\(idaes.generic_models.properties.interrogator.reactions_interrogator.ReactionInterrogatorData](#)
[method](#)), 484 [method](#)), 447
[list_properties_required_by_model\(\)](#) [model_check\(\)](#) ([idaes.generic_models.unit_models.mixer.MixerData](#)
[\(idaes.generic_models.properties.interrogator.properties_interrogator.PropertyInterrogatorData](#)
[method](#)), 482 [method](#)), 542
[list_properties_required_by_model\(\)](#) [model_check\(\)](#) ([idaes.generic_models.unit_models.pressure_changer.P](#)
[\(idaes.generic_models.properties.interrogator.reactions_interrogator.ReactionInterrogatorData](#)
[method](#)), 484 [method](#)), 555
[list_required_properties\(\)](#) [model_check\(\)](#) ([idaes.generic_models.unit_models.separator.Separator](#)
[\(idaes.generic_models.properties.interrogator.properties_interrogator.PropertyInterrogatorData](#)
[method](#)), 482 [method](#)), 567
[list_required_properties\(\)](#) [module](#) [idaes.apps.caprese.nmpc](#), 206
[\(idaes.generic_models.properties.interrogator.reactions_interrogator.ReactionInterrogatorData](#)
[method](#)), 484 [idaes.apps.matopt.materials.canvas](#),
201
[lock_attribute_creation_context\(\)](#) [idaes.apps.matopt.materials.design](#),
201
[\(idaes.core.property_base.StateBlockData](#) [idaes.apps.matopt.materials.lattices.lattice](#),
[method](#)), 345 201
[lock_attribute_creation_context\(\)](#) [idaes.apps.matopt.opt.mat_modeling](#),
203
[\(idaes.core.reaction_base.ReactionBlockDataBase](#) [idaes.core.components](#), 358
[method](#)), 351 [idaes.core.control_volume0d](#), 306
[idaes.core.control_volume1d](#), 320
[idaes.core.flowsheet_model](#), 302
[idaes.core.phases](#), 361
[M](#)
[map_scaling_factor\(\)](#) (in module [idaes.core.util.scaling](#)), 414

idaes.core.process_base, 299
 idaes.core.process_block, 297
 idaes.core.property_base, 340
 idaes.core.reaction_base, 349
 idaes.core.unit_model, 354
 idaes.core.util.dyn_utils, 363
 idaes.core.util.homotopy, 370
 idaes.core.util.initialization, 371
 idaes.core.util.misc, 438
 idaes.core.util.model_serializer, 376
 idaes.core.util.model_statistics, 392
 idaes.core.util.scaling, 407
 idaes.core.util.tables, 415
 idaes.core.util.unit_costing, 424
 idaes.gas_solid_contactors.unit_models.bubbling_power_generation, 759
 idaes.gas_solid_contactors.unit_models.moving_bed, 770
 idaes.generic_models.properties.activity_coeff_model, 454
 idaes.generic_models.properties.cubic_eos, 446
 idaes.generic_models.properties.helmholtz, 457
 idaes.generic_models.properties.iapws95, 458
 idaes.generic_models.properties.swco2, 465
 idaes.generic_models.unit_models.cstr, 487
 idaes.generic_models.unit_models.equilibrium, 491
 idaes.generic_models.unit_models.feed_NMPCSim, 495
 idaes.generic_models.unit_models.feed_flash, 499
 idaes.generic_models.unit_models.flash, 503
 idaes.generic_models.unit_models.gibbs_reaction, 508
 idaes.generic_models.unit_models.heat_exchanger, 528
 idaes.generic_models.unit_models.heater, 511
 idaes.generic_models.unit_models.mixer, 538
 idaes.generic_models.unit_models.plugin_reactor, 545
 idaes.generic_models.unit_models.pressure_changer, 551
 idaes.generic_models.unit_models.product, 556
 idaes.generic_models.unit_models.separator, 561
 idaes.generic_models.unit_models.statejunction, 568
 idaes.generic_models.unit_models.stoichiometric, 571
 idaes.generic_models.unit_models.translator, 575
 idaes.generic_models.unit_models.valve, 578
 idaes.power_generation.unit_models.feedwater_heater, 595
 idaes.power_generation.unit_models.helm.turbine, 601
 idaes.power_generation.unit_models.helm.turbine, 606
 idaes.power_generation.unit_models.helm.turbine, 625
 idaes.power_generation.unit_models.helm.turbine, 613
 idaes.power_generation.unit_models.helm.turbine, 620
 idaes.power_generation.unit_models.helm.valve, 635
 idaes.surrogate.pysmo.kriging, 187
 idaes.surrogate.pysmo.polynomial_regression, 173
 idaes.surrogate.pysmo.radial_basis_function, 180

N

name (*idaes.apps.matopt.opt.mat_modeling.MaterialDescriptor* attribute), 202
 NDA, 255
 NMPCSim (*class in idaes.apps.caprese.nmpc*), 206
 number_activated_blocks() (*in module idaes.core.util.model_statistics*), 397
 number_activated_constraints() (*in module idaes.core.util.model_statistics*), 398
 number_activated_equalities() (*in module idaes.core.util.model_statistics*), 398
 number_activated_inequalities() (*in module idaes.core.util.model_statistics*), 398
 number_activated_objectives() (*in module idaes.core.util.model_statistics*), 398
 number_active_variables_in_deactivated_blocks() (*in module idaes.core.util.model_statistics*), 398
 number_deactivated_blocks() (*in module idaes.core.util.model_statistics*), 398
 number_deactivated_constraints() (*in module idaes.core.util.model_statistics*), 399
 number_deactivated_equalities() (*in module idaes.core.util.model_statistics*), 399

`number_deactivated_inequalities()` (in module `idaes.core.util.model_statistics`), 399
`number_deactivated_objectives()` (in module `idaes.core.util.model_statistics`), 399
`number_derivative_variables()` (in module `idaes.core.util.model_statistics`), 399
`number_expressions()` (in module `idaes.core.util.model_statistics`), 400
`number_fixed_unused_variables()` (in module `idaes.core.util.model_statistics`), 400
`number_fixed_variables()` (in module `idaes.core.util.model_statistics`), 400
`number_fixed_variables_in_activated_equalities()` (in module `idaes.core.util.model_statistics`), 400
`number_fixed_variables_only_in_inequalities()` (in module `idaes.core.util.model_statistics`), 400
`number_large_residuals()` (in module `idaes.core.util.model_statistics`), 400
`number_total_blocks()` (in module `idaes.core.util.model_statistics`), 401
`number_total_constraints()` (in module `idaes.core.util.model_statistics`), 401
`number_total_equalities()` (in module `idaes.core.util.model_statistics`), 401
`number_total_inequalities()` (in module `idaes.core.util.model_statistics`), 401
`number_total_objectives()` (in module `idaes.core.util.model_statistics`), 401
`number_unfixed_variables()` (in module `idaes.core.util.model_statistics`), 401
`number_unfixed_variables_in_activated_equalities()` (in module `idaes.core.util.model_statistics`), 402
`number_unused_variables()` (in module `idaes.core.util.model_statistics`), 402
`number_variables()` (in module `idaes.core.util.model_statistics`), 402
`number_variables_in_activated_constraints()` (in module `idaes.core.util.model_statistics`), 402
`number_variables_in_activated_equalities()` (in module `idaes.core.util.model_statistics`), 402
`number_variables_in_activated_inequalities()` (in module `idaes.core.util.model_statistics`), 403
`number_variables_near_bounds()` (in module `idaes.core.util.model_statistics`), 403
`number_variables_only_in_inequalities()` (in module `idaes.core.util.model_statistics`), 403

O
`optimize()` (`idaes.apps.matopt.opt.mat_modeling.MatOptModel` method), 204

P
`partition_outlet_flows()` (`idaes.generic_models.unit_models.separator.SeparatorData` method), 567
`path` dmf-init command line option, 99
`path_from_block()` (in module `idaes.core.util.dyn_utils`), 367
`PFR` (class in `idaes.generic_models.unit_models.plug_flow_reactor`), 545
`PFRData` (class in `idaes.generic_models.unit_models.plug_flow_reactor`), 548
`Phase` (class in `idaes.core.phases`), 361
`PhysicalParameterBlock` (class in `idaes.core.property_base`), 340
`PIDBlock` (class in `idaes.generic_models.control.pid_controller`), 589
`PIDBlockData` (class in `idaes.generic_models.control.pid_controller`), 590
`PolynomialRegression` (class in `idaes.surrogate.pysmo.polynomial_regression`), 174
`populate()` (`idaes.apps.matopt.opt.mat_modeling.MatOptModel` method), 204
`predict_output()` (`idaes.surrogate.pysmo.kriging.KrigingModel` method), 188
`predict_output()` (`idaes.surrogate.pysmo.polynomial_regression.PolynomialRegression` method), 178
`predict_output()` (`idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunction` method), 184
`PressureChanger` (class in `idaes.generic_models.unit_models.pressure_changer`), 551
`PressureChangerData` (class in `idaes.generic_models.unit_models.pressure_changer`), 554
`print_models_requiring_property()` (`idaes.generic_models.properties.interrogator.properties_interrogator` method), 482
`print_models_requiring_property()` (`idaes.generic_models.properties.interrogator.reactions_interrogator` method), 484
`print_properties_required_by_model()` (`idaes.generic_models.properties.interrogator.properties_interrogator` method), 482
`print_properties_required_by_model()` (`idaes.generic_models.properties.interrogator.reactions_interrogator` method), 485

[print_required_properties\(\)](#) ([idaes.generic_models.properties.interrogator.properties_interrogator](#) [method](#)), 483
[print_required_properties\(\)](#) ([idaes.generic_models.properties.interrogator.reactions_interrogator](#) [method](#)), 485
[ProcessBlock](#) (class in [idaes.core.process_block](#)), 298
[ProcessBlockData](#) (class in [idaes.core.process_base](#)), 299
[Product](#) (class in [idaes.generic_models.unit_models.product](#)), 556
[ProductData](#) (class in [idaes.generic_models.unit_models.product](#)), 557
[propagate_indexed_component_scaling_factors\(\)](#) (in module [idaes.core.util.scaling](#)), 411
[propagate_state\(\)](#) (in module [idaes.core.util.initialization](#)), 372
[PropertyInterrogatorBlock](#) (class in [idaes.generic_models.properties.interrogator.properties_interrogator](#)), 481
[PropertyInterrogatorData](#) (class in [idaes.generic_models.properties.interrogator.properties_interrogator](#)), 481
[Proportional-Integral-Derivative \(PID\) Controller](#) ([idaes.power_generation.control.pid_controller](#)), 742
[idaes.unit_models.heat_exchanger](#), 587
[Pyomo](#), 14
[Python programming environments](#), 14
R
[r2_calculation\(\)](#) ([idaes.surrogate.pysmo.kriging.KrigingModel](#) [static method](#)), 188
[r2_calculation\(\)](#) ([idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunctions](#) [static method](#)), 184
[RadialBasisFunctions](#) (class in [idaes.surrogate.pysmo.radial_basis_function](#)), 181
[ReactionBlockBase](#) (class in [idaes.core.reaction_base](#)), 352
[ReactionBlockDataBase](#) (class in [idaes.core.reaction_base](#)), 351
[ReactionInterrogatorBlock](#) (class in [idaes.generic_models.properties.interrogator.reactions_interrogator](#)), 483
[ReactionInterrogatorData](#) (class in [idaes.generic_models.properties.interrogator.reactions_interrogator](#)), 483
[ReactionParameterBlock](#) (class in [idaes.core.reaction_base](#)), 349
[release_state\(\)](#) ([idaes.core.control_volume0d.ControlVolume0DBlockData](#) [method](#)), 329
[release_state\(\)](#) ([idaes.core.control_volume1d.ControlVolume1DBlockData](#) [method](#)), 329
[release_state\(\)](#) ([idaes.generic_models.unit_models.mixer.MixerData](#) [method](#)), 543
[release_state\(\)](#) ([idaes.generic_models.unit_models.separator.Separator](#) [method](#)), 567
[report\(\)](#) ([idaes.core.control_volume1d.ControlVolume1DBlockData](#) [method](#)), 329
[report\(\)](#) ([idaes.core.property_base.StateBlock](#) [method](#)), 345
[report_statistics\(\)](#) (in module [idaes.core.util.model_statistics](#)), 391
[results_plot\(\)](#) ([idaes.gas_solid_contactors.unit_models.bubbling_flow_model](#) [method](#)), 764
[results_plot\(\)](#) ([idaes.gas_solid_contactors.unit_models.moving_bed_model](#) [method](#)), 775
[revert_state_vars\(\)](#) (in module [idaes.core.util.initialization](#)), 372
[residuals](#) ([idaes.surrogate.matopt.opt.mat_modeling.MaterialDescriptor](#) [attribute](#)), 203
[sample_points\(\)](#) ([idaes.surrogate.pysmo.sampling.CVTSampling](#) [method](#)), 198
[sample_points\(\)](#) ([idaes.surrogate.pysmo.sampling.HaltonSampling](#) [method](#)), 194
[sample_points\(\)](#) ([idaes.surrogate.pysmo.sampling.HammersleySampling](#) [method](#)), 196
[sample_points\(\)](#) ([idaes.surrogate.pysmo.sampling.LatinHypercubeSampling](#) [method](#)), 191
[sample_points\(\)](#) ([idaes.surrogate.pysmo.sampling.UniformSampling](#) [method](#)), 192
[Separator](#) (class in [idaes.generic_models.unit_models.separator](#)), 561
[set_additional_terms\(\)](#) ([idaes.surrogate.pysmo.polynomial_regression.PolynomialRegression](#) [method](#)), 178
[set_default_scaling\(\)](#) ([idaes.core.property_base.PhysicalParameterBlock](#) [method](#)), 341
[set_read_callback\(\)](#) ([idaes.core.util.model_serializer.StoreSpec](#) [method](#)), 384
[set_scaling_factor\(\)](#) (in module [idaes.core.util.scaling](#)), 409

set_write_callback() (*idaes.core.util.model_serializer.StoreSpec* method), 384
 SimpleVarLikeExpression (class in *idaes.core.util.misc*), 439
 solve_indexed_blocks() (in module *idaes.core.util.initialization*), 373
 StateBlock (class in *idaes.core.property_base*), 345
 StateBlockData (class in *idaes.core.property_base*), 343
 StateJunction (class in *idaes.generic_models.unit_models.statejunction*), 568
 StateJunctionData (class in *idaes.generic_models.unit_models.statejunction*), 569
 SteamHeater (*idaes.power_generation.unit_models.steamheater*), 667
 StoichiometricReactor (class in *idaes.generic_models.unit_models.stoichiometric_reactor*), 571
 StoichiometricReactorData (class in *idaes.generic_models.unit_models.stoichiometric_reactor*), 574
 StoreSpec (class in *idaes.core.util.model_serializer*), 381
 stream_states_dict() (in module *idaes.core.util.tables*), 417
 stream_table() (*idaes.core.flowsheet_model.FlowsheetBlockData* method), 303
 stream_table_dataframe_to_string() (in module *idaes.core.util.tables*), 418
 SWCO2ParameterBlock (class in *idaes.generic_models.properties.swco2*), 468
 SWCO2ParameterBlockData (class in *idaes.generic_models.properties.swco2*), 470
 SWCO2StateBlock (*idaes.generic_models.properties.swco2*), 465
 SWCO2StateBlock (class in *idaes.generic_models.properties.swco2*), 467
 SWCO2StateBlockData (class in *idaes.generic_models.properties.swco2*), 468
T
 tag_state_quantities() (in module *idaes.core.util.tables*), 418
 throttle_cv_fix() (*idaes.power_generation.unit_models.helm.turbine_multistage.HelmTurbineMultistageData* method), 634
 to_json() (in module *idaes.core.util.model_serializer*), 379
 total_blocks_set() (in module *idaes.core.util.model_statistics*), 403
 total_constraints_set() (in module *idaes.core.util.model_statistics*), 403
 total_equalities_generator() (in module *idaes.core.util.model_statistics*), 403
 total_equalities_set() (in module *idaes.core.util.model_statistics*), 404
 total_inequalities_generator() (in module *idaes.core.util.model_statistics*), 404
 total_inequalities_set() (in module *idaes.core.util.model_statistics*), 404
 total_objectives_generator() (in module *idaes.core.util.model_statistics*), 404
 total_objectives_set() (in module *idaes.core.util.model_statistics*), 404
 training() (*idaes.surrogate.pysmo.kriging.KrigingModel* method), 189
 training() (*idaes.surrogate.pysmo.polynomial_regression.PolynomialRegressionModel* method), 178
 training() (*idaes.surrogate.pysmo.radial_basis_function.RadialBasisFunctionModel* method), 184
 Translator (class in *idaes.generic_models.unit_models.translator*), 575
 TranslatorData (class in *idaes.generic_models.unit_models.translator*), 576
 turbine_inlet_cf_fix() (*idaes.power_generation.unit_models.helm.turbine_multistage.HelmTurbineMultistageData* method), 634
 turbine_outlet_cf_fix() (*idaes.power_generation.unit_models.helm.turbine_multistage.HelmTurbineMultistageData* method), 634
 Txy_data() (in module *idaes.core.util.phase_equilibria*), 373
 Txy_diagram() (in module *idaes.core.util.phase_equilibria*), 375
 TXYDataClass() (in module *idaes.core.util.phase_equilibria*), 374
U
 unfix_initial_conditions() (*idaes.core.process_base.ProcessBlockData* method), 300
 unfixed_variables_generator() (in module *idaes.core.util.model_statistics*), 404
 unfixed_variables_in_activated_equalities_set() (in module *idaes.core.util.model_statistics*), 405

`unfixed_variables_set()` (in module `idaes.core.util.model_statistics`), 405
`UniformSampling` (class in module `idaes.surrogate.pysmo.sampling`), 192
`UnitModelBlock` (class in `idaes.core.unit_model`), 357
`UnitModelBlockData` (class in `idaes.core.unit_model`), 354
`unscaled_constraints_generator()` (in module `idaes.core.util.scaling`), 414
`unscaled_variables_generator()` (in module `idaes.core.util.scaling`), 413
`unset_default_scaling()` (`idaes.core.property_base.PhysicalParameterBlock` method), 341
`unset_scaling_factor()` (in module `idaes.core.util.scaling`), 409
`unused_variables_set()` (in module `idaes.core.util.model_statistics`), 405
`use_equal_pressure_constraint()` (`idaes.generic_models.unit_models.mixer.MixerData` method), 543
`use_minimum_inlet_pressure_constraint()` (`idaes.generic_models.unit_models.mixer.MixerData` method), 543
`variables_near_bounds_generator()` (in module `idaes.core.util.model_statistics`), 406
`variables_near_bounds_set()` (in module `idaes.core.util.model_statistics`), 406
`variables_only_in_inequalities()` (in module `idaes.core.util.model_statistics`), 407
`variables_set()` (in module `idaes.core.util.model_statistics`), 407
`VarLikeExpression` (class in `idaes.core.util.misc`), 438
`visualize()` (`idaes.core.flowsheet_model.FlowsheetBlockData` method), 304
`visualize()` (in module `idaes.ui.fsvis.fsvis`), 148

W

`WaterPipe`
`idaes.power_generation.unit_models.waterpipe`, 703
`WaterwallSection`
`idaes.power_generation.unit_models.waterwall`, 653

V

`value()` (`idaes.core.util.misc._GeneralVarLikeExpressionData` property), 439
`value()` (`idaes.core.util.model_serializer.StoreSpec` class method), 384
`value_isfixed()` (`idaes.core.util.model_serializer.StoreSpec` class method), 384
`value_isfixed_isactive()` (`idaes.core.util.model_serializer.StoreSpec` class method), 384
`Valve`
`idaes.generic_models.unit_models.valve`, 578
`Valve` (class in `idaes.generic_models.unit_models.valve`), 583
`ValveData` (class in `idaes.generic_models.unit_models.valve`), 586
`variables_in_activated_constraints_set()` (in module `idaes.core.util.model_statistics`), 405
`variables_in_activated_equalities_set()` (in module `idaes.core.util.model_statistics`), 405
`variables_in_activated_inequalities_set()` (in module `idaes.core.util.model_statistics`), 405