IDAES Documentation

Release 1.4.4

IDAES team

Feb 08, 2020

Contents

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 CO_2 capture technologies, as well as integration with other new technologies such as supercritical CO_2. In addition, the tools and capabilities will be applicable to renewable energy development, such as biofuels, green chemistry, Nuclear and Environmental Management, such as the design of complex, integrated waste treatment facilities.

CHAPTER 2

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)

Contact, contributions and more information

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>

Contents

4.1 Installation

4.1.1 Installation using Docker

One way to install the IDAES PSE Framework is by using the pre-built Docker image.

A Docker image is essentially an embedded instance of Linux (even if you are using Windows or Mac OSX) that has all the code for the IDAES PSE framework pre-installed. You can run commands and Jupyter Notebooks in that image. This section describes how to set up your system, get the Docker image, and interact with it.

Install Docker on your system

1. Install the community edition (CE) of Docker (website: https://docker.io).

2. Start the Docker daemon. How to do this will depend on your operating system.

   OS X You should install Docker Desktop for Mac. Docker should have been installed to your Applications directory. Browse to it and click on it from there. You will see a small icon in your toolbar that indicates that the daemon is running.

   Linux Install Docker using the package manager for your OS. Then start the daemon. If you are using Ubuntu or a Debian-based Linux distro, the Docker daemon will start automatically once Docker is installed. For CentOS, start Docker manually, e.g., run `sudo systemctl start docker`.

   Windows You should install Docker Desktop for Windows. Docker will be started automatically.

Get the IDAES Docker image

You need to get the ready made Docker image containing the source code and solvers for the IDAES PSE framework. This image is available for download from DockerHub (an online portal where Docker images are stored). Images on

DockerHub are versioned according to the release version. See the **Releases_** page on GitHub for information about what is different about each version.

If you want the latest version, simply use the tag "latest" as the version number. Thus, **running the following in a terminal will download the latest version**:

```
docker pull idaes/jupyterhub:latest
```

-.. _Releases: https://github.com/IDAES/idaes-pse/releases

Run the IDAES Docker image

To start the Docker image, use a graphical user interface or a console or shell command-line interface.

From the command-line, if you want to start up the Jupyter Notebook server, e.g. to view and run the examples and tutorials, then run this command:

```
$ docker run -p 8888:8888 -it idaes/jupyterhub
... <debugging output from Jupyter>
...
Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://(305491ce063a or 127.0.0.1):8888/?
→token=812a290619211bef9177b0e8c0fd7e4d1f673d29909ac254
```

Copy and paste the URL provided at the end of the output into a browser window and you should get a working Jupyter Notebook. You can browse to the examples directory under /home/idaes/examples and click on the Jupyter Notebooks to open them.

To interact with the image directly from the command-line (console), you can run the following command:

```
$ docker run -p 8888:8888 -it idaes/jupyterhub /bin/bash
jovyan@10c11ca29008:~$ cd /home/idaes
...
```

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.

The Docker installation works on any platform that supports Docker, but of course requires installation of, and some understanding of, Docker itself to operate.

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

System	Section
Linux	*Linux*
Windows	*Windows*
Mac OSX	*Mac/OSX*
Generic	*Generic install*
Docker-based	*Installation using Docker*

4.1.2 Windows

Install Miniconda

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 powershell (Start -> "Anaconda Powershell Prompt").

4. In the Anaconda Powershell, follow the *Generic install* instructions.

4.1.3 Linux

Install Miniconda

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).

4. Follow the *Generic install* instructions.

5. The IPOPT solver depends on the GNU FORTRAN libraries, which are not bundled with it. Unless you know that these are already installed on your system, you should manually install them using "apt-get", "yum" or other appropriate package manager. Depending on how your system is set up you either need to use the "sudo" command or install packages as the "root" user.

apt-get (Debian or Ubuntu based distributions):

```
apt-get install libgfortran3
```

yum (RedHat based distributions):

```
yum install libgfortran
```

4.1.4 Mac/OSX

Install Miniconda

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).

4. Follow the *Generic install* instructions.

4.1.5 Generic install

Once you have Conda installed, the remaining steps, performed in either the Anaconda Powershell (Prompt) or a Linux terminal, are the same.

If you are familiar with Python/Conda environments, you will probably want to create a new environment for your IDAES installation before starting to install Python and/or Conda packages, *e.g.*, `conda create -n <env> python=3.7` then `conda activate <env>`. If you are not familiar with these commands, don't worry, this is an optional step.

Install IDAES

1. Install IDAES with pip:

```
pip install idaes-pse
```

2. Run the idaes command to install the compiled binaries:

```
idaes get-extensions
```

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

3. Run tests:

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

4. 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 failed and succeed test 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.)

4.2 IDAES Modeling Standards

Contents

- *IDAES Modeling Standards*
  - *Model Formatting and General Standards*
    - *Headers and Meta-data*
    - *Coding Standard*
    - *Model Organization*
    - *Commenting*
  - *Units of Measurement and Reference States*
  - *Standard Variable Names*
    - *Standard Naming Format*
    - *Constants*
    - *Thermophysical and Transport Properties*
    - *Reaction Properties*
    - *Solid Properties*
    - *Naming Examples*

4.2.1 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 provides a brief description of the purpose of the model and how it was developed. Some suggested information to include is:

- Model name,
- Model publication date,
- Model author
- Any necessary licensing and disclaimer information (see below).
- Any additional information the modeler feels should be included.

Coding Standard

All code developed as part of IDAES should conform to the PEP-8 standard.

Model Organization

Whilst the overall IDAES modeling framework enforces a hierarchical structure on models, model developers are still encouraged 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 property packages, it is recommended that all the equations necessary for calculating a given property be grouped together, clearly separated and identified by using comments.

Additionally, model developers are encouraged to consider breaking their model up into a number of smaller methods where this makes sense. This can facilitate modification of the code by allowing future users to inherit from the base model and selectively overload sub-methods where desired.

Commenting

To help other modelers and users understand the how a model works, model builders are strongly encouraged to comment their code. It is suggested 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.

4.2.2 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 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 consider to be part of the liquid phase, as they will be handled via pumps rather than compressors.

4.2.3 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 the IDAES modeling framework, this occurs most frequently with information regarding the state and properties of the material within the system, which is calculated in specialized property blocks, and then used in others parts of the model. This section of the documentation discusses the standard naming conventions used within the IDAES modeling framework.

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

Constant	Standard Name
Gas Constant	gas_const

Thermophysical and Transport Properties

Below is a list of all the thermophysical properties which currently have a standard name associated with them in the IDAES framework.

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

Continued on next page

Table 1 – continued from previous page

Variable	Standard Name
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
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 currently have a standard name associated with them in the IDAES framework.

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 currently have a standard name associated with them in the IDAES framework.

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

4.3 Core Library

4.3.1 Core Contents

IDAES Framework Configuration

The IDAES framework can be configured with configuration files in TOML format. Supplying a configuration file is optional. Currently this file sets logging configuration and modules that should be searched for plugins. The configuration is done when first importing any idaes.* module. The IDAES framework will first attempt to read a user-level configuration file at `%LOCALAPPDATA%\idaes\idaes.conf` on Windows or `$HOME/.idaes/idaes.conf` on other operating systems (e.g. Linux or Mac). Next if an idaes.conf file exists in the working directory it will be read. Configuration files in the working directory will override settings in the user-level configuration file. The user level configuration file will override default settings. Not all setting need to be set in a configuration file.

An example configuration file is given below with the default settings.

```
[plugins]
  required = []
  optional = []
[logging]
  version = 1
  disable_existing_loggers = false
  [logging.formatters.f1]
    format = "%(asctime)s - %(levelname)s - %(name)s - %(message)s"
    datefmt = "%Y-%m-%d %H:%M:%S"
  [logging.handlers.console]
    class = "logging.StreamHandler"
```

(continues on next page)

```
    formatter = "f1"
    stream = "ext://sys.stderr"
[logging.loggers.idaes]
    level = "INFO"
    handlers = ["console"]
```

The Python dictConfig method is used to set up the logger. The required and optional elements under plugins are string lists of modules to search for Pyomo style plugins. Any failure to import plugins in the required modules will raise an exception, while any failure to import optional plugins will only result in the exception being logged and execution continuing.

Process Blocks

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.

```python
from pyomo.environ import *
from pyomo.common.config import ConfigValue
from idaes.core import ProcessBlockData, declare_process_block_class


@declare_process_block_class("MyBlock")
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.

```python
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.

```python
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.

```python
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 requirment 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*, ***kwargs*)

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 Block-Data 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 overide 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 sepecified 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 implimented 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

unfix_initial_conditions()
    This method unfixed the initial conditions for dynamic models.

        Parameters **None** –

    Returns : None

IDAES Modeling Concepts

Contents

- *IDAES Modeling Concepts*
    - *Introduction*
    - *Time Domain*
    - *Flowsheets*
    - *Unit Models*
    - *Component References*
    - *What Belongs in Each Type of Block?*

Introduction

The purpose of this section of the documentation is to explain the different parts of the IDAES modeling framework, and what components belong in each part for the hierarchy. Each component is described in greater detail later in the documentation, however this section provides a general introduction to different types of components.

Time Domain

Before starting on the different types of models present in the IDAES framework, it is important to discuss how time is handled by the 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 documentation). 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 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 PropertyBlock 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, however this is generally a single point at time = 0.0. However, models still contain a reference to the time domain, and any components are still indexed by time even in a steady-state model (e.g. PropertyBlocks).

Flowsheets

The top level of the IDAES modeling framework is the Flowsheet model. Flowsheet models represent traditional process flowsheets, containing a number of Unit models representing process unit operations connected together into a flow network. Flowsheets generally contain three types of components:

1. Unit models, representing unit operations,

2. Arcs, representing connections between Unit models, and,

3. Property Parameter blocks, representing the parameters associated with different materials present within the flowsheet.

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.

Unit Models

Unit models generally represent individual pieces of equipment present within a process which perform a specific task. Unit models in turn are generally composed of two main types of components:

1. Control Volume Blocks, which represent volume of material over which we wish to perform material, energy and/or momentum balances, and,

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 Unit models to connect to other Unit models.

Unit models will also contain Constraints describing the performance of the unit, which will relate terms in the balance equations to different phenomena.

Control Volumes

A key feature of the IDAES modeling framework is the use of Control Volume Blocks. As mentioned above, Control Volumes represent a volume of material over which material, energy and/or momentum balances can be performed. Control Volume Blocks contain methods to automate the task of writing common forms of these balance equations. Control Volume Blocks can also automate the creation of StateBlocks and ReactionBlocks associated with the control volume.

Property Blocks

Property blocks represent the state of a material at a given point in space and time within the process flowsheet, and contain the state variables, thermophysical, transport and reaction properties of a material (which are functions solely of the local state of the material). Within the IDAES process modeling framework, properties are divided into two types:

- Physical properties (StateBlocks), including thermophysical and transport properties, and

- Reaction properties (ReactionBlocks), which include all properties associated with chemical reactions.

Additionally, StateBlocks contain information on the extensive flow of material at that point in space and time, which is a departure from how engineers generally think about properties. This is required to facilitate the flexible formulation

of the IDAES Framework by allowing the property package to dictate what form the balance equations will take, which requires the StateBlock to know the extensive flow information.

The calculations involved in property blocks of both types generally require a set of parameters which are constant across all instances of that type of property block. Rather than each property block containing its own copy of each of these parameters (thus duplicating parameters between blocks), each type of property block is associated with a Property Parameter Block (PhysicalParameterBlock or ReactionParameterBlock). Property Parameter Blocks serve as a centralized location for the constant parameters involved in property calculations, and all property blocks of the associated type link to the parameters contained in the parameter block.

Component References

There are many situations in the IDAES modeling framework where a developer may want to make use of a modeling component (e.g. a variable or parameter) from one Block in another Block. The time domain is a good example of this - almost all Blocks within an IDAES model need to make use of the time domain, however the time domain exists only at the top level of the flowsheet structure. In order to make use of the time domain in other parts of the framework, references to the time domain are used instead. By convention, all references within the IDAES modeling framework are indicated by the suffix "_ref" attached to the name of the reference. E.g. all references to the time domain within the framework are called "time_ref".

What Belongs in Each Type of Block?

A common question with the hierarchical structure of the IDAES framework is where does a specific variable or constraint belong (or conversely, where can I find a specific variable or constraint). In general, variables and constraints are divided based on the following guidelines:

1. Property Parameter Blocks - any parameter or quantity that is consistent across all instances of a Property Block belongs in the Property Parameter Block. This includes:

   - component lists,

   - lists of valid phases,

   - universal constants (e.g. R, π),

   - constants used in calculating properties (e.g. coefficients for calculating c_p,

   - reference states (e.g. P_{ref} and T_{ref}),

   - lists of reaction identifiers,

   - reaction stoichiometry.

2. Property Blocks - all state variables (including extensive flow information) and any quantity that is a function only of state variables plus the constraints required to calculate these. These include:

   - flow rates (can be of different forms, e.g. mass or molar flow, on a total or component basis),

   - temperature,

   - pressure,

   - intensive and extensive state functions (e.g. enthalpy); both variables and constraints.

3. Control Volume Blocks - material, energy and momentum balances and the associated terms. These include:

   - balance equations,

   - holdup volume,

   - material and energy holdups; both variables and constraints,

- material and energy accumulation terms (Pyomo.dae handles the creation of the associated derivative constraints),

- material generation terms (kinetic reactions, chemical and phase equilibrium, mass transfer),

- extent of reaction terms and constraints relating these to the equivalent generation terms,

- phase fraction within the holdup volume and constrain on the sum of phase fractions,

- heat and work transfer terms,

- pressure change term

- diffusion and conduction terms (where applicable) and associated constraints,

- Mixer and Splitter blocks for handling multiple inlets/outlets.

4. Unit Model - any unit performance constraints and associated variables, 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).

5. Flowsheet Model - any constraints related to interaction of unit models and associated variables. Examples include:

   - control constraints relating behavior between different units (e.g. a constraint on valve opening based on the level in another unit).

Flowsheet Model Class

Contents

- *Flowsheet Model Class*
    - *Default Property Packages*
    - *Flowsheet Configuration Arguments*
    - *Flowsheet Classes*

Flowsheet models make up the top level of the IDAES modeling framework, and represent the flow of material and energy through a process. Flowsheets will generally contain a number of UnitModels to represent unit operations within the process, and will contain one or more Property Packages which represent the thermophysical and transport properties of material within the process.

Flowsheet models are responsible for establishing and maintaining the time domain of the model, including declaring whether the process model will be dynamic or steady-state. This time domain is passed on to all models attached to the flowsheet (such as Unit Models and sub-Flowsheets). The Flowsheet model also serves as a centralized location for organizing property packages, and can set one property package to use as a default throughout the flowsheet.

Flowsheet Blocks may contain other Flowsheet Blocks in order to create nested flowsheets and to better organize large, complex process configurations. In these cases, the top-level Flowsheet Block creates the time domain, and each sub-flowsheet creates a reference this time domain. Sub-flowsheets may make use of any property package declared at a higher level, or declare new property package for use within itself - any of these may be set as the default property package for a sub-Flowsheet.

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 ContunuousSet. 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_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 signfiicant 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
>
> **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

serialize(*file_base_name*, *overwrite=False*)
> Serializes the flowsheet and saves it to a file that can be read by the idaes-model-vis jupyter lab extension.
>
> > **Parameters** `file_base_name` – The file prefix to the .idaes.vis file produced.
>
> The file is created/saved in the directory that you ran from Jupyter Lab. :param overwrite: Boolean to overwrite an existing file_base_name.idaes.vis. If True, the existing file with the same file_base_name will be overwritten. This will cause you to lose any saved layout. If False and there is an existing file with that file_base_name, you will get an error message stating that you cannot save a file to the file_base_name (and therefore overwriting the saved layout). If there is not an existing file with that file_base_name then it saves as normal. Defaults to False. :return: 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

class idaes.core.flowsheet_model.**FlowsheetBlock**(**args*, ***kwargs*)
> 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].

> **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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (FlowsheetBlock) New instance

Property Packages

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.

- 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 PropertyParameterBlocks. Inheriting models should call super().build.

　　　　　　Parameters None –

　　　　　　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).

StateBlockData Class

StateBlockData contains the code necessary for implementing the as needed construction of variables and constraints.

class `idaes.core.property_base.`**`StateBlockData`**(*component*)

    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 StateBlockDatas.

            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.

StateBlock Class

class idaes.core.property_base.**StateBlock**(**args*, ***kwargs*)
> 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 StateBlock-Data 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*
>   - *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.

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.

- 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`**(*component*)

> 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

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 bases 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).

ReactionBlockDataBase Class

ReactionBlockDataBase contains the code necessary for implementing the as needed construction of variables and constraints.

class `idaes.core.reaction_base.`**`ReactionBlockDataBase`**(*component*)

> This is the base class for reaction block data objects. These are blocks that contain the Pyomo components associated with calculating a set of reacion properties for a given material.

> **build**()
>
> > General build method for PropertyBlockDatas. 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.

ReactionBlockBase Class

class idaes.core.reaction_base.**ReactionBlockBase**(**args*, ***kwargs*)

>   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

IDAES Property Packages

The IDAES process modeling framework divides property calculations into two parts;

  - physical and transport properties

  - chemical reaction properties

Defining the calculations to be used when calculating properties is done via "property packages", which contain a set of related calculations for a number of properties of interest. Property packages may be general in purpose, such as ideal gas equations, or specific to a certain application.

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, the IDAES modeling framework 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 to the framework 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.

Unit Model Class

The UnitModelBlock is 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_outlets 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 to use for Port object`** (`name`) –
>
> - **`= an instance of a ControlVolume or StateBlock to use as the`** (`block`) – 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 string for Port object`** (`doc`) –
>
> **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 to use for Port object`** (`name`) –
>
> - **`= an instance of a ControlVolume or StateBlock to use as the`** (`block`) – 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 string for Port object`** (`doc`) –
>
> **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 name = name to use for Port object.: :keyword block = an instance of a StateBlock to use as the source to: populate the Port object :keyword doc = doc string for Port object:

> **Returns** A Pyomo Port object and associated components.

`build`()

General build method for UnitModelBlockData. 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=6*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)

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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

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*, ***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 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:** { **True** - construct holdup terms, **False** - do not construct holdup terms}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (UnitModelBlock) New instance

Control Volume Classes

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*, ***kwargs*)

> 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 encompases 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 ProcessBlockData 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 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.}
>
>   **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 construciton.}

- **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 overide 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 momentum 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:*   *1>*,
    has_phase_equilibrium=None)
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*, *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

- **– 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=6*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)
Initialization routine for 0D 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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>
> - **optarg** – solver options dictionary object (default=None)
>
> - **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')
>
> - **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=6*)
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):

$$\sum_p \frac{\partial M_{t,p,j}}{\partial t} = \sum_p F_{in,t,p,j} - \sum_p F_{out,t,p,j} + \sum_p N_{kinetic,t,p,j} + \sum_p N_{equilibrium,t,p,j} + \sum_p N_{pe,t,p,j} + \sum_p N_{transfer,t,p,j} + N_{custo}$$

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
enthalpy_holdup	$E_{t,p}$	t, p	has_holdup = True
phase_fraction	$\phi_{t,p}$	t, p	has_holdup = True
enthalpy_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

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):

$$s_{energy} \times \sum_p \frac{\partial E_{t,p}}{\partial t} = s_{energy} \times \sum_p H_{in,t,p} - s_{energy} \times \sum_p H_{out,t,p} + s_{energy} \times Q_t + s_{energy} \times W_t + s_{energy} \times Q_{rxn,t} + s_{energy}$$

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)*:

$$E_{t,p} = h_{t,p} \times V_t \times \phi_{t,p}$$

where $h_{t,p}$ is the enthalpy density (specific enthalpy) 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*, ***kwargs*)

    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 encompases 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 ProcessBlockData 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 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.}

    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 construciton.}

    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 domian, 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 transformating 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 overide the default behavior 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.

> - **–** – a new ContinuousSet if length_domain is not provided (default = [0.0, 1.0]).

> - **– 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*, *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

- **– 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=6*, *optarg=None*, *solver='ipopt'*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default=None)

- **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')

- **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=6*)
> 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. $mol/m \cdot s$).

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):

$$L \times \frac{\partial M_{t,x,p,j}}{\partial t} = fd \times \frac{\partial F_{t,x,p,j}}{\partial x} + L \times N_{kinetic,t,x,p,j} + L \times N_{equilibrium,t,x,p,j} + L \times N_{pe,t,x,p,j} + L \times N_{transfer,t,x,p,j} + L \times N_c$$

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):

$$L \times \sum_p \frac{\partial M_{t,x,p,j}}{\partial t} = fd \times \sum \frac{\partial F_{t,x,p,j}}{\partial x} + L \times \sum_p N_{kinetic,t,x,p,j} + L \times \sum_p N_{equilibrium,t,x,p,j} + L \times \sum_p N_{transfer,t,x,p,j} + L \times \ldots$$

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 PhysicalPa-rameterBlock).

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 PhysicalPa-rameterBlock).

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
enthalpy_holdup	$E_{t,x,p}$	t, x, p	has_holdup = True
phase_fraction	$\phi_{t,x,p}$	t, x, p	has_holdup = True
enthalpy_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

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):

$$s_{energy} \times L \times \sum_p \frac{\partial E_{t,x,p}}{\partial t} = s_{energy} \times fd \, times \sum_p \frac{\partial H_{t,x,p}}{\partial x} + s_{energy} \times L \times Q_{t,x} + s_{energy} \times L \times W_{t,x} + s_{energy} \times L \times Q_{rxn,t}$$

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)*:

$$E_{t,x,p} = h_{t,x,p} \times A \times \phi_{t,x,p}$$

where $h_{t,x,p}$ is the enthalpy density (specific enthalpy) 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.

Control Volumes are the center of the IDAES process modeling framework, and 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 Control Volume 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 Control Volume 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 Control Volume:

- *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 Control Volume classes are built on a common core (*ControlVolumeBlockData*) which defines a set of common tasks required for all Control Volumes. The more specific Control Volume classes then build upon these common tasks to provide tools appropriate for their specific application.

All Control Volume 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`.

More details on these steps is *provided later*.

Setting up the time domain

The first common task the Control Volume block performs is to determine if it should be dynamic or steady-state and to collect the time domain from the UnitModel. Control Volume blocks 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 Control Volume block will inherit this argument from its parent `UnitModel`.

Finally, the Control Volume 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 `UnitModel` has a `property_package` argument set, and uses this if present. Otherwise, the Control Volume block 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 Control Volumes 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 Control Volume 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 Control Volumes is to automate as much of the task of writing a unit model as possible. For this purpose, Control Volumes 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 Control Volumes, 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.

class idaes.core.control_volume_base.**ControlVolume**(**args*, ***kwargs*)
    This class is not usually used directly. Use ControlVolume0DBlock or ControlVolume1DBlock instead.

        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, **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 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.}

                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 construciton.}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (ControlVolume) New instance

class idaes.core.control_volume_base.**ControlVolumeBlockData**(*component*)

The ControlVolumeBlockData Class forms the base class for all IDAES ControlVolume models. The purpose of this class is to automate the tasks common to all control volume blockss and ensure that the necessary attributes of a control volume block are present.

The most signfiicant role of the ControlVolumeBlockData class is to set up the construction arguments for the control volume block, automatically link to the time domain of the parent block, and to get the information about the property and reaction packages.

add_energy_balances(*balance_type=<EnergyBalanceType.useDefault: -1>*, ***kwargs*)

General method for adding energy balances to a control volume. This method makes calls to specialised sub-methods for each type of energy balance.

Parameters

- **balance_type** (`EnergyBalanceType`) – Enum indicating which type of energy balance should be constructed.

- **has_heat_of_reaction** (`bool`) – whether terms for heat of reaction should be included in energy balance

- **has_heat_transfer** (`bool`) – whether generic heat transfer terms should be included in energy balances

- **has_work_transfer** (`bool`) – whether generic mass transfer terms should be included in energy balances

- **custom_term** (`Expression`) – a Pyomo Expression representing custom terms to be included in energy balances

Returns Constraint objects constructed by sub-method

add_geometry(**args*, ***kwargs*)

Method for defining the geometry of the control volume.

See specific control volume documentation for details.

add_material_balances(*balance_type=<MaterialBalanceType.useDefault: -1>*, ***kwargs*)

General method for adding material balances to a control volume. This method makes calls to specialised sub-methods for each type of material balance.

Parameters

- – **MaterialBalanceType Enum indicating which type of** (`balance_type`) – material balance should be constructed.

- – **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.

- **– a Pyomo Expression representing custom terms to** – be included in
  material balances on a mass basis.

> **Returns** Constraint objects constructed by sub-method

add_momentum_balances(*balance_type=<MomentumBalanceType.pressureTotal: 1>*, ***kwargs*)
General method for adding momentum balances to a control volume. This method makes calls to specialised sub-methods for each type of momentum balance.

> **Parameters**
>
> - **balance_type** (*MomentumBalanceType*) – Enum indicating which type of momentum balance should be constructed. Default = MomentumBalanceType.pressureTotal.
>
> - **has_pressure_change** (*bool*) – whether default generation terms for pressure change should be included in momentum balances
>
> - **custom_term** (*Expression*) – a Pyomo Expression representing custom terms to be included in momentum balances
>
> **Returns** Constraint objects constructed by sub-method

add_phase_component_balances(**args*, ***kwargs*)
Method for adding material balances indexed by phase and component to the control volume.

See specific control volume documentation for details.

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(**args*, ***kwargs*)
Method for adding ReactionBlocks to the control volume.

See specific control volume documentation for details.

add_state_blocks(**args*, ***kwargs*)
>   Method for adding StateBlocks to the control volume.

>   See specific control volume documentation for details.

add_total_component_balances(**args*, ***kwargs*)
>   Method for adding material balances indexed by component to the control volume.

>   See specific control volume documentation for details.

add_total_element_balances(**args*, ***kwargs*)
>   Method for adding total elemental material balances indexed to the control volume.

>   See specific control volume documentation for details.

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(**args*, ***kwargs*)
>   Method for adding a total enthalpy balance to the control volume.

>   See specific control volume documentation for details.

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(**args*, ***kwargs*)
>   Method for adding a total pressure balance to the control volume.

>   See specific control volume documentation for details.

build()
>   General build method for Control Volumes blocks. This method calls a number of sub-methods which automate the construction of expected attributes of all ControlVolume blocks.

>   Inheriting models should call *super().build*.

>>   **Parameters None** –

>>   **Returns** None

Auto-Construct Method

To reduce the demands on unit model developers even further, Control Volumes have an optional auto-construct feature that will attempt to populate the Control Volume based on a set of instructions provided at the Unit Model level. If the `auto_construct` configuration argument is set to *True*, the following methods are called automatically in the following order when instantiating the Control Volume.

1. *add_geometry*

2. *add_state_blocks*

3. *add_reaction_blocks*

4. *add_material_balances*

5. *add_energy_balances*

6. *add_momentum_balances*

7. *apply_transformation*

To determine what terms are required for the balance equations, the Control Volume expects the Unit Model to have the following configuration arguments, which are used as arguments to the methods above.

- dynamic

- has_holdup

- material_balance_type

- energy_balance_type

- momentum_balance_type

- has_rate_reactions

- has_equilibrium_reactions

- has_phase_equilibrium

- has_mass_transfer

- has_heat_of_reaction

- has_heat_transfer

- has_work_transfer

- has_pressure_change

- property_package

- property_package_args

- reaction_package

- reaction_package_args

For convenience, a template ConfigBlock (named *CONFIG_Template*) is available in the control_volume_base.py module which contains all the necessary arguments which can be inherited by unit models wishing to use the auto-construct feature.

Utility Methods

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.`**`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 StateBlock 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*, ***kwds*)
> 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
> >
> > - **`kwds`** – a dict of argumnets to be passed to the solver
> >
> > **Returns** A Pyomo solver results object

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.

```python
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)
    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*, *return_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 possilbe
>
> - **metadata** – A dictionary of addtional metadata to add.
>
> - **wts** – is What To Save, this is a StoreSpec object that specifies what object types and attributes to save. If None, the default is used which saves the state of the compelte model state.
>
> - **metadata** – addtional 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*)

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-compoent of o. If the saved state contains extra information, it is ignored. If the save state doesn't contain an enetry 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

- **`wts`** – 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 perfomance 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   'py-*
                                                                      omo.core.base.param.Param'>,
                                                                      ('_mutable',   )),   (<class   'py-
                                                                      omo.core.base.var.Var'>,
                                                                      ()),   (<class   'py-
                                                                      omo.core.base.component.Component'>,
                                                                      ('active',))), data_classes=((<class
                                                                      'pyomo.core.base.var._VarData'>,
                                                                      ('fixed',   'stale',   'value',
                                                                      'lb',   'ub')),   (<class   'py-
                                                                      omo.core.base.param._ParamData'>,
                                                                      ('value',   )),   (<class   'int'>,
                                                                      ('value',   )),   (<class   'float'>,
                                                                      ('value',   )),   (<class   'py-
                                                                      omo.core.base.component.ComponentData'>,
                                                                      ('active',   ))),
                                                                      skip_classes=(<class   'py-
                                                                      omo.core.base.external.ExternalFunction'>,
                                                                      <class   'pyomo.core.base.sets.Set'>,
                                                                      <class   'pyomo.network.port.Port'>,
                                                                      <class   'py-
                                                                      omo.core.base.expression.Expression'>,
                                                                      <class   'py-
                                                                      omo.core.base.rangeset.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 tupple) containing
>   the following elements: (1) class (compared using isinstance) (2) attribute list or None,
>   an emptry 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 obejct
>   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 excpetion 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 regaurdless of this argument.

- **suffix** – If True store suffixes and component ids. If false, don't store suffixes.

- **suffix_filter** – None to store all siffixes 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*)
Retur 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 were the component name does not matter when reading the serialized data.

```
{
    "__metadata__": {
        "format_version": 3,
        "date": "2018-12-21",
        "time": "11:34:39.714323",
        "other": {
        },
        "__performance__": {
            "n_components": 219,
            "etime_make_dict": 0.003}
```

(continues on next page)

```
    },
    "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 thier 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": {
              "...": "..."
            }
          },
          "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":0.0009555816650390625
      }
    },
    "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":{
                "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
          }
        }
      },
      "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":{
```

Chapter 4. Contents

```
          "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
        }
      }
    }
  }
}
```

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 calcuation 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 deactived 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 differnt 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 deacitvated Block in a model.

> **Parameters block** – model to be studied

> **Returns** A ComponentSet including any Var components which belong to a deacitvated Block but appear in an activate 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 indeactivated 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*)
> 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

>> **Returns** A ComponentSet including all Constraint components with a residual greater than tol which appear in block

`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 deacitvated Block in a model.

>> **Parameters** **`block`** – model to be studied

>> **Returns** Number of Var components which belong to a deacitvated 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 activagted 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 inequality 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 Objective 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 Objective 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)
Generator which returns 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 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*)
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)

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

Table Methods

The IDAES toolset contians a number of methods for generating and dislpaying summary tables of data in the form of pandas `DataFrames`.

Available Methods

`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 StateBlocks.
>
> - **`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*)

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 srings*) – A list of strings that will be used as column headings. If None the attribute names will be used.
>
> **Returns** A Pandas dataframe containing a data table
>
> **Return type** (DataFrame)

`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

---

4.3.2 Core Overview

All components of the IDAES process modeling framework are built of Pyomo Block components (see Pyomo documentation).

The ProcessBlock class is the base class of IDAES models, and provides the common foundation for all other components.

FlowsheetModel objects represent the top level of the IDAES modeling hierarchy, and contain connected networks of unit models, or even contain other flowsheet models, which are connected by Pyomo Arcs.

Physical property packages supply information about a material's state including physical properties and flow rates. Reaction property packages are used in systems where chemical reactions may take place, and supply information on reaction rates and stoichiometry, based on a material's state.

Equipment models are derived from UnitModel. Unit models contain control volumes and have ports which can be used to connect material and energy flows between unit models. On top of the balance equations usually contained in control volumes unit models contain additional performance equations that may calculate things like heat and mass transfer or efficiency curves.

ControlVolumes are the basic building block used to construct unit models that contain material and energy holdup and flows in and out. These blocks contain energy, mass, and momentum balances, as well as state and reaction blocks associated with the material within the control volume.

More detail on the different types of modeling objects is available in the Modeling Concepts section.

4.4 Transformations

Transformations offer a convenient way to make systematic changes to a model.

4.4.1 Variable Replacement

There are a number of cases where it can be convenient to replace one variable for another. IDAES offers a convenient variable replacement transformation. This transformation is not reversible and can significantly alter the model structure.

An example use of this transformation, is a parameter estimation problem where a model contains several instances of a particular sub-model and each model contains a variable (β) for a model parameter to be estimated. In many cases β should be the same across all sub-models. One approach to this problem would be to add equality constraints to equate all the β's. Another approach would be to use the variable replacement transformation to replace the individual β's with a single global β variable.

Example

The following example demonstrates the basic usage of the transformation.

```python
import idaes # Load IDAES plugins including the replacement transformation
import pyomo.environ as pyo

# Use Pyomo's transformation factory to create the transformation object
rp = pyo.TransformationFactory("replace_variables")

# Create an example model
m = pyo.ConcreteModel()
```

```python
m.x = pyo.Var({1,2,3}, initialize=2)
m.new_x = pyo.Var({1,2,3}, initialize=3)
m.e1 = pyo.Expression(expr=sum(m.x[i] for i in m.x))

# Apply the transformation to the model, the substitute argument contains a list
# of replacements, each element is a list-like object where the first element is
# a variable to be replaced by the second element.
rp.apply_to(m, substitute=[(m.x, m.new_x)])

# See that the variable was replaced
print(pyo.value(m.e1)) # since new_x has a value of 3 the expression value is 9
```

Output:

```
9
```

Usage

There are three basic steps to using the variable replacement transformation.

1. Import anything from the `idaes` package; this will cause the IDAES plugins to be loaded.

2. Use Pyomo's transformation factory to create a variable replacement transformation object (e.g. `rp = TransformationFactory("replace_variables")`.

3. Call the transformation object's `apply_to()` method to apply the transformation.

The `apply_to(instance, substitute)` method takes two arguments `instance` and `substitute`. The instance argument is a model or block to apply the transformation to. The substitute argument is a list-like object with substitutions. Each element is a two-element list-like object where the first element is a Pyomo Var, IndexedVar element or Reference to the variable to replace and the second element is a Pyomo Var, IndexedVar element or Reference to replace the first element with.

Indexed variables are allowed. The index set of the variable to replace must be a subset of the index set of the variable to replace it with. It can also be useful to use a Pyomo Reference to emulate an indexed variable, so this is also supported.

ReplaceVariables Class

The transformation object class is ReplaceVariables.

class idaes.plugins.variable_replace.**ReplaceVariables**(***kwds*)
Replace variables in a model or block with other variables.

Keyword arguments below are specified for the `apply_to(instance, **kwargs)` method.

> **Keyword Arguments substitute** – List-like of tuples where the first item in a tuple is a Pyomo variable to be replaced and the second item in the tuple is a Pyomo variable to replace it with. This transformation is not reversible.

4.5 Unit Model Library

4.5.1 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 `has_holdup = True` this is a reference to `control_volume.volume`, otherwise a Var attached to the Unit Model
Q_t	heat	Only if `has_heat_transfer = True`, reference to `control_volume.heat`

CSTR Class

class idaes.unit_models.cstr.**CSTR**(**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 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:** { **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 flowsheet, **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`) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (CSTR) New instance

CSTRData Class

class idaes.unit_models.cstr.**CSTRData** (*component*)
> Standard CSTR Unit Model Class

> **build** ()
> > Begin building model (pre-DAE transformation). :param None:

> > **Returns** None

4.5.2 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.

Vari-able	Name	Notes
V_t	vol-ume	If `has_holdup = True` this is a reference to `control_volume.volume`, otherwise a Var attached to the Unit Model
Q_t	heat	Only if `has_heat_transfer = True`, reference to `control_volume.heat`

EquilibriumReactor Class

class idaes.unit_models.equilibrium_reactor.**EquilibriumReactor**(**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 ProcessBlockData 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 momentum balances, **MomentumBalanceType.pressureTotal** - single pressure balance for material, **MomentumBalance-**

Type.pressurePhase - pressure balances for each phase, **MomentumBalance-Type.momentumTotal** - single momentum balance for material, **MomentumBalance-Type.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 equlibirum 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 flowsheet, **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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (EquilibriumReactor) New instance

EquilibriumReactorData Class

class idaes.unit_models.equilibrium_reactor.**EquilibriumReactorData**(*component*)
   Standard Equilibrium Reactor Unit Model Class

   build()
      Begin building model.

> **Parameters** `None` –

> **Returns** None

4.5.3 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 StateBlock (named properties), each with one Outlet Port (named outlet). Feed Blocks also contain References to the state variables defined within the StateBlock

Additional Constraints

Feed Blocks write no additional constraints to the model.

Variables

Feed blocks add no additional Variables.

Feed Class

class idaes.unit_models.feed.**Feed**(**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 ProcessBlockData 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 flowsheet, **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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (Feed) New instance

FeedData Class

class idaes.unit_models.feed.**FeedData**(*component*)

> Standard Feed Block Class

build()

> Begin building model.
>
> > **Parameters None** –
> >
> > **Returns** None

initialize(*state_args={}*, *outlvl=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
> >
> > - **optarg** – solver options dictionary object (default={'tol': 1e-6})
> >
> > - **solver** – str indicating which solver to use during initialization (default = 'ipopt')
> >
> > **Returns** None

4.5.4 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 (FeedFlash) 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.unit_models.feed_flash.**FeedFlash**(**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 ProcessBlockData 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 flowsheet, **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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (FeedFlash) New instance

FeedFlashData Class

class idaes.unit_models.feed_flash.**FeedFlashData**(*component*)
> Standard Feed block with phase equilibrium

build()
> Begin building model.

> > **Parameters None** –

> > **Returns** None

4.5.5 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 mutile 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 temeprature 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 problme 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 writting 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 calacuations, 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
dynamic	False
include_holdup	False
material_balance_type	MaterialBalanceType.componentPhase
energy_balance_type	EnergyBalanceType.enthalpyTotal
momentum_balance_type	MomentumBalanceType.pressureTotal
has_phase_equilibrium	True
has_heat_transfer	True
has_pressure_change	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.unit_models.flash.**Flash**(*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 ProcessBlockData 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** - oulet 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 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.}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (Flash) New instance

FlashData Class

class idaes.unit_models.flash.**FlashData**(*component*)

> Standard Flash Unit Model Class

build()

> Begin building model (pre-DAE transformation).
>
> > **Parameters None** –
> >
> > **Returns** None

4.5.6 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.

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

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):

$$0 = g_{partial,t,j} + \sum_e \left(L_{t,e} \times \alpha_{j,e} \right)$$

where $g_{partial,t,j}$ 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. $g_{partial,t,j}$ and $\alpha_{j,e}$ come from the outlet StateBlock.

GibbsReactor Class

class idaes.unit_models.gibbs_reactor.**GibbsReactor**(**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 ProcessBlockData 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:** { **EnergyBalance- Type.useDefault - refer to property package for default balance type **Energy- BalanceType.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, **MomentumBalance- Type.pressureTotal** - single pressure balance for material, **MomentumBalance- Type.pressurePhase** - pressure balances for each phase, **MomentumBalance- Type.momentumTotal** - single momentum balance for material, **MomentumBalance- Type.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 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.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block- Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

  Returns (GibbsReactor) New instance

GibbsReactorData Class

class idaes.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 repect to product species mole flow rates and the multiples are specified as constraints.

build()
> Begin building model (pre-DAE transformation).
>
> > **Parameters None** –
> >
> > **Returns** None

4.5.7 Heater

The Heater model is a simple 0D model that adds or removes heat from a material stream.

Example

```python
import pyomo.environ as pe # Pyomo environment
from idaes.core import FlowsheetBlock, StateBlock
from idaes.unit_models import Heater
from idaes.property_models 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.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 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:** { **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 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.}

- **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 (Heater) New instance

HeaterData Class

class idaes.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

4.5.8 HeatExchanger (0D)

The HeatExchanger model can be imported from idaes.unit_models, while additional rules and utility functions can be imported from idaes.unit_models.heat_exchanger.

Example

The example below demonstrates how to initialize the HeatExchanger model, and override the default temperature difference calculation.

```python
import pyomo.environ as pe # Pyomo environment
from idaes.core import FlowsheetBlock, StateBlock
from idaes.unit_models import HeatExchanger
from idaes.unit_models.heat_exchanger import delta_temperature_amtd_callback
from idaes.property_models 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.heat_exchanger = HeatExchanger(default={
```

(continues on next page)

```
        "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.unit_models.heat_exchanger.**HeatExchanger**(**args*, ***kwargs*)

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 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:** { **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 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.}

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 **Energy-

BalanceType.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 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.}

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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (HeatExchanger) New instance

class idaes.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=1000*)
    Heat exchanger initialization method.

Parameters

- **state_args_1** – a dict of arguments to be passed to the property initialization for side_1 (see documentation of the specific property package) (default = {}).

- **state_args_2** – a dict of arguments to be passed to the property initialization for side_2 (see documentation of the specific property package) (default = {}).

- **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **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 transfered (default = 10000)

Returns None

set_scaling_factor_energy(*f*)

This function sets scaling_factor_energy for both side_1 and side_2. This factor multiplies the energy balance and heat transfer equations in the heat exchnager. The value of this factor should be about 1/(expected heat duty).

Parameters **f** – Energy balance scaling factor

Callbacks

A selection of functions for constructing the delta_temperature variable or expression are provided in the idaes.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.unit_models.heat_exchanger.**delta_temperature_lmtd_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.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.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.

4.5.9 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, collocation 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 TransformationFactory

- 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
dynamic	useDefault
has_holdup	False
material_balance_type	'componentTotal'
energy_balance_type	'enthalpyTotal'
momentum_balance_type	'pressureTotal'
has_phase_equilibrium	False
has_heat_transfer	True
has_pressure_change	False

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.unit_models.heat_exchanger_1D.**HeatExchanger1D**(**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 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:** { **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:** { **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 transformating 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:** { **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 transformating 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), - WallConduction-
> > > Type.one_dimensional - 1D wall model along the thickness of the tube, - WallCon-
> > > ductionType.two_dimensional - 2D wall model along the lenghth and thickness of the
> > > tube

> > - **initialize** (`dict`) – ProcessBlockData config for individual elements. Keys are Block-
> >   Data 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 overide
> >   the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (HeatExchanger1D) New instance

HeatExchanger1dDataClass

class idaes.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=5*, *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 * 0 = Use default idaes.init logger
>   >   setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for
>   >   each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final
>   >   initialization status and exceptions * 6 = No output

>   >   - **optarg** – solver options dictionary object (default={'tol': 1e-6})

>   >   - **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')

>   >   **Returns** None

4.5.10 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	Symbol	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 = MomemntumMixingType.minimize

Parameters

Mixer units have the following parameters:

Variable Name	Symbol	Notes
eps_pressure	ϵ	Only if momentum_mixing_type = MomemntumMixingType.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 *componentPhase*:

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 *componentTotal*:

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:

$$P_{min,t,i} = smin(P_{min,t,i-1}, P_{t,i}, eps)$$

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 $smin$ 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, them 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.unit_models.mixer.**Mixer**(**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 ProcessBlockData 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 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.}
>   >
>   > **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 minimimum 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 StateBock 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (Mixer) New instance

MixerData Class

class idaes.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 specified 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 mutiple 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 fuction.

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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={})

- **solver** – str indicating whcih 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=6*)
> 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 = mimimum 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 = mimimum 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.

4.5.11 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 control_volume.length
A	area	Reference to control_volume.area
V	volume	Reference to control_volume.volume
$Q_{t,x}$	heat	Only if has_heat_transfer = True, reference to holdup.heat
$\Delta P_{t,x}$	deltaP	Only if has_pressure_change = True, reference to holdup.deltaP

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.unit_models.plug_flow_reactor.**PFR**(**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 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:** { **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 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.}

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 transformating 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 to use when transforming length domain, **default** - 3.

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (PFR) New instance

PFRData Class

class idaes.unit_models.plug_flow_reactor.**PFRData**(*component*)
> Standard Plug Flow Reactor Unit Model Class

> **build**()
>> Begin building model (pre-DAE transformation).
>>
>>> **Parameters None** –
>>>
>>> **Returns** None

4.5.12 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 has_rate_reactions = True, reference to control_volume.rate_reaction_extent
$W_{mechanical,t}$	work_mechanical	Reference to control_volume.work
$W_{fluid,t}$	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.unit_models.pressure_changer.**PressureChanger**(*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 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:** { **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 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.}
>
> - **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (PressureChanger) New instance

PressureChangerData Class

class idaes.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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

> > • **optarg** – solver options dictionary object (default={'tol': 1e-6})

> > • **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')

> **Returns** None

initialize (*state_args=None*, *routine=None*, *outlvl=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

> > • **optarg** – solver options dictionary object (default={'tol': 1e-6})

> > • **solver** – str indicating whcih 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

4.5.13 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.unit_models.product.**Product**(**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 ProcessBlockData 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 flowsheet, **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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (Product) New instance

ProductData Class

class idaes.unit_models.product.**ProductData**(*component*)

> Standard Product Block Class
>
> **build**()
>
> > Begin building model.

Parameters `None` –

Returns None

`initialize` (*state_args={}*, *outlvl=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- `optarg` – solver options dictionary object (default={'tol': 1e-6})

- `solver` – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

4.5.14 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 *split_basis*

Constraints

Separator units have the following Constraints, unless *ideal_separation* is True.

- If *material_balance_type* is *componentPhase*:

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 *componentTotal*:

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.unit_models.separator.**Separator**(**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 ProcessBlockData 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 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.}

    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 **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** - oulet 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 spearation 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 StateBock 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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Separator) New instance

SeparatorData Class

class idaes.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 mutiple 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*)
   Creates outlet Port objects and tries to partiton 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 SeparatorData. 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=6*, *optarg={}*, *solver='ipopt'*, *hold_state=False*)
   Initialization routine for separator (default solver ipopt)

>   **Keyword Arguments**

- **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default=None)

- **solver** – str indicating whcih 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

partition_outlet_flows(*mb*, *outlet_list*)

Creates outlet Port objects and tries to partiton 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=6*)

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

4.5.15 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.unit_models.statejunction.**StateJunction**(**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 ProcessBlockData 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 flowsheet, **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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (StateJunction) New instance

StateJunctionData Class

class idaes.unit_models.statejunction.**StateJunctionData**(*component*)

> Standard StateJunction Unit Model Class
>
> **build**()
>
> > Begin building model. :param None:

> **Returns** None

initialize(*state_args={}*, *outlvl=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

>   - **optarg** – solver options dictionary object (default={'tol': 1e-6})

>   - **solver** – str indicating which solver to use during initialization (default = 'ipopt')

>   **Returns** None

4.5.16 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
$deltaP_t$	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.unit_models.stoichiometric_reactor.**StoichiometricReactor**(**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 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:** { **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 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 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.}

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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (StoichiometricReactor) New instance

StoichiometricReactorData Class

class idaes.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

4.5.17 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.unit_models.translator.**Translator**(**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 ProcessBlockData 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 sumation 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 notcalculate 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Translator) New instance

TranslatorData Class

class idaes.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=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

4.5.18 Power Generation 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 used 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. 1: Feedwater Heater

Example

The example below shows how to setup a feedwater heater with all tree sections. The feedwater flow rate, steam conditions, heat transfer coefficents and areas are not necessarily realistic.

```python
import pyomo.environ as pyo
from idaes.core import FlowsheetBlock
from idaes.unit_models.heat_exchanger import (delta_temperature_underwood_callback,
    delta_temperature_lmtd_callback)
from idaes.property_models import iapws95
from idaes.unit_models.power_generation import FWH0D

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 = FWH0D(default={
        "has_desuperheat":True,
        "has_drain_cooling":True,
        "has_drain_mixer":True,
        "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 (*FWHCondensing0D*)
desuperheat	has_desuperheat	Desuperheating section (*HeatExchanger*)
cooling	has_drain_cooling	Drain cooling section (*HeatExchanger*)
drain_mix	has_drain_mixer	Mixer for steam and other FWH drain (*Mixer*)

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 *IAWPS95* 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 are 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.unit_models.power_generation.feedwater_heater_0D.**FWHCondensing0D**(*args, **kwargs*)

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:** { **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 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.}

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:** { **Materi-**

alBalanceType.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 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.}

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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns: (FWHCondensing0D) New instance

FWHCondensing0DData Class

class idaes.unit_models.power_generation.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 acti-
> > vates the constraint and calculates a steam inlet flow rate.

Turbine (Inlet Stage)

This is a steam power generation turbine model for the inlet stage. 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

```python
from pyomo.environ import ConcreteModel, SolverFactory, TransformationFactory
from idaes.core import FlowsheetBlock
from idaes.unit_models.power_generation import TurbineInletStage
from idaes.property_models import iapws95

m = ConcreteModel()
m.fs = FlowsheetBlock(default={"dynamic": False})
m.fs.properties = iapws95.Iapws95ParameterBlock()
m.fs.turb = TurbineInletStage(default={"property_package": m.fs.properties})
hin = iapws95.htpx(T=880, P=2.4233e7)
# 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. See the variables section for more information.

Model Structure

The turbine inlet stage model contains one *ControlVolume0DBlock block* called control_volume and inherits the *PressureChanger model* using the isentropic option.

Variables

The variables below are defined in the TurbineInletStage model. Additional variables are inherited from the *PressureChanger model* model.

Variable	Symbol	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	Symbol	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 *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:

$$\eta_{isen} = 2\frac{V_{rbl}}{V_0}\left[\left(\sqrt{1-R}-\frac{V_{rbl}}{V_0}\right)+\sqrt{\left(\sqrt{1-R}-\frac{V_{rbl}}{V_0}\right)^2+R}\right]$$

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. 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`.

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.unit_models.power_generation.turbine_inlet.**TurbineInletStage**(**args*, ***kwargs*)

> 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 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:** { **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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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, **MomentumBalance-Type.pressureTotal** - single pressure balance for material, **MomentumBalance-Type.pressurePhase** - pressure balances for each phase, **MomentumBalance-Type.momentumTotal** - single momentum balance for material, **MomentumBalance-Type.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 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.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (TurbineInletStage) New instance

TurbineInletStageData Class

class idaes.unit_models.power_generation.turbine_inlet.**TurbineInletStageData**(*component*)

build()

Parameters None –

Returns None

initialize (*state_args={}*, *outlvl=6*, *solver='ipopt'*, *optarg={'max_iter': 30, 'tol': 1e-06}*)
Initialize the inlet 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** (*int*) – Amount of output (0 to 3) 0 is lowest
> - **solver** (*str*) – Solver to use for initialization
> - **optarg** (*dict*) – Solver arguments dictionary

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

```python
from pyomo.environ import ConcreteModel, SolverFactory
from idaes.core import FlowsheetBlock
from idaes.unit_models.power_generation import TurbineOutletStage
from idaes.property_models import iapws95

m = ConcreteModel()
m.fs = FlowsheetBlock(default={"dynamic": False})
m.fs.properties = iapws95.Iapws95ParameterBlock()
m.fs.turb = TurbineOutletStage(default={"property_package": m.fs.properties})
# 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 outlet stage model contains one *ControlVolume0DBlock block* called control_volume and inherits the *PressureChanger model* using the isentropic option.

Variables

The variables below are defined int the TurbineInletStage model. Additional variables are in inherited from the *PressureChanger model* 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	Symbol	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.

$$\text{TEL} = 1 \times 10^6 * \left(-0.0035 f^5 + 0.022 f^4 - 0.0542 f^3 + 0.0638 f^2 - 0.0328 f + 0.0064\right)$$

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:

$$\eta_{isen} = \eta_{dry} x \left(1 - 0.65(1 - x)\right) * \left(1 + \frac{\text{TEL}}{\Delta h_{isen}}\right)$$

Where x is the steam quality (vapor fraction).

The pressure-flow relation is given by the Stodola Equation:

$$\dot{m}\sqrt{Tin - 273.15} = C_{flow} P_{in} \sqrt{1 - Pr^2}$$

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. 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`.

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.unit_models.power_generation.turbine_outlet.**TurbineOutletStage**(**args*, ***kwargs*)

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 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:** { **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, **MomentumBalance-**

       Type.momentumTotal - single momentum balance for material, **MomentumBalance-Type.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 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.}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

      Returns (TurbineOutletStage) New instance

TurbineOutletStageData Class

class idaes.unit_models.power_generation.turbine_outlet.**TurbineOutletStageData** (*component*)

    build()

        Parameters None –

        Returns None

    initialize (*state_args={}*, *outlvl=6*, *solver='ipopt'*, *optarg={'max_iter': 30, 'tol': 1e-06}*)
        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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **solver** (*str*) – Solver to use for initialization

- **optarg** (*dict*) – Solver arguments dictionary

Turbine (Stage)

This is a steam power generation turbine model for the stages between the inlet and outlet. This model inherits the *PressureChanger model* with the isentropic options. The initialization scheme is the same as the *TurbineInletStage model*.

Example

```python
from pyomo.environ import ConcreteModel, SolverFactory

from idaes.core import FlowsheetBlock
from idaes.unit_models.power_generation import TurbineStage
from idaes.property_models import iapws95

m = ConcreteModel()
m.fs = FlowsheetBlock(default={"dynamic": False})
m.fs.properties = iapws95.Iapws95ParameterBlock()
m.fs.turb = TurbineStage(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.efficiency_isentropic[:].fix(0.8)
m.fs.turb.ratioP[:].fix(0.7)
m.fs.turb.initialize()
```

Variables

This model adds a variable to the base `PressureChanger model` to account for mechanical efficiency .

Variable	Symbol	Index Sets	Doc
efficiency_mech	η_{mech}	None	Mechanical Efficiency (accounts for losses in bearings...)

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 $\left(\frac{P_{out}}{P_{in}}\right)$

$\eta_{isentropic,t}$ efficiency_isentropic Isentropic assumption only

Expressions

This model provides two expressions that are not available in the pressure changer model.

Variable	Symbol	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]

Constraints

There are no additional constraints.

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.

TurbineStage Class

class idaes.unit_models.power_generation.turbine_stage.**TurbineStage**(*args, **kwargs*)

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 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:** { **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 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.}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (TurbineStage) New instance

TurbineStageData Class

class idaes.unit_models.power_generation.turbine_stage.**TurbineStageData**(*component*)

build()

Parameters None –

Returns None

initialize(*state_args={}, outlvl=6, 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

- **state_args** (*dict*) – Initial state for property initialization

- **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **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 mutistage 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. 2: 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 feedwater 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.

```python
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.unit_models.power_generation import (
    TurbineMultistage, TurbineStage, TurbineInletStage, TurbineOutletStage)
from idaes.property_models 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 = TurbineMultistage(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
```

(continues on next page)

```python
    "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})
# 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 effiency and pressure ratios of stages other than inlet and outlet
for i, s in turb.hp_stages.items():
    s.ratioP[:] = 0.88
    s.efficiency_isentropic[:] = 0.9
for i, s in turb.ip_stages.items():
    s.ratioP[:] = 0.85
    s.efficiency_isentropic[:] = 0.9
for i, s in turb.lp_stages.items():
    s.ratioP[:] = 0.82
    s.efficiency_isentropic[:] = 0.9
# 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 = \
        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)
```

```python
# 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 (*Separator*)
`throttle_valve`	Admission Arcs	Throttle valves for each admission arc (*SteamValve*)
`inlet_stage`	Admission Arcs	Parallel inlet turbine stages that represent admission arcs (*TurbineInlet*)
`inlet_mix`	None	Mixer to combine the streams from each arc back to one stream (*Mixer*)
`hp_stages`	HP stages	Turbine stages in the high-pressure section (*TurbineStage*)
`ip_stages`	IP stages	Turbine stages in the intermediate-pressure section (*TurbineStage*)
`lp_stages`	LP stages	Turbine stages in the low-pressure section (*TurbineStage*)
`hp_splits`	subset of HP stages	Extraction splitters in the high-pressure section (*Separator*)
`ip_splits`	subset of IP stages	Extraction splitters in the high-pressure section (*Separator*)
`lp_splits`	subset of LP stages	Extraction splitters in the high-pressure section (*Separator*)
`outlet_stage`	None	The final stage in the turbine, which calculates exhaust losses (*TurbineOutlet*)

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.

TurbineMultistage Class

class idaes.unit_models.power_generation.turbine_multistage.**TurbineMultistage**(**args*,
**kwargs)

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 ProcessBlockData config
>
> > **Keys**
> >
> > **dynamic** Indicates whether the model is dynamic.
> >
> > **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}
> >
> > **has_phase_equilibrium** Argument indicating whether phase equilibrium should be cal-
> > culated for the resulting mixed stream, **default** - False. **Valid values:** { **True** - calcu-
> > late phase equilibrium in mixed stream, **False** - do not calculate equilibrium in mixed
> > stream.}
> >
> > **material_balance_type** Indicates what type of mass balance should be constructed, **de-
> > fault** - MaterialBalanceType.componentTotal'. **Valid values:** { **MaterialBalance-
> > Type.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use
> > phase component balances, **MaterialBalanceType.componentTotal** - use total com-
> > ponent balances, **MaterialBalanceType.elementTotal** - use total element balances,
> > **MaterialBalanceType.total** - use total material balance.}
> >
> > **property_package** Property parameter object used to define property calculations, **de-
> > fault** - 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 prop-
> > erty package for documentation.}
> >
> > **num_parallel_inlet_stages** Number of parallel inlet stages to simulate partial arc admis-
> > sion. Default=4
> >
> > **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 addtional 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 addtional 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 addtional 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`) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (TurbineMultistage) New instance

TurbineMultistageData Class

class idaes.unit_models.power_generation.turbine_multistage.**TurbineMultistageData**(*component*)

build()
General build method for UnitModelBlockData. 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=6*, *solver='ipopt'*, *optarg={'max_iter': 35, 'tol': 1e-06}*)
Initialize

throttle_cv_fix(*value*)
Fix the thottle 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

Steam/Water Valve

This is a steam power generation turbine model for the stages between the inlet and outlet. 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

```python
from pyomo.environ import ConcreteModel, SolverFactory, TransformationFactory

from idaes.core import FlowsheetBlock
from idaes.unit_models.power_generation import SteamValve
from idaes.property_models 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 = SteamValve(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)
```

Parameters

Expression	Symbol	Index Sets	Doc
`flow_scale`	s_f	None	Factor for scaling the pressure-flow equation, should be same magnitude as expected flow rate

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.

$$\frac{1}{s_f^2} F^2 = \frac{1}{s_f^2} C_v^2 \left(P_{in} - P_{out}\right) f(x)^2$$

If the `phase` option is set to `"Vap"` the following equation describes the pressure-flow relation.

$$\frac{1}{s_f^2} F^2 = \frac{1}{s_f^2} C_v^2 \left(P_{in}^2 - P_{out}^2\right) f(x)^2$$

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.

SteamValve Class

class idaes.unit_models.power_generation.valve_steam.**SteamValve**(**args*, ***kwargs*)

Basic steam valve models

> **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 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}
> >
> > **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, **MaterialBal-**

anceType.componentTotal - use total component balances, **MaterialBalance-Type.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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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:** { **Mo-mentumBalanceType.none** - exclude momentum balances, **MomentumBalance-Type.pressureTotal** - single pressure balance for material, **MomentumBalance-Type.pressurePhase** - pressure balances for each phase, **MomentumBalance-Type.momentumTotal** - single momentum balance for material, **MomentumBalance-Type.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) - ThermodynamicAs-sumption.isentropic - ThermodynamicAssumption.pump - ThermodynamicAssump-tion.adiabatic

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.}

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, ValveFunction-Type.equal_percentage, ValveFunctionType.custom}

valve_function_rule This is a rule that returns a time indexed valve function expression. This is required only if valve_function==ValveFunctionType.custom

phase Expected phase of fluid in valve in {"Liq", "Vap"}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (SteamValve) New instance

SteamValveData Class

class idaes.unit_models.power_generation.valve_steam.**SteamValveData**(*component*)

>>> **build**()

>>>>>> **Parameters None** –

>>>>>> **Returns** None

>>> **initialize**(*state_args={}*, *outlvl=6*, *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**

>>>>>>> - **state_args** (*dict*) – Initial state for property initialization

>>>>>>> - **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

>>>>>>> - **solver** (*str*) – Solver to use for initialization

>>>>>>> - **optarg** (*dict*) – Solver arguments dictionary

BoilerHeatExchanger

The BoilerHeatExchanger model can be used to represent boiler heat exchangers in sub-critical and super critical power plant flowsheets (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 and radiative 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.unit_models.power_generation, while additional rules and utility functions can be imported from idaes.unit_models.power_generation.boiler_heat_exchanger.

Example

The example below demonstrates how to initialize the BoilerHeatExchanger model, and override the default temperature difference calculation.

```python
# Import Pyomo libraries
from pyomo.environ import ConcreteModel, SolverFactory, value
# Import IDAES core
from idaes.core import FlowsheetBlock
# Import Unit Model Modules
from idaes.property_models import iapws95
# import ideal flue gas prop pack
from idaes.unit_models.power_generation.IdealProp_FlueGas import FlueGasParameterBlock
# Import Power Plant HX Unit Model
from idaes.unit_models.power_generation.boiler_heat_exchanger import \
    BoilerHeatExchanger, TubeArrangement, \
    DeltaTMethod
```

(continues on next page)

```python
import pyomo.environ as pe # Pyomo environment
from idaes.core import FlowsheetBlock, StateBlock
from idaes.unit_models import HeatExchanger
from idaes.unit_models.heat_exchanger import delta_temperature_amtd_callback
from idaes.property_models 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()

# 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.htpx(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)
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)
```

```python
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
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. 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 `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	t	Heat transferred from hot side to the cold side
area	A	None	Heat transfer area
U	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 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:

$$\frac{1}{U} * fcorrection_{htc} = [\frac{1}{hconv_{tube}} + \frac{1}{hconv_{shell}} + r + tube_{rfouling} + shell_{rfouling}]$$

$$hconv_{tube} = \frac{Nu_{tube}k}{tube_{di}}$$

$$Nu_{tube} = 0.023Re_{tube}^{0.8}Pr_{tube}^{0.4}$$

$$Pr_{tube} = \frac{Cp\mu}{kMw}$$

$$Re_{tube} = \frac{tube_{di}V\rho}{\mu}$$

$$hconv_{shell} = \frac{Nu_{shell}k_{fluegas}}{tube_{do}}$$

$$Nu_{shell} = f_{arrangement}0.33Re_{tube}^{0.6}Pr_{tube}^{0.3333}$$

$$Pr_{shell} = \frac{Cp\mu}{kMw}$$

$$Re_{shell} = \frac{tube_{do}V\rho}{\mu}$$

$$tube_{do} = 2 * tube_{thickness} + tube_{di}$$

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 m^2 / W)

- k : thermal conductivity of the tube wall (W / m / K)

- ρ : density (kg/m^3)

- μ : viscocity (kg/m/s)

- tube_r_fouling : tube side fouling resistance (K m^2 / W)

- shell_r_fouling : shell side fouling resistance (K m^2 / W)

- fcorrection_htc: correction factor for overall heat trasnfer

- 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 data). This variable is also called cleanniness factor (when = 1, its considered a clean tube with no fouling).

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 `BoilerHeatExchanger` includes an argument to compute heat tranfer 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, $deltaP_{uturn} and friction_{factor}$ are calculated

Tube side:

$$\Delta P_{tube} = \Delta P_{tubefriction} + \Delta P_{tubeuturn} - elevation * g * \frac{\rho_{in} + \rho_{out}}{2}$$

$$\Delta P_{tubefriction} = f(tube_{di}\rho, V_{tube}, number of tubes, tube_{length})$$

$$\Delta P_{tubeuturn} = f(\rho, v_{tube}, k_{lossuturn})$$

where:

- $k_{lossuturn}$: pressure loss coeficient of a tube u-turn

- g : is the acceleration of gravity 9.807 (m/s^2)

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:

In-line: $\Delta P_{shellfriction} = \dfrac{0.044 + \dfrac{0.08(\frac{P_x}{tube_{do}})}{(\frac{P_y}{tube_{do}}-1)^{\left(0.43+\frac{1.13}{(\frac{P_x}{tube_{do}})}\right)}}}{Re^{0.15}}$

Staggered: $\Delta P_{shellfriction} = \dfrac{0.25 + \dfrac{0.118}{(\frac{P_y}{tube_{do}}-1)^{1.08}}}{Re^{0.16}}$

Figure. 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.unit_models.power_generation.boiler_heat_exchanger.**BoilerHeatExchanger**(**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 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.}

P_x

Y_{min} V P_y

D_o

Flow Direction

P_x

In-line Tube Arrangement

Staggered Tube Arrange

Fig. 3: Tube Arrangement

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 ethalpy bal-

ance for material, **EnergyBalanceType.enthalpyPhase** - ethalpy 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 T calculations (default = 'counter-current')

– 'counter-current' : counter-current flow arrangement

    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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

    Returns (BoilerHeatExchanger) New instance

class idaes.unit_models.power_generation.boiler_heat_exchanger.**BoilerHeatExchangerData**(*comp*
    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={'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 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 infomation (tee=True)

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating whcih 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

4.6 Property Model Library

4.6.1 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 the 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 show in the table below.

Equation	u	w	$Omega_A$	$Omega_B$	$kappa_j$
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 :math'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:

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

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/m3
`dens_mass_phase`	Mass density indexed by phase	kg/m3
`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.property_models.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.property_models.cubic_eos.cubic_prop_pack.**CubicStateBlock**(**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 ProcessBlockData 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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (CubicStateBlock) New instance

class idaes.property_models.cubic_eos.cubic_prop_pack.**CubicStateBlockData**(*component*)
> 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.

4.6.2 Vapor-Liquid Equilibrium Property Models (Ideal Gas - Non-ideal Liquids)

This property package supports phase equilibrium calucations 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 the 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 asuming 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 probablity 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 calucalations 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 calcualtions 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 enthapies (`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 calcuated 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]:

$$\log \gamma_i = \frac{\sum_j x_j \tau_j G_{ji}}{\sum_k x_k G_{ki}} + \sum_j \frac{x_j G_{ij}}{\sum_k x_k G_{kj}} [\tau_{ij} - \frac{\sum_m x_m \tau_{mj} G_{mj}}{\sum_k x_k G_{kj}}]$$

$$G_{ij} = \exp(-\alpha_{ij} \tau_{ij})$$

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
entrolpy_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 2: 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 3: 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.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffPar**

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

> **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`) – 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 (ActivityCoeffParameterBlock) New instance

class idaes.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffSta**

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

  > **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`) – 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 (ActivityCoeffStateBlock) New instance

class idaes.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffSta**
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 Ghouse, A Lee, LT Biegler, DC Miller. "A Smooth, Square Flash Formulation for Equation Oriented Flowsheet Optimization", Computer Aided Chemical Engineering 44, 871-876, 2018

4.6.3 Water/Steam - IAPWS95

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.

Example

Theses modules can be imported as:

```
from idaes.property_models import iapws95
```

The Heater unit model *example*, provides a simple example for using water properties.

```python
import pyomo.environ as pe # Pyomo environment
from idaes.core import FlowsheetBlock, MaterialBalanceType
from idaes.unit_models import Heater
from idaes.property_models 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(default={
  "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.

```python
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 StateBlocks and PropertyParameterBlocks work see the *StateBlock documentation*.

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 to be used internally and all external function calls use units of kg, kJ, kPa, and K. These generally are not needed by the end user.

Methods

These methods use the IAPWS-95 formulation for scientific use for thermodynamic properties (*Wagner and Pruss, 2002*; *IAPWS, 2016*). To solve the phase equilibrium, the method of *Akasaka (2008)* was used. For solving these equations, some relations from the IAPWS-97 formulation for industrial use are used as initial values (*Wagner et al., 2002*). The industrial formulation is slightly discontinuous between different regions, so it may not be suitable for optimization. In addition to thermodynamic quantities, viscosity and thermal conductivity are calculated (*IAPWS, 2008*; *IAPWS, 2011*).

External Functions

The IAPWS-95 formulation uses density and temperature as state variables. For most applications those state variables are not the most convenient choices. Using other state variables requires solving equations to get density and temperature from the chosen state variables. These equations can have numerous solutions only one of which is physically meaningful. Rather than solve these equations as part of the full process simulation, external functions were developed that can solve the equations required to change state variables and guarantee the correct roots.

The external property functions are written in C++ and complied such that they can be called by AMPL solvers. See the *Installation* page for information about compiling these functions. The external functions provide both first and second derivatives for all property function calls, however at phase transitions some of these functions may be non-smooth.

IDAES Framework Wrapper

A wrapper for the external functions is provided for compatibility with the IDAES framework. Most properties are available as Pyomo Expressions from the wrapper. Only the state variables are model variables. Benefits of using mostly expressions in the property package are: no initialization is required specifically for the property package, the model has fewer equations, and all properties can be easily calculated after the model is solved from the state variable values even if they were not used in the model. Calls to the external functions are used within expressions so users do not need to directly call any functions. The potential downside of the extensive use of expressions here is that combining the expressions to form constraints could yield equations that are more difficult to solve than, they would have been if an equivalent system of equations was written with more variables and simpler equations. Quantifying the effect of writing larger equations with fewer variables is difficult. Experience suggests in this particular case more expressions and fewer variables is better.

Although not generally used, the wrapper provides direct access to the ExternalFunctions, including intermediate functions. For more information see section *ExternalFunctions*. These are mostly available for testing purposes.

Phase Presentation

The property package wrapper can present fluid phase information to the IDAES framework in different ways. See the *class reference* for details on how to set these options. The `phase_presentation=PhaseType.MIX` option looks like one phase called "Mix" to the IDAES framework. The property package will calculate a phase fraction. This will bypass any two phase handling equations written for unit models, and should work with any unit model options as long as you do not want to separate the phases. The benefit of this option is that it can potentially lead to a simpler set of equations.

The `phase_presentation=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_equlibrium=True`. This will still write a material balance per phase, but will add a phase generation term to the model. For the IAPWS-95 package, it is generally recommended that specifying total material balances is best because it results in a problem with fewer variables.

There are also two single phase options `phase_presentation=PhaseType.L` and `phase_presentation=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 know 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.

Pressure-Enthalpy Formulation

The advantage of this choice of state variables is that it is very 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 not be possible.

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 not included in the state variable set. For single phase, the complementarity constraint is also deactivated.

A complementarity constraint is required for the T-P-x formulation. 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 by an IDAES utility function 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 saturated pressure 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 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 does provide another constraint, that $P = P_{\text{sat}}$. When two phases are present x can be found by the unit model energy balance and the temperature will be T_{sat}.

An alternative approach is sometimes useful. If you know for certain that you have 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"`

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)
`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)
`therm_cond_phase[phase]`	Thermal conductivity of phase (W/K/m)
`vapor_frac`	Vapor fraction, if PH form
`visc_d_phase[phase]`	Viscosity of phase (Pa/s)
`visc_k_phase[phase]`	Kinimatic viscosity of phase (m^2/s)
`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 ExternalFuctions 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 $\frac{T_c}{T}$, 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 $\frac{f}{RT}$.

Pyomo Function	C Function	Returns	Arguments
func_p	p	pressure (kPa)	δ, 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_tau	tau	τ (unitless)	h (kJ/kg), P (kPa)
func_vf	vf	vapor fraction (unitless)	h (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

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.

Convenience Functions

idaes.property_models.iapws95.**htpx**(*T*, *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 one (and only one) of arguments P and x.
>
> **Parameters**
>
> - **T** – Temperature [K] (between 200 and 3000)
>
> - **P** – Pressure [Pa] (between 1 and 1e9), None if saturated steam
>
> - **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.property_models.iapws95.**Iapws95StateBlock**(**args*, ***kwargs*)

> 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 ProcessBlockData 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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> > **Returns** (Iapws95StateBlock) New instance

Iapws95StateBlockData Class

class idaes.property_models.iapws95.**Iapws95StateBlockData**(*component*)

> This is a property package for calculating thermophysical properties of water

> **build**(**args*)
> > 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**()
> > 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**(*p*)
> > Method which returns a valid expression for enthalpy density to use in the energy balances.

> **get_enthalpy_flow_terms**(*p*)
> > Method which returns a valid expression for enthalpy flow to use in the energy balances.

> **get_material_density_terms**(*p, j*)
> > Method which returns a valid expression for material density to use in the material balances .

> **get_material_flow_terms**(*p, j*)
> > Method which returns a valid expression for material flow to use in the material balances.

Iapws95ParameterBlock Class

class idaes.property_models.iapws95.**Iapws95ParameterBlock**(**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 ProcessBlockData 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 indepedently. The LG option presents as two sperate 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}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Iapws95ParameterBlock) New instance

Iapws95ParameterBlockData Class

class idaes.property_models.iapws95.**Iapws95ParameterBlockData**(*component*)

build()
General build method for PropertyParameterBlocks. Inheriting models should call super().build.

Parameters None –

Returns None

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

4.7 Visualization

4.7.1 Contents

Drawing heat exchanger network diagrams

The following example demonstrates how to generate a heat exchanger network diagram.

In the code below, different streams are defined in the *streams* list. For each stream, we expect a name (*name*), a list of temperatures (*temps*) and a *type* field specifying if this is a hot stream (*HENStreamType.hot*) or a cold one (*HENStreamType.cold*).

The *exchangers* list defines the heat exchangers. Each exchanger is defined by its hot/cold stream (*hot*, *cold*) which must match one of the streams in the *streams* list above. We also require for each exchanger the area (*A*),the amount of heat transferred from one stream to another (*Q*), annual cost (*annual_cost*) and stage (*stg*). If the *utility_type* key is passed and it's set to *HENStreamType.cold_utility* then we draw the cold stream of the exchanger as water. If the *utility_type* key is passed and it's set to *HENStreamType.hot_utility* then we draw the hot stream of the exchanger as steam.

The color-codes of each stage are picked randomly in the final diagram.

```python
from bokeh.io import output_notebook
from bokeh.plotting import show
from idaes.vis.plot import Plot
from idaes.vis.plot_utils import HENStreamType

exchangers = [
    {'hot': 'H2', 'cold': 'C1', 'Q': 1400, 'A': 159, 'annual_cost': 28358, 'stg': 2},
    {'hot': 'H1', 'cold': 'C1', 'Q': 667, 'A': 50, 'annual_cost': 10979, 'stg': 3},
    {'hot': 'H1', 'cold': 'C1', 'Q': 233, 'A': 10, 'annual_cost': 4180, 'stg': 1},
    {'hot': 'H1', 'cold': 'C2', 'Q': 2400, 'A': 355, 'annual_cost': 35727, 'stg': 2},
    {'hot': 'H2', 'cold': 'W', 'Q': 400, 'A': 50, 'annual_cost': 10979, 'stg': 3,
↪'utility_type': HENStreamType.cold_utility},
    {'hot': 'S', 'cold': 'C2', 'Q': 450, 'A': 50, 'annual_cost': 0, 'stg': 1,
↪'utility_type': HENStreamType.hot_utility}
]

streams = [
    {'name':'H2', 'temps': [423, 423, 330, 303], 'type': HENStreamType.hot},
    {'name':'H1', 'temps': [443, 435, 355, 333], 'type': HENStreamType.hot},
    {'name':'C1', 'temps': [408, 396, 326, 293], 'type': HENStreamType.cold},
    {'name':'C2', 'temps': [413, 413, 353, 353], 'type': HENStreamType.cold}
]
plot_obj = Plot.heat_exchanger_network(exchangers, streams,
    mark_temperatures_with_tooltips=True)
plot_obj.show()
```

By default tooltips are used to mark stream temperatures. We can disable those and add labels instead as seen below. They can be a bit crowded and for now you can just zoom in to decipher crowded labels (but we're working on that!)

```python
plot_obj = Plot.heat_exchanger_network(exchangers, streams,
    mark_temperatures_with_tooltips=False)
plot_obj.show()
```

In case a stream exchanges with multiple streams in the same stage, this is handled through a stage split. We also currently support describing modules for each exchanger that are added as tooltips to the area label on each exchanger. The example below demonstrates this functionality:

```python
exchangers = [
    {'hot': 'H1', 'cold': 'C2', 'Q': 2400, 'A': 355, 'annual_cost': 35727, 'stg': 2},
    {'hot': 'H2', 'cold': 'C2', 'Q': 1700, 'A': 159, 'annual_cost': 28358, 'stg': 2},


    {'hot': 'H1', 'cold': 'C2', 'Q': 1700, 'A': 159, 'annual_cost': 28358, 'stg': 3},
    {'hot': 'H1', 'cold': 'C1', 'Q': 667, 'A': 50, 'annual_cost': 10979, 'stg': 3,
↪'modules': {10: 1, 20: 2}},
```

(continues on next page)

```
    {'hot': 'H2', 'cold': 'C3', 'Q': 1700, 'A': 159, 'annual_cost': 28358, 'stg': 3},
    {'hot': 'H2', 'cold': 'C2', 'Q': 1700, 'A': 159, 'annual_cost': 28358, 'stg': 3,
→'modules': {10: 1, 20: 2}},
    {'hot': 'H3', 'cold': 'C2', 'Q': 1700, 'A': 159, 'annual_cost': 28358, 'stg': 3},

    {'hot': 'H2', 'cold': 'W', 'Q': 400, 'A': 50, 'annual_cost': 10979, 'stg': 3,
→'utility_type': HENStreamType.cold_utility},
    {'hot': 'S', 'cold': 'C2', 'Q': 450, 'A': 50, 'annual_cost': 0, 'stg': 1,
→'utility_type': HENStreamType.hot_utility}
]

streams = [
    {'name':'H3', 'temps': [423, 423, 330, 303], 'type': HENStreamType.hot},
    {'name':'H2', 'temps': [423, 423, 330, 303], 'type': HENStreamType.hot},
    {'name':'H1', 'temps': [443, 435, 355, 333], 'type': HENStreamType.hot},
    {'name':'C1', 'temps': [408, 396, 326, 293], 'type': HENStreamType.cold},
    {'name':'C2', 'temps': [413, 413, 353, 353], 'type': HENStreamType.cold},
    {'name':'C3', 'temps': [413, 413, 353, 353], 'type': HENStreamType.cold}
]
plot_obj = Plot.heat_exchanger_network(exchangers, streams,
    mark_temperatures_with_tooltips=True,
    mark_modules_with_tooltips=True,
    stage_width=2,
    y_stream_step=1)
plot_obj.show()
```

Plotting profile plots from the MEA example

> **Warning:** The following has not been tested recently and should be considered a work in progress.

The following examples demonstrate the resize, annotation and saving functionalities.

In the following example, we being by preparing a data frame from our flowsheet variables.

```
# Absorber CO2 Levels
from pandas import DataFrame
import os
tmp = fs.absorb.make_profile(t=0)
tmp = fs.regen.make_profile(t=0)

plot_dict = {'z':fs.absorb.profile_1['z'],
             'y1':fs.absorb.profile_1.y_vap_CO2*101325.0,
             'y2':fs.absorb.profile_1.P_star_CO2}
plot_data_frame = DataFrame(data=plot_dict)
```

We can then plot the data frame we just made, show it, resize it and save it.

```
absorber_co2_plot = Plot.profile(plot_data_frame,
                                 x = 'z',
                                 y = ['y1','y2'],
                                 title = 'Absorber CO2 Levels',
                                 xlab = 'Axial distance from top (m)',
                                 ylab = 'Partial Pressure CO2 (Pa)',
```

```
                                   legend = ['Bulk vapor','Equilibrium'])

absorber_co2_plot.show()
absorber_co2_plot.save('/home/jovyan/model_contrib/absorber_co2_plot.html')
assert(os.path.isfile('/home/jovyan/model_contrib/absorber_co2_plot.html'))
```

```
absorber_co2_plot.resize(height=400,width=600)
absorber_co2_plot.show()
absorber_co2_plot.save('/home/jovyan/model_contrib/absorber_co2_plot_resized.html')
assert(os.path.isfile('/home/jovyan/model_contrib/absorber_co2_plot_resized.html'))
```

The following demonstrates the annotate functionality by plotting a second plot from the same flowsheet.

```
from IPython.core.display import display,HTML
stripper_co2_plot = Plot.profile(plot_data_frame,
                                 x = 'z',
                                 y = ['y1','y2'],
                                 title = 'Stripper CO2 Levels',
                                 xlab = 'Axial distance from top (m)',
                                 ylab = 'Partial Pressure CO2 (Pa)',
                                 legend = ['Bulk vapor','Equilibrium'])
stripper_co2_plot.show()
stripper_co2_plot.save('/home/jovyan/model_contrib/stripper_co2_plot.html')
assert(os.path.isfile('/home/jovyan/model_contrib/stripper_co2_plot.html'))
```

We can then annotate the "Reboiler vapor" point as shown below:

```
stripper_co2_plot.annotate(rloc,rco2p,'Reboiler vapor')
stripper_co2_plot.show()
stripper_co2_plot.save('/home/jovyan/model_contrib/stripper_co2_plot_annotated.html')
```

> **Warning:** The visualization library is still in active development and we hope to improve on it in future releases.
> Please use its functionality at your own discretion.

4.7.2 Overview

The *idaes.vis* subpackage contains the framework and implementation of plots that are expected to be of general utility within the IDAES framework.

For users, an entry point is provided for IDAES classes to produce plots with the *idaes.vis.plotbase.PlotRegistry* singleton.

Plots will inherit from the interface in *idaes.vis.plotbase.PlotBase*, which provides some basic methods.

The current implementations all use the Python "bokeh" package, and can be found in *idaes.vis.bokeh_plots*.

4.8 Data Management Framework

4.8.1 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.

Contents

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
id   type desc    modified
```

(continues on next page)

```
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 a form that is accepted by the Pendulum parse function. The special token `..` is used to indicate date ranges, as in:

- `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
    }
  ],
  "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": "feee44ad365b6b1ec75c5621a0ad067371102854", "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
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
```

(continues on next page)

```
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 registerd 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
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"
                            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/
↪a27f98c24d1848eaba1b26e5ef87be88
  desc
    shoes.txt
  doc_id
    10
  id_
    dba0a5dc7d194040ac646bf18ab5eb50
  modified
    '2019-04-11 20:17:28'
```

(continues on next page)

```
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
    - 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:

```python
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
            │
            ├──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
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
```

4.8.2 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 Python *API* or command-line interfaces. There is work in progress on adding graphical interfaces for Jupyter Notebooks and stand-alone desktop apps.

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.
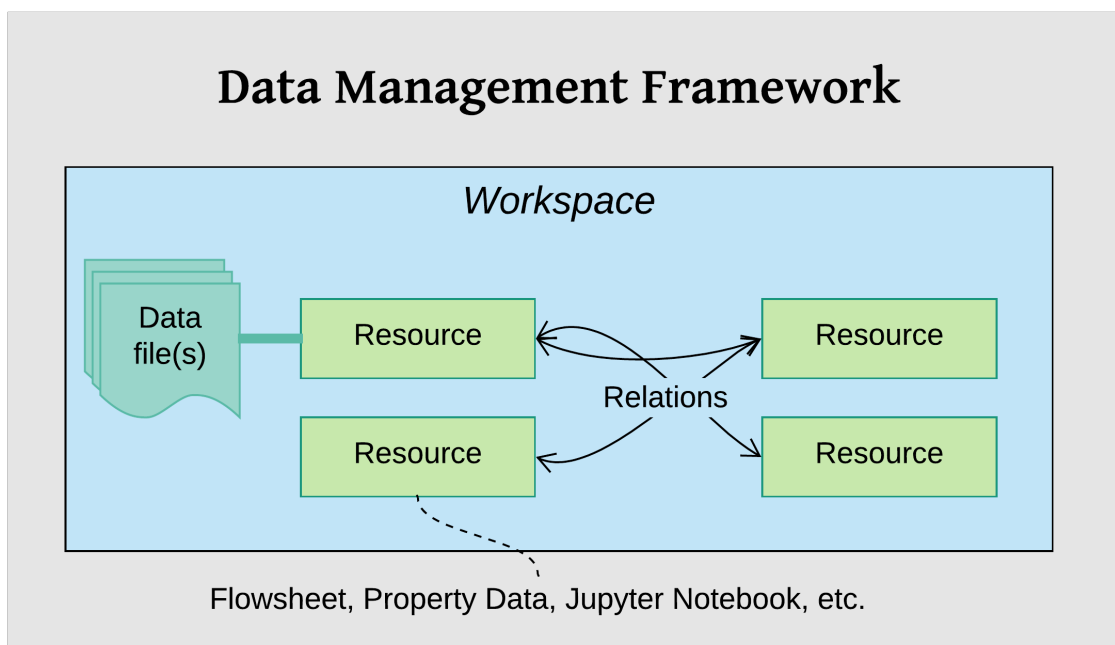
4.8.3 Configuration

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.

4.8.4 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_volume1d
%dmf help control_volume1d

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

# Get help on a class
from idaes.core.control_volume1d import ControlVolume1DBlock
%dmf help ControlVolume1DBlock

# Get help on a class (by name, no import)
%dmf help idaes.core.control_volume1d.ControlVolume1DBlock

# 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_volume1d.ControlVolume1DBlock()
%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).

4.8.5 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.

4.8.6 Reference

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

4.9 Data Driven Machine Learning

4.9.1 ALAMOPY : ALAMO Python

ALAMOPY.ALAMO Options

This page lists in more detail the ALAMOPY options and the relation of ALAMO and ALAMOPY.

Contents

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 = x1, exp(x1), log(x1), sin(x1), cos(x1) * **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 = x1*x2, (x1*x2)^2, (x1*x2)^3

- Trinomial functions = (x1*x2*x3), (x1*x2*x3)^2, (x1*x2*x3)^3

- **ratiopower**: list of ratio powers. For example

```
ratiopower = (1,2,3)
```

This results in basis functions = (x1/x2), (x1/x2)^2, (x1/x2)^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 beused for model building (1-8)

    - 1. **BIC**: Bayesian infromation 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 eviation 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, **kargs)
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 xlabels 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).

```
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 xlabels assigned to the input parameters.

```
addCustomConstraints(custom_constraint_list, **kargs)
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 **z** are an unknown function **f** of the system inputs **x**. The software identifies a function **f**, i.e., a relationship between the inputs and outputs of the system, that best matches data (pairs of **x** and corresponding **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, **kargs)
```

where ****kargs** 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, sinfcns, linfcns, intercept. These are '0-1' options to activate these functions

- monomialpower, multi2power, multi3power, ratiopower. List of terms to be used in the respective basis functions

- modeler - integer 1-7 determines the choice of fitness metrice

- 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 controlif 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 fille to faciliate advanced user access in an automated fashion

ALAMOPY Output

There are mutliple 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)
```
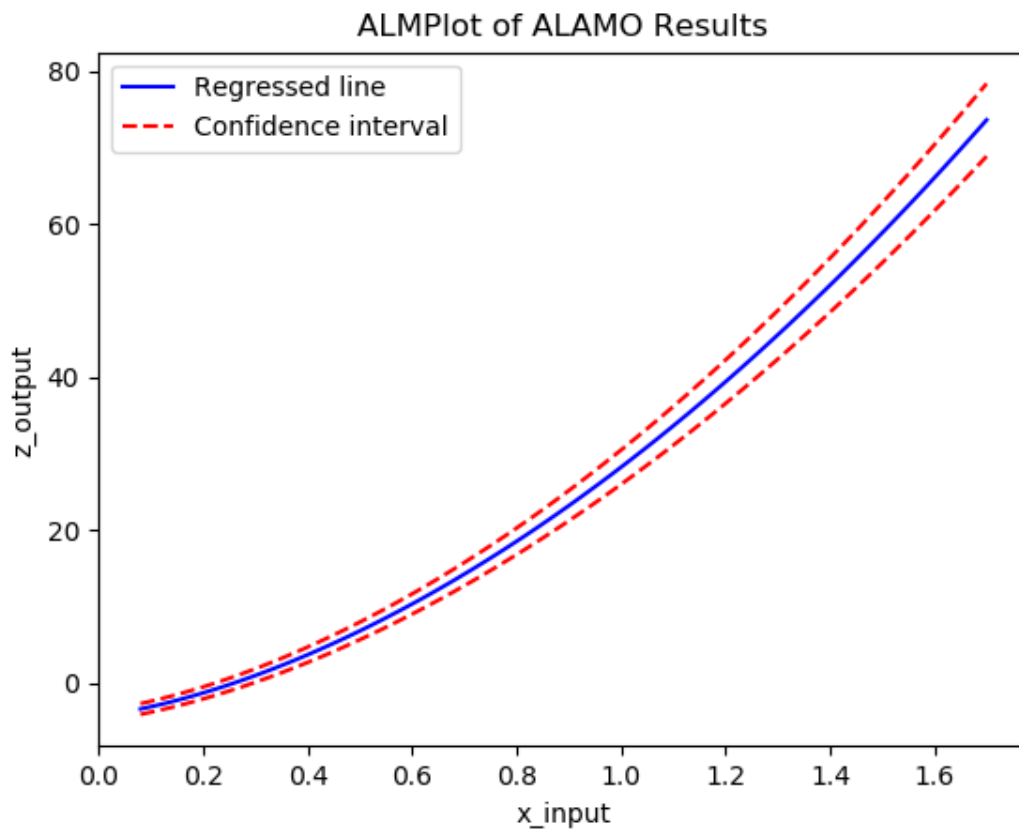

Advanced Regression Capabilities

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 xlabels 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 xlabels assigned to the input parameters.

```
addCustomConstraints(custom_constraint_list, **kargs)
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 notebok

4.9.2 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 availible 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 stoiciometry 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 defiend as a list of list, where reactants and products are defined as negative and positive integers , respectively, according to their stoichiometric coeffecints. 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 availible 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 availible, 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 termpeature 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 resutls

- zscale - linear scaling of observed responses between -1 and 1

- ascale - linear scaling of activities between -1 and 1

- hide_output - surpress 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().

4.9.3 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.

```python
import helmet.Helmet as Helmet
```

The methods available in helmet.Helmet peform the necessary steps of the regression properties.

1. **initialize(**kargs)**

   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 orgnization at http://www.iapws.org/95data.html for IAPWS Formulation 1995 for Thermodynamic Properties of Odrinary Water Substance for General and Scientific Use.

> **Warning:** The ddm-learning library is still in active development and we hope to improve on it in future releases. Please use its functionality at your own discretion.

4.9.4 Overview

The Data Driven Machine Language (ddm-learning) repository contains regression tools for the development of property models for kinetics and thermodynamics of a system. The provided tools include both ALAMOpy and RIPE that can access ALAMO and other solvers through the Python API. Examples for both tool are provided.

4.9.5 Contributing

By contributing to this software you are agreeing to all the terms laid out in the *License* and *Copyright*.

4.10 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.).

4.10.1 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:

```python
import idaes
print(idaes.__version__)
# prints a version like "1.2.3"
```

4.10.2 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:

```python
>>> 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:

```python
>>> 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.package_version = <idaes.ver.Version object>
```
   Package's version as an object

```
idaes.ver.__version__ = '1.4.4'
```
   Package's version as a simple string

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.

4.11 Tutorials

The tutorials linked below are Jupyter Notebooks, which create and run IDAES models. They provide a thorough introduction to the capabilities of the IDAES PSE framework. They were originally presented at a stakeholder meeting in May of 2019. Each tutorial presents the creation of models, etc., as a series of steps with extensive context and

information. Each tutorial builds on information from the prior one, so it is recommended that the new user view them in order.

If you want to run these Jupyter notebooks yourself, you need to download the source code for the IDAES toolkit and then navigate to *examples/workshops* and its subdirectories. You would load a given tutorial with the command:

```
jupyter notebook <notebook-file-name.ipynb>
```

Then, in the Jupyter interface, you could select "Run all" to see the tutorial executed in front of you.

4.12 JupyterLab

4.12.1 Flowsheet Viewer
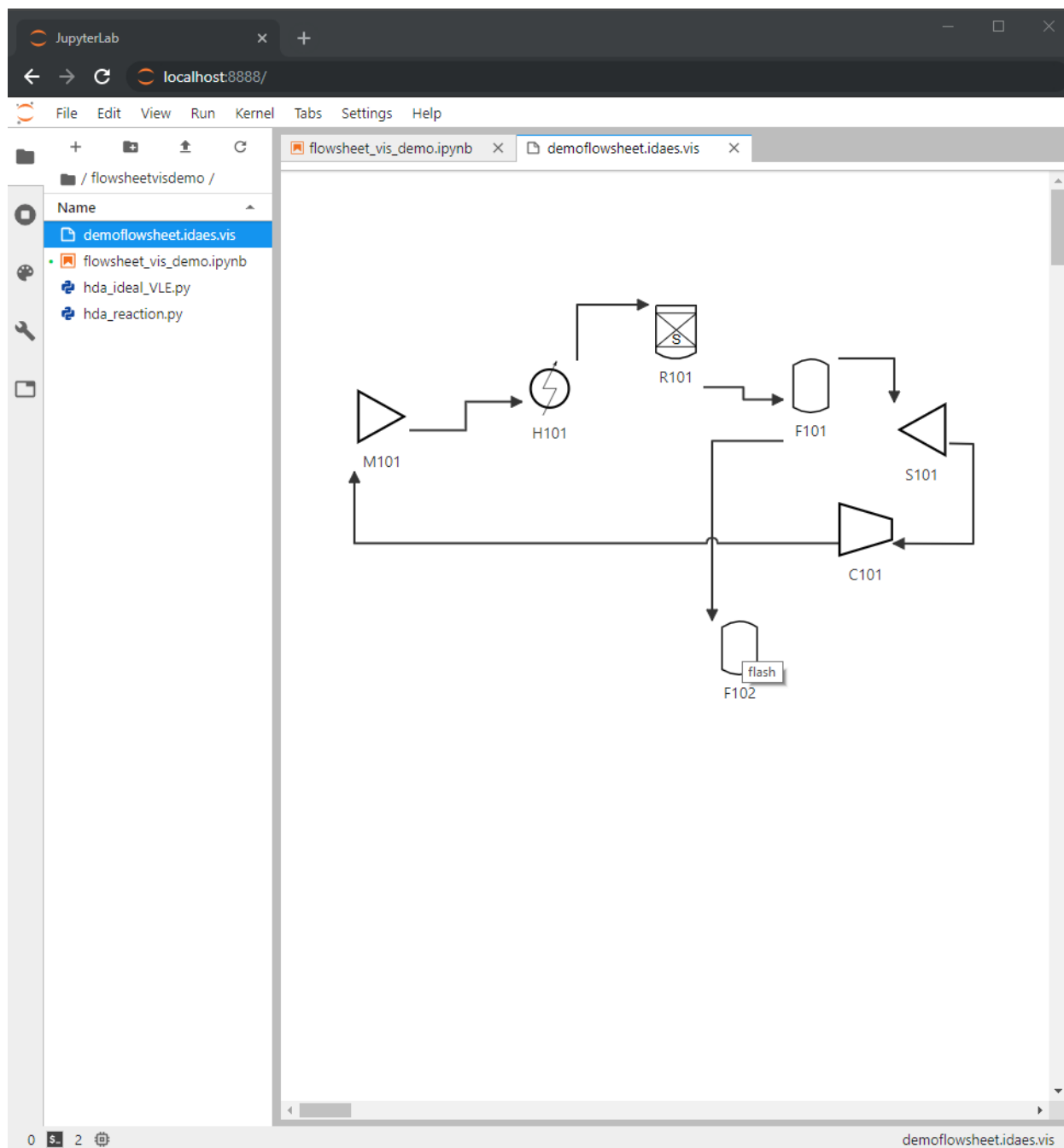
Note: The flowsheet viewer requires the use of JupyterLab.

Overview

Flowsheets may be serialized to ".idaes.vis" files which, in conjunction with the flowsheet viewer, produce interactive visual representations of flowsheets. The resultant flowsheet diagrams can be rearranged and saved.

Instructions

1. *Ensure that the latest IDAES is installed.*

2. Install JupyterLab. If you are not using Conda environments, use the *pip install* instructions.

Installation

From your terminal, call the following commands to build and install the extension:

```
cd <repository>/ui/modelvis/idaes-model-vis
npm install # takes a few minutes
npm run build
jupyter labextension link . # takes a few minutes
```

Usage

1. Launch JupyterLab (run `jupyter lab` from a folder you wish to work out of).

2. Create a new Python 3 notebook from the JupyterLab Launcher, or select a preexisting notebook from the directory navigation pane on the left. An example (depicted) is located in idaes-pse/ui/modelvis/flowsheetdemo.
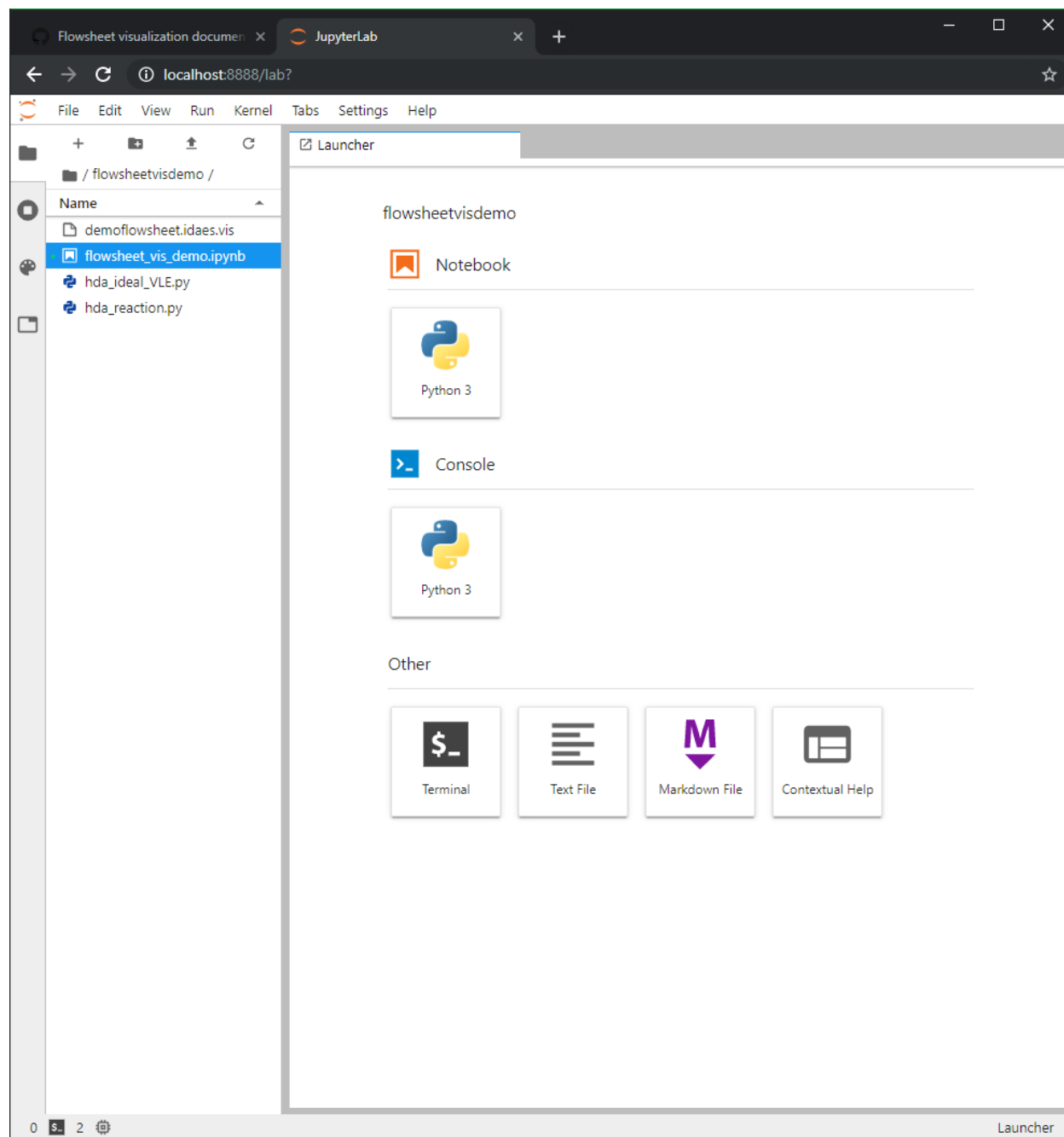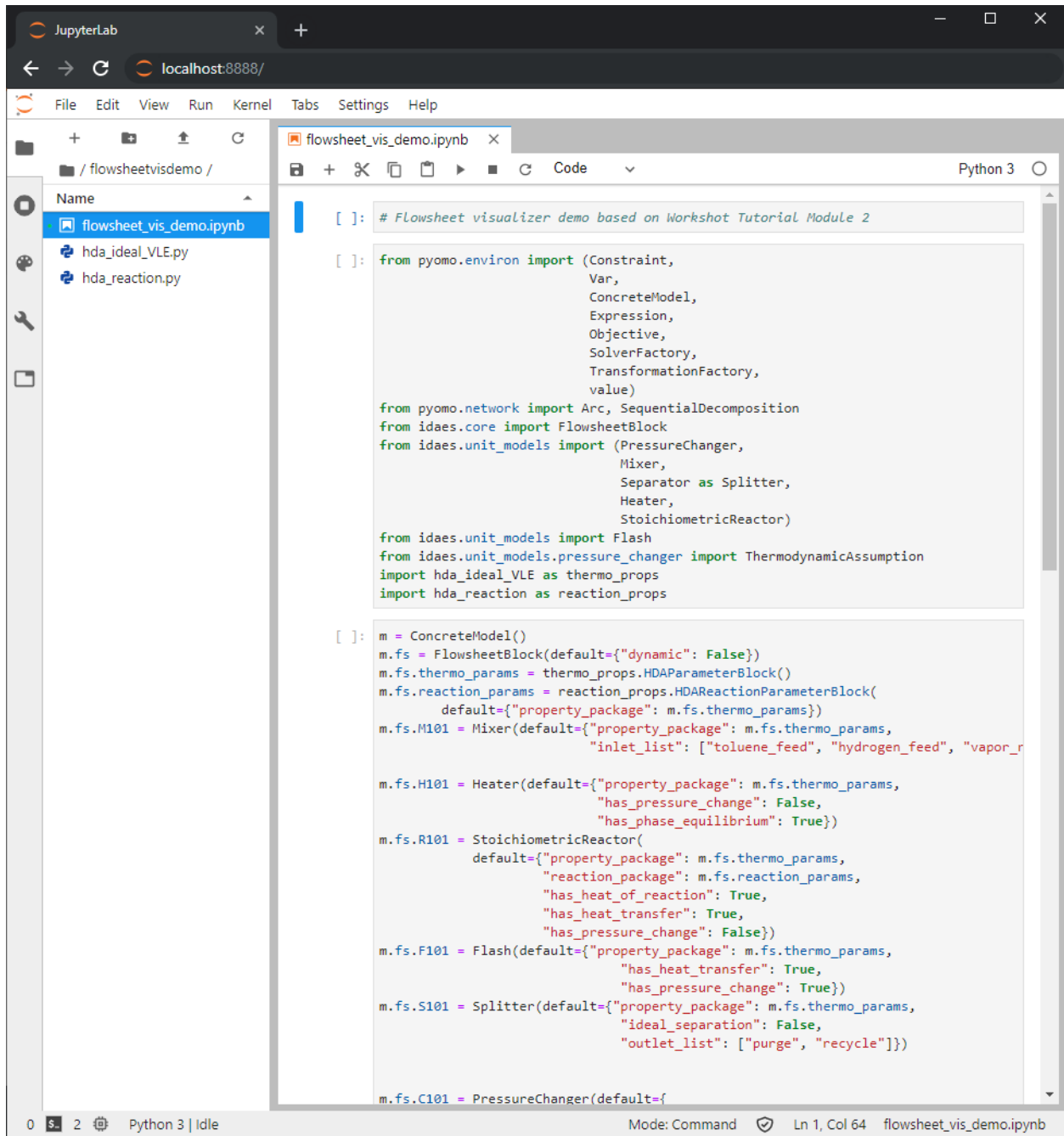
3. In the notebook, construct a flowsheet as usual (add unit models, set connections, etc.).

4. Run the *.serialize()* method from the flowsheet, as below:

```
m.fs.serialize('myflowsheetname')
```

A `.idaes.vis` file should be created with the chosen filename (e.g. `myflowsheetname.idaes.vis`), and become visible in the JupyterLab file browser. If there is an existing file with the same name, you must either choose a different filename or add the additional optional argument `overwrite=True` (in which case the file will be overwritten).
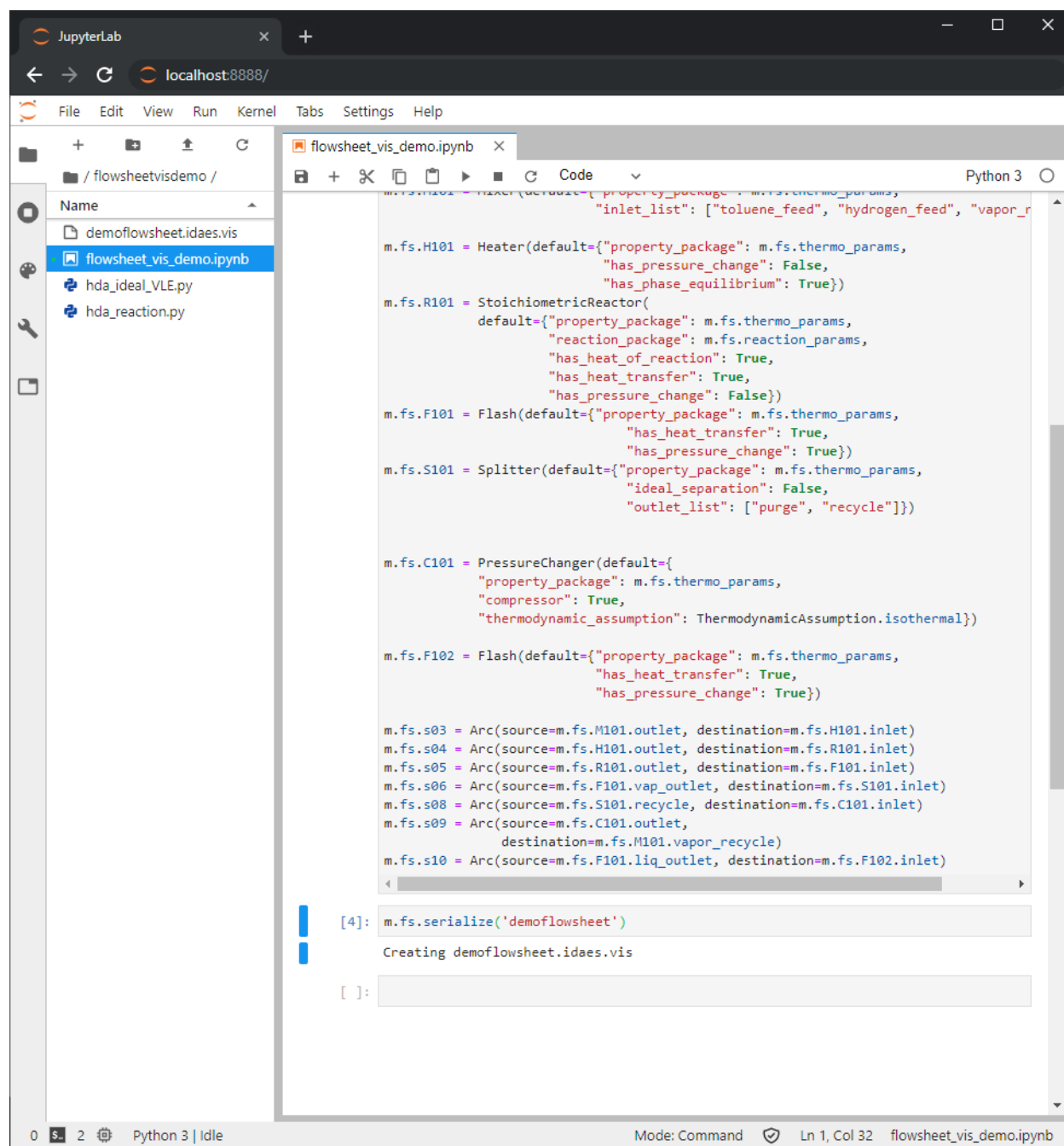
8. Open the created .idaes.vis file in JupyterLab. A tab should open and display a graph representation of the serialized flowsheet; the components are tiled diagonally by default, and can be rearranged to your liking.
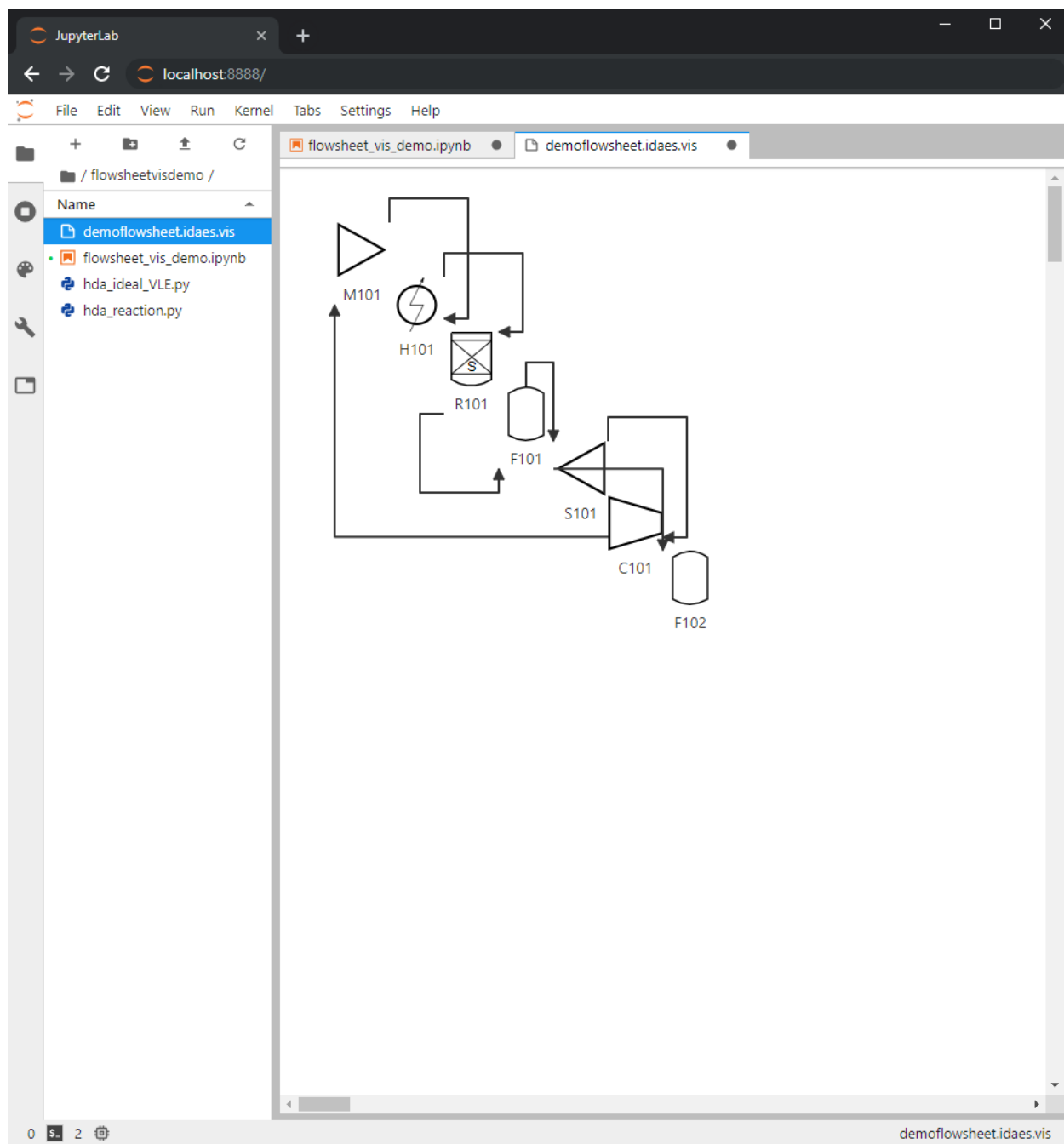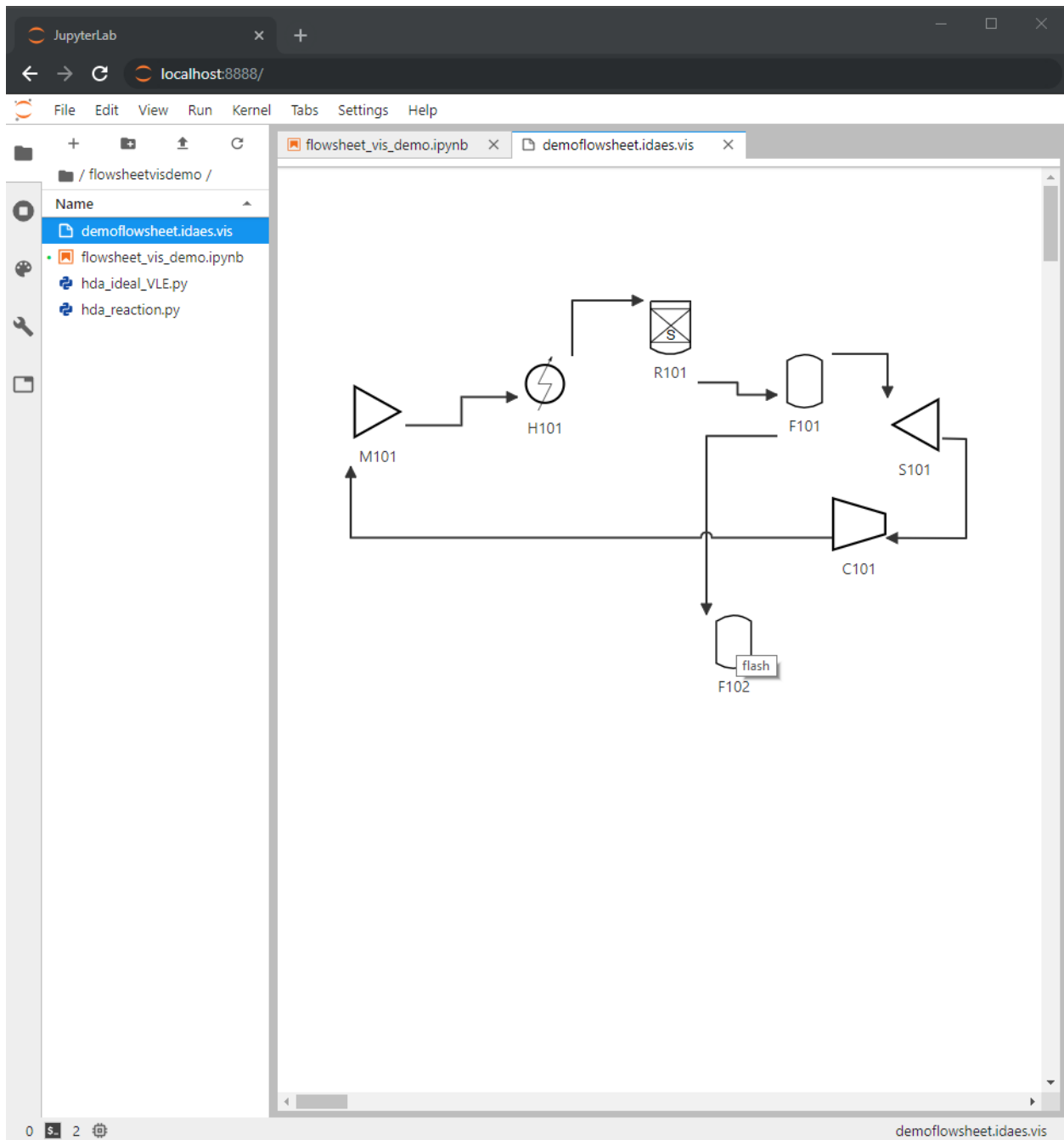
9. The layout of the graph can be saved into the serialized file by using JupyterLab's `File->Save` menu item (or the equivalent hotkey Ctrl+s/Command+s). Autosaving can also be configured by using JupyterLab's `Settings->Advanced Settings Editor` option under `Document Manager`.
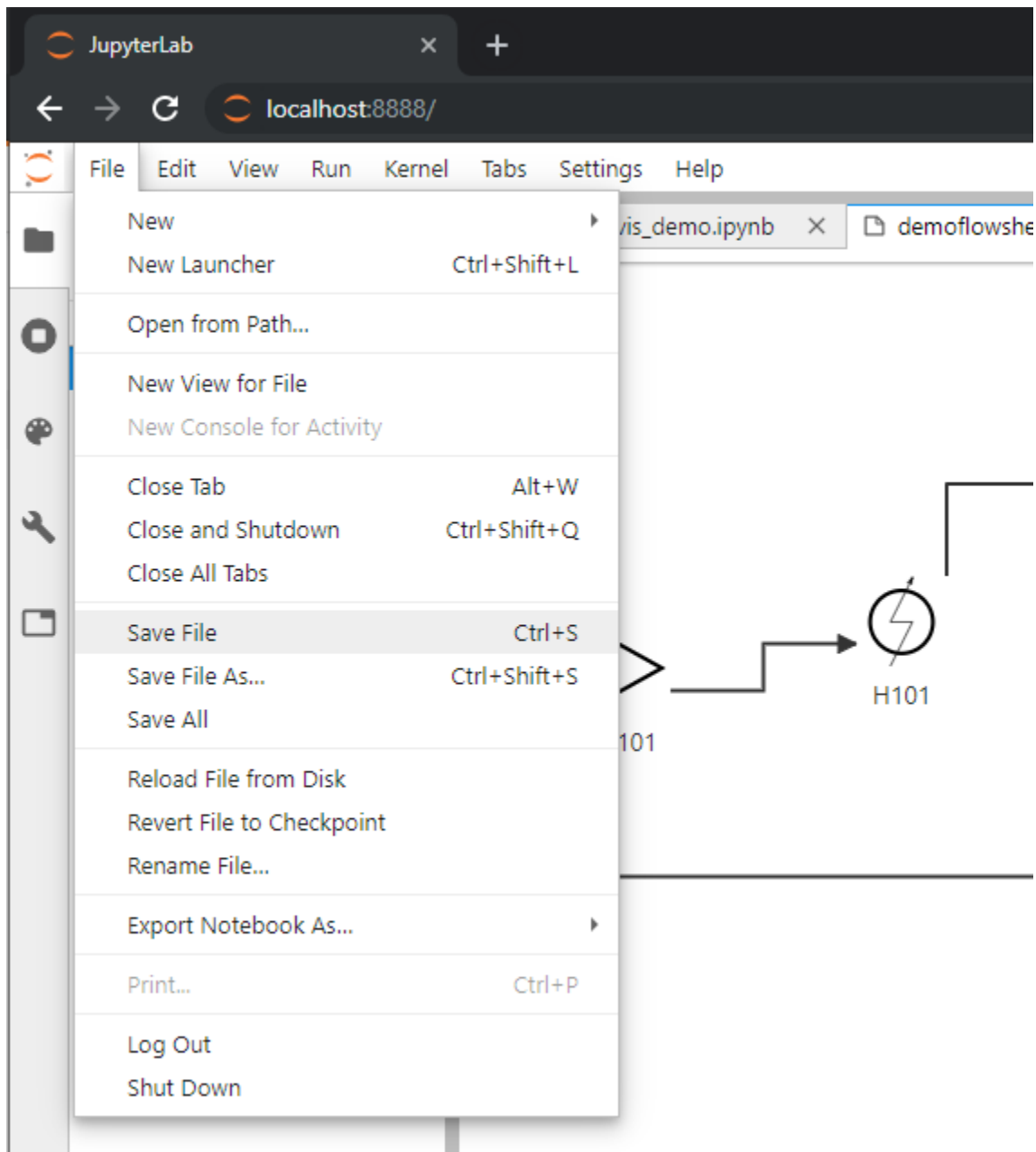
Miscellany

- Unit model icons can be rotated by right-clicking on the icon.

- Connections paths between unit models can be moved by clicking on the link, then dragging the link vertex that appears. Double-click the vertex to remove it.

- JupyterLab tabs can be rearranged by clicking and dragging the top of the tab, and resized by dragging the borders.

Developer notes

Rebuilding

After making changes to the TypeScript, rebuild the extension and reinstall it into JupyterLab:

```
npm run build
jupyter lab build
```

4.12.2 Overview

JupyterLab is an interface for working with Jupyter Notebooks simultaneously with files, plots, terminals, and other customizable interfaces.

4.12.3 Flowsheet Viewer

The Flowsheet Viewer is a JupyterLab extension that displays flowsheets serialized with `flowsheet.serialize()`.
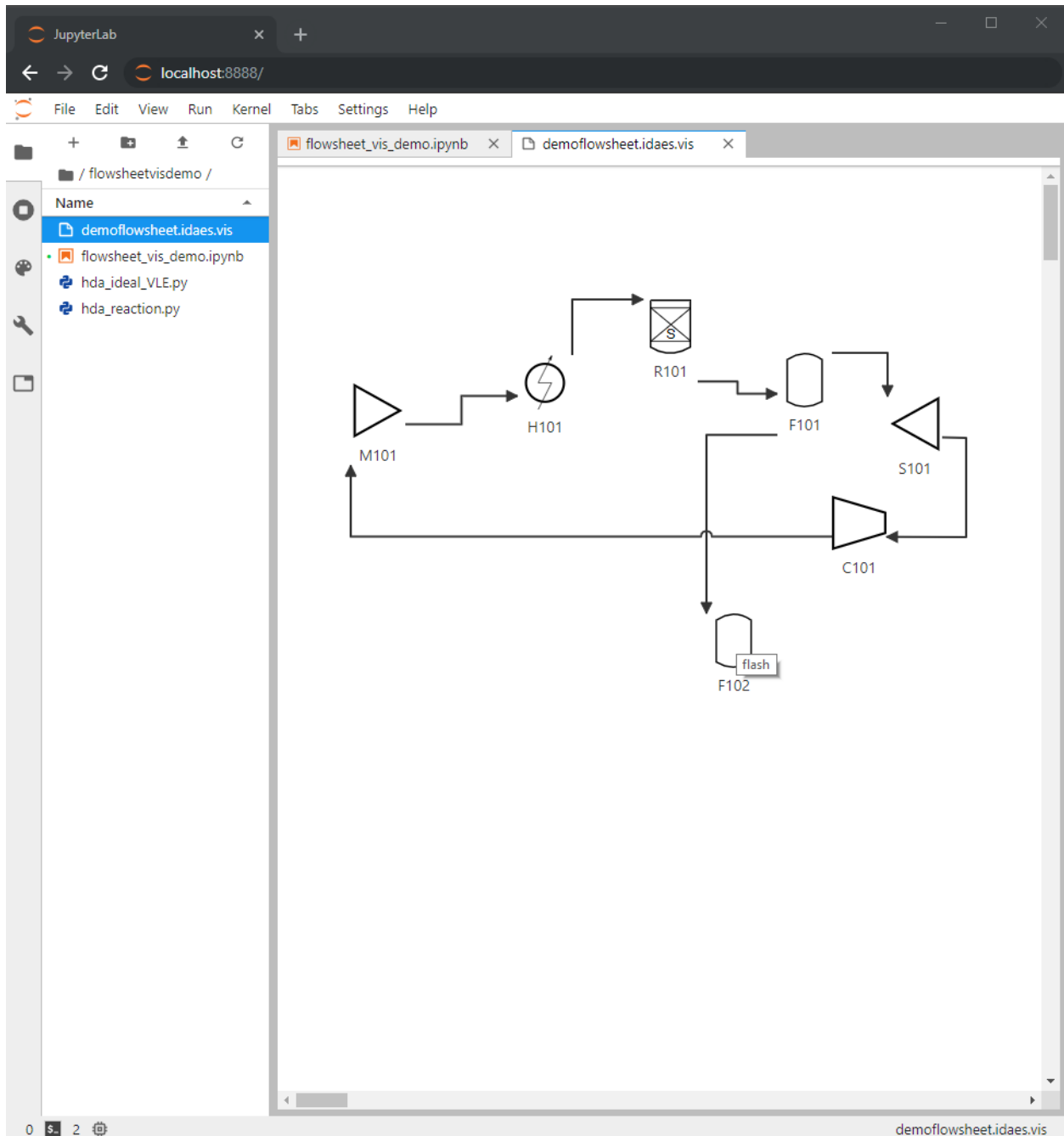
4.13 Developer Documentation

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.

4.13.1 Developer Contents

Developer introductory material

This section gives a high-level introduction for collaborative software development on the IDAES project. It serves as background for understanding the collaborative development procedures.

Please refer to the IDAES contributor guide *for specifics on writing, testing, and documenting code for the IDAES project.*

There are many more useful things to learn about git and Github. For more information, please refer to the excellent Atlassian Github tutorials and the online Git documentation and Github help.

Terminology

Git A "version control system", for keeping track of changes in a set of files

Github A hosting service for Git repositories that adds many other features that are useful for collaborative software development.

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*.

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).

Branches

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 "master" 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 master branch in the main (e.g., idaes/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.

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 *ideas-pse* 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. Setup: Prepare your local system for collaborative development
2. Initiate: Notify collaborators of intent to make some changes
3. Develop: Make local changes
4. Collaborate: Push the changes to Github, get feedback and merge

The rest of this page describes the what and how of each of these phases.

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.
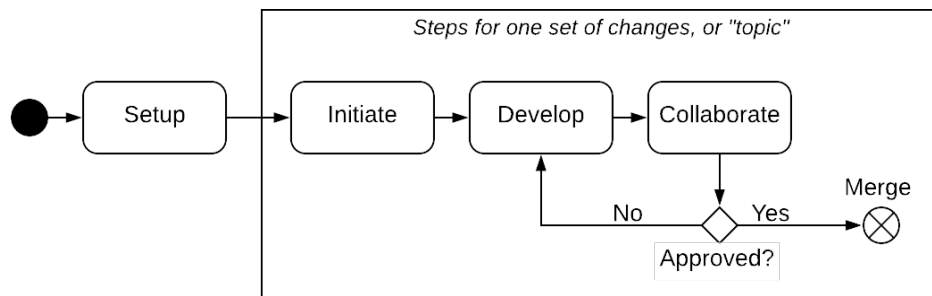
Fig. 4: Figure 1. Overview of software development workflow

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 you want to put the source code and run the command:

Fig. 5: Figure 2. Screenshot showing where to click to fork the Github repo

```
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.

Chapter 4. Contents
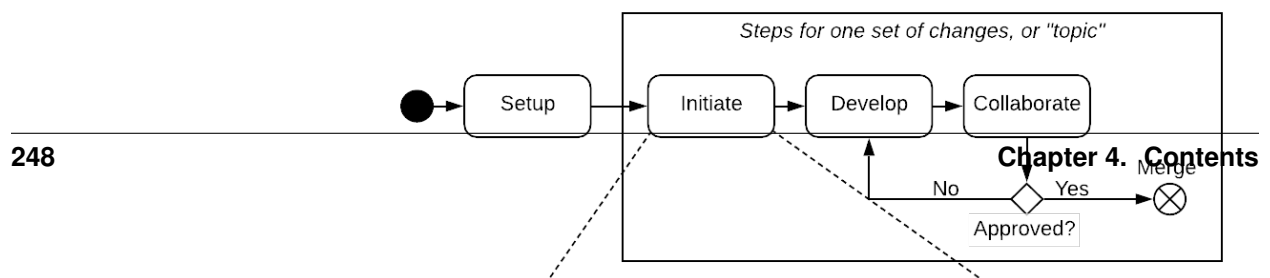
Create an issue on Github

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.

Create a branch on your fork

It is certainly possible to do your work on your fork in the "master" 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 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.
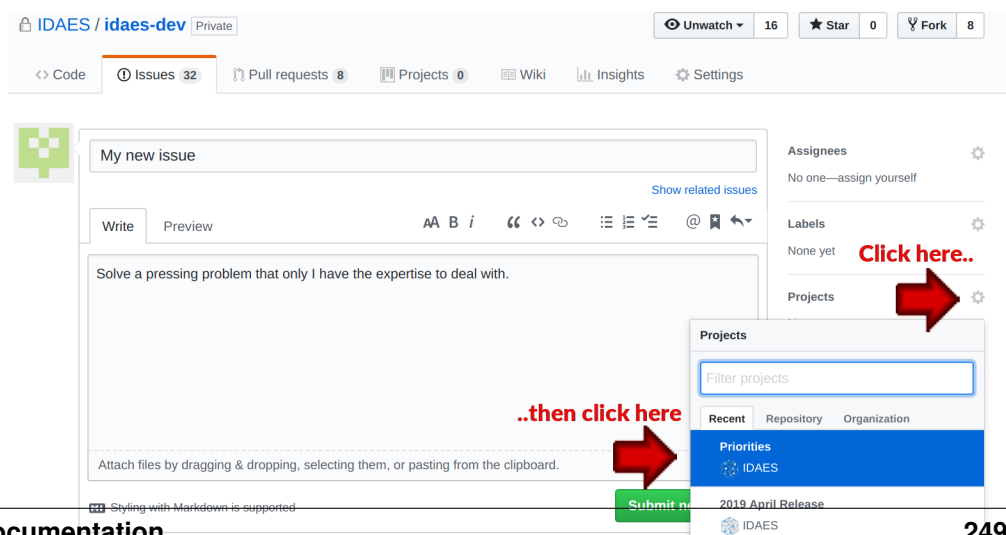
Fig. 7: Figure 4. Screenshot for creating an issue on Github

One con-
vention for
the name
that can
be help-
ful is to
include
the Is-
sue num-
ber 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'
```

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 in-
formation. Remem-
ber to link to the is-
sue you created ear-
lier.

Depending on the
Github plan, there
may be a pull-down
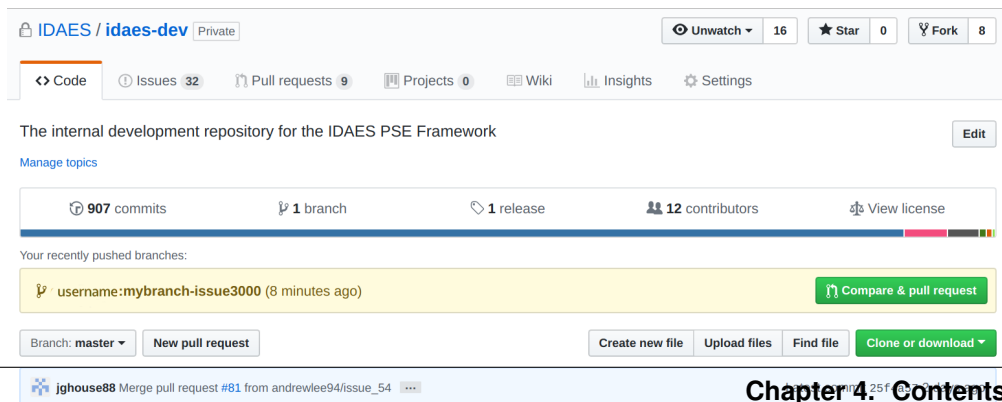
Fig. 8: Figure 5. Screenshot for starting a Pull Request on Github

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.

---

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
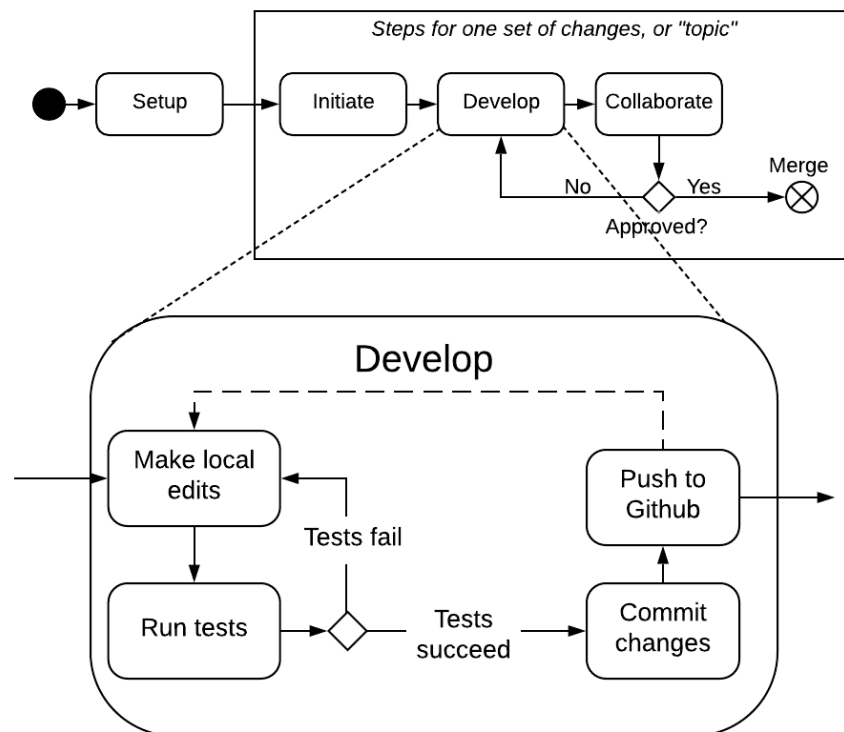
Fig. 9: Figure 6. Software development workflow

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 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"
[master 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/master
```

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):

---

[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.

```
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.
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 master 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

To request review of a pull request, navigate to the pull request in the main (e.g., "idaes-dev") repository and select some names in the "Reviewers"

Fig. 10: Figure 7. Collaborate phase workflow
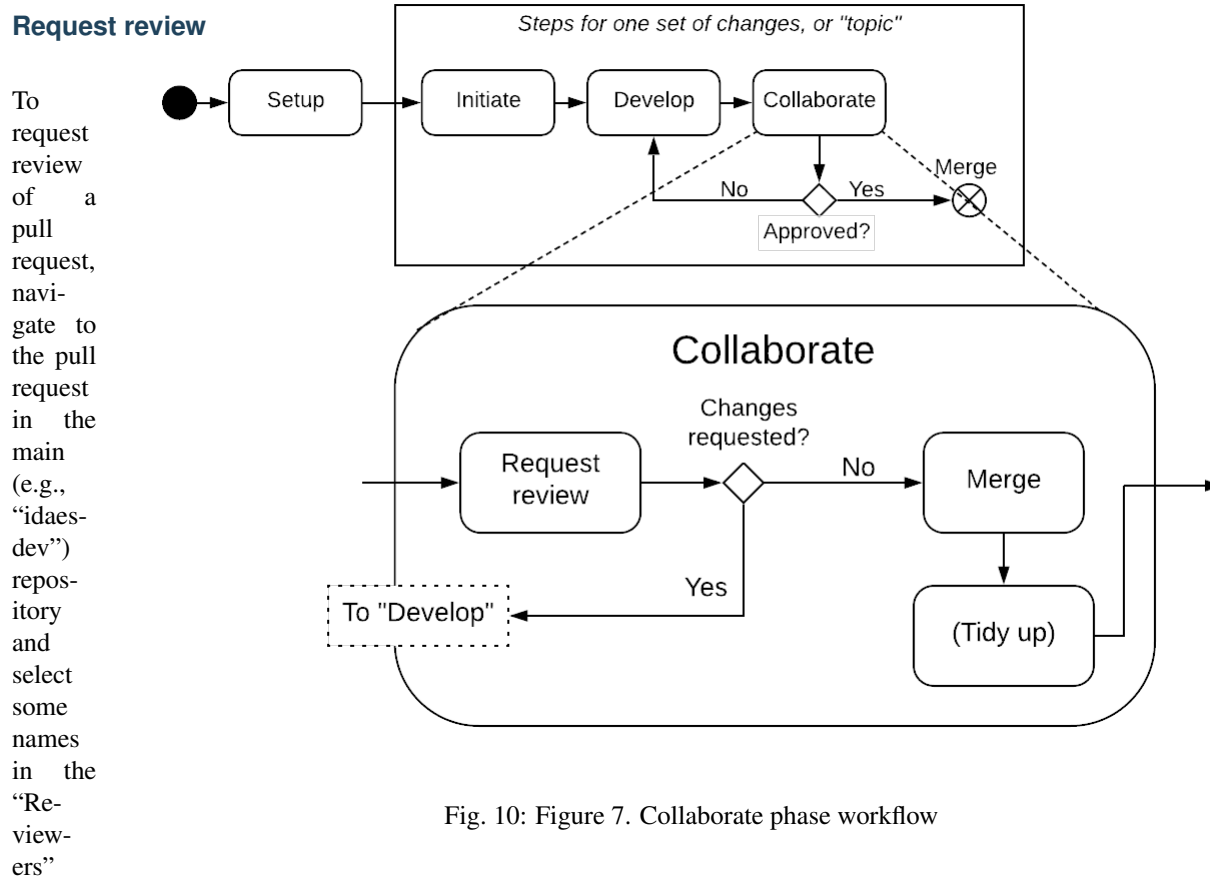
pull-
down
on the
right-hand side. You need to have two approving reviews. The reviewers should get an email, but you can also "@"
people in a comment in the pull request to give them a little extra nudge.

See the full *code review* procedure for more details.

Make changes

You need to keep track of the comments and reviews, and make changes accordingly. Think of a pull request as a
discussion. Normally, 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 reviewed use the normal *Develop* workflow.

Shared forks

Other developers can also make changes in your fork. All they need to do is `git clone` your fork (not the main
repository), 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 example, if Jack wants to make some edits on Rose's fork, on a topic branch called "changes-issue51" he could
do the following:

```
$ git clone https://github.com/rose/idaes-dev # clone Rose's fork
$ git checkout changes-issue51  # checkout the topic branch
$ 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
```

Hopefully it also is obvious that developers working this way have less safeguards for overwriting each other's work,
and thus should make an effort to communicate clearly and in a timely manner.

Merge

Once all the tests pass and you have enough approving reviews, it's time to merge the code! This is the easy part: go
to the bottom of the Pull Request and hit the big green "merge" button.

Before you close the laptop and go down to the pub, you should tidy up. First, delete your local branch (you can also
delete that branch on Github):

```
git checkout master # switch back to master branch
git branch -d mychanges-issue3000
```

Next, you should make sure your master reflects the current state of the main master branch, i.e. go back and *synchronize with the upstream remote*, i.e. run `git pull`.

Now you can go and enjoy a tasty beverage. Cheers!

Testing

Testing is essential to the process of creating software. "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 contributor guide.

There are different kinds of tests: functional, acceptance, performance, usability. We will primarily concern ourselves with *functional* testing 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

Fig. 11: Figure 1. Conceptual space of functional testing

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.

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`):

```python
# 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:

```python
# 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 rerun 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 master branch – if tests fail, then no merge is allowed. Following this procedure, it is not possible for the master 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
```

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*. The documentation for the *apps/* directory is not (currently) being built automatically.

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:

```python
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")
    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).

Note: On Windows use the provided `make.bat` command (in the `docs` dir) rather than *make* in the below examples.

To build the documentation locally, there is a "Makefile" in the *docs/* directory:

```
cd docs
make allclean
make all
```

The above commands will do a completely clean build to create HTML output. They will also attempt to execute the tutorials. During development, more specific Makefile targets may save time:

make html Only build the HTML from the existing *.rst* files and generated API docs. Does not rebuild the tutorials or regenerate the API docs.

make apidoc Just regenerate API documentation source from the Python code. This does not change the HTML output.

make tutorials Generate HTML web pages from the Jupyter Notebook tutorials

Like any other Makefile, you can use these targets together. So, if you are editing source code and want to preview the generated documentation, you should run: `make apidoc html`. This will regenerate *.rst* files from the source code, then build those files together with hand-edited files into the HTML output.

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.

4.14 idaes

4.14.1 idaes package

__init__.py for idaes module

Set up logging for the idaes module, and import plugins.

Subpackages

idaes.commands package

Submodules

idaes.commands.base module

Base command for 'idaes' commandline script

idaes.commands.data_directory module

Commandline Utilities for Managing the IDAES Data Directory

idaes.commands.extensions module

Commandline Utilities for Managing the IDAES Data Directory

idaes.core package

Subpackages

idaes.core.util package

Subpackages

idaes.core.util.convergence package

Submodules

idaes.core.util.convergence.convergence module

This module is a command-line script for executing convergence evaluation testing on IDAES models.

Convergence evaluation testing is used to verify reliable convergence of a model over a range of conditions for inputs and parameters. The developer of the test must create a ConvergenceEvaluation class prior to executing any convergence testing (see convergence_base.py for documentation).

Convergence evaluation testing is a two step process. In the first step, a json file is created that contains a set of points sampled from the provided inputs. This step only needs to be done once - up front. The second step, which should be executed any time there is a major code change that could impact the model, takes that set of sampled points and solves the model at each of the points, collecting convergence statistics (success/failure, iterations, and solution time).

To find help on convergence.py:

```
$ python convergence.py --help
```

You will see that there are some subcommands. To find help on a particular subcommand:

```
$ python convergence.py  <subcommand> --help
```

To create a sample file, you can use a command-line like the following (this should be done once by the model developer for a few different sample sizes):

```
$ python ../../../core/util/convergence/convergence.py create-sample-file
     -s PressureChanger-10.json
     -N 10 --seed=42
     -e idaes.models.convergence.pressure_changer.
         pressure_changer_conv_eval.PressureChangerConvergenceEvaluation
```

More commonly, to run the convergence evaluation:

```
$ python ../../../core/util/convergence/convergence.py run-eval
     -s PressureChanger-10.json
```

Note that the convergence evaluation can also be run in parallel if you have installed MPI and mpi4py using a command line like the following:

```
$ mpirun -np 4 python ../../../core/util/convergence/convergence.py run-eval
     -s PressureChanger-10.json
```

idaes.core.util.convergence.convergence_base module

This module provides the base classes and methods for running convergence evaluations on IDAES models. The convergence evaluation runs a given model over a set of sample points to ensure reliable convergence over the parameter space.

The module requires the user to provide:

> • a set of inputs along with their lower bound, upper bound, mean,

and standard deviation.

> • an initialized Pyomo model
>
> • a Pyomo solver with appropriate options

The module executes convergence evaluation in two steps. In the first step, a json file is created that containsa set of points sampled from the provided inputs. This step only needs to be done once - up front. The second step, which should be executed any time there is a major code change that could impact the model, takes that set of sampled points and solves the model at each of the points, collecting convergence statistics (success/failure, iterations, and solution time).

This can be used as a tool to evaluate model convergence reliability over the defined input space, or to verify that convergence performance is not decreasing with framework and/or model changes.

In order to write a convergence evaluation for your model, you must inherit a class from ConvergenceEvaluation, and implement three methods:

- **get_specification: This method should create and return a** ConvergenceEvaluationSpecification object. There are methods on ConvergenceEvaluationSpecification to add inputs. These inputs contain a string that identifies a Pyomo Param or Var object, the lower and upper bounds, and the mean and standard deviation to be used for sampling. When samples are generated, they are drawn from a normal distribution, and then truncated by the lower or upper bounds.

- **get_initialized_model: This method should create and return a Pyomo model** object that is already initialized and ready to be solved. This model will be modified according to the sampled inputs, and then it will be solved.

- **get_solver: This method should return an instance of the Pyomo solver that** will be used for the analysis.

There are methods to create the sample points file (on ConvergenceEvaluationSpecification), to run a convergence evaluation (run_convergence_evaluation), and print the results in table form (print_convergence_statistics).

However, this package can also be executed using the command-line interface. See the documentation in convergence.py for more information.

idaes.core.util.convergence.convergence_base.**print_convergence_statistics**(*inputs*, *results*, *s*)

> Print the statistics returned from run_convergence_evaluation in a set of tables

> **Parameters**

>> - **inputs** (*dict*) – The inputs dictionary returned by run_convergence_evaluation

>> - **results** (*dict*) – The results dictionary returned by run_convergence_evaluation

> **Returns**

> **Return type** N/A

idaes.core.util.convergence.convergence_base.**run_convergence_evaluation**(*sample_file_dict*, *conv_eval*)

> Run convergence evaluation and generate the statistics based on information in the sample_file.

> **Parameters**

>> - **sample_file_dict** (*dict*) – Dictionary created by ConvergenceEvaluationSpecification that contains the input and sample point information

>> - **conv_eval** (*ConvergenceEvaluation*) – The ConvergenceEvaluation object that should be used

> **Returns**

> **Return type** N/A

idaes.core.util.convergence.convergence_base.**save_results_to_dmf**(*dmf*, *inputs*, *results*, *stats*)

> Save results of run, along with stats, to DMF.

> **Parameters**

>> - **dmf** (*DMF*) – Data management framework object

>> - **inputs** (*dict*) – Run inputs

- **results** (*dict*) – Run results

- **stats** (*Stats*) – Calculated result statistics

Returns None

idaes.core.util.convergence.convergence_base.**write_sample_file**(*eval_spec*, *file-
name*, *conver-
gence_evaluation_class_str*,
n_points,
seed=None)

Samples the space of the inputs defined in the eval_spec, and creates a json file with all the points to be used in executing a convergence evaluation

Parameters

- **filename** (*str*) – The filename for the json file that will be created containing all the points to be run

- **eval_spec** (*ConvergenceEvaluationSpecification*) – The convergence evaluation specification object that we would like to sample

- **convergence_evaluation_class_str** (*str*) – Python string that identifies the convergence evaluation class for this specific evaluation. This is usually in the form of module.class_name.

- **n_points** (*int*) – The total number of points that should be created

- **seed** (*int or None*) – The seed to be used when generating samples. If set to None, then the seed is not set

Returns

Return type N/A

idaes.core.util.convergence.mpi_utils module

Submodules

idaes.core.util.config module

This module contains utility functions useful for validating arguments to IDAES modeling classes. These functions are primarily designed to be used as the *domain* argument in ConfigBlocks.

idaes.core.util.config.**is_physical_parameter_block**(*val*)
Domain validator for property package attributes

Parameters **val** – value to be checked

Returns ConfigurationError if val is not an instance of PhysicalParameterBlock or useDefault

idaes.core.util.config.**is_port**(*arg*)
Domain validator for ports

Parameters **arg** – argument to be checked as a Port

Returns Port object or Exception

idaes.core.util.config.**is_reaction_parameter_block**(*val*)
Domain validator for reaction package attributes

Parameters **val** – value to be checked

> **Returns** ConfigurationError if val is not an instance of ReactionParameterBlock

`idaes.core.util.config.`**`is_state_block`**(*val*)

> Domain validator for state block as an argument
>
> > **Parameters** **`val`** – value to be checked
> >
> > **Returns** ConfigurationError if val is not an instance of StateBlock or None

`idaes.core.util.config.`**`is_time_domain`**(*arg*)

> Domain validator for time domains
>
> > **Parameters**
> >
> > - **`arg`** – argument to be checked as a time domain (i.e. Set or
> > - **`ContinuousSet)`** –
> >
> > **Returns** Set, ContinuousSet or Exception

`idaes.core.util.config.`**`is_transformation_method`**(*arg*)

> Domain validator for transformation methods
>
> > **Parameters** **`arg`** – argument to be checked for membership in recognized strings
> >
> > **Returns** Recognised string or Exception

`idaes.core.util.config.`**`is_transformation_scheme`**(*arg*)

> Domain validator for transformation scheme
>
> > **Parameters** **`arg`** – argument to be checked for membership in recognized strings
> >
> > **Returns** Recognised string or Exception

`idaes.core.util.config.`**`list_of_floats`**(*arg*)

> Domain validator for lists of floats
>
> > **Parameters** **`arg`** – argument to be cast to list of floats and validated
> >
> > **Returns** List of strings

`idaes.core.util.config.`**`list_of_strings`**(*arg*)

> Domain validator for lists of strings
>
> > **Parameters** **`arg`** – argument to be cast to list of strings and validated
> >
> > **Returns** List of strings

idaes.core.util.exceptions module

This module contains custom IDAES exceptions.

exception `idaes.core.util.exceptions.`**`BalanceTypeNotSupportedError`**

> IDAES exception to be used when a control volumedoes not support a given type of balance equation.

exception `idaes.core.util.exceptions.`**`BurntToast`**

> General exception for when something breaks badly in the core.

exception `idaes.core.util.exceptions.`**`ConfigurationError`**

> IDAES exception to be used when configuration arguments are incorrect or inconsistent.

exception `idaes.core.util.exceptions.`**`DynamicError`**

> IDAES exception for cases where settings associated with dynamic models are incorrect.

exception `idaes.core.util.exceptions.`**`PropertyNotSupportedError`**
   IDAES exception for cases when a models calls for a property which is not supported by the chosen property package.

   Needs to inherit from AttributeError for Pyomo interactions.

exception `idaes.core.util.exceptions.`**`PropertyPackageError`**
   IDAES exception for generic errors arising from property packages.

   Needs to inherit from AttributeError for Pyomo interactions.

idaes.core.util.expr_doc module

class `idaes.core.util.expr_doc.`**`Pyomo2SympyVisitor`**(*object_map*)
   This is based on the class of the same name in pyomo.core.base.symbolic, but it catches ExternalFunctions and does not decend into named expressions.

class `idaes.core.util.expr_doc.`**`PyomoSympyBimap`**
   This is based on the class of the same name in pyomo.core.base.symbolic, but it adds mapping latex symbols to the sympy symbols. This will get you pretty equations when using sympy's LaTeX writer.

`idaes.core.util.expr_doc.`**`deduplicate_symbol`**(*x*, *v*, *used*)
   Check if x is a duplicated LaTeX symbol if so add incrementing Di subscript

   Parameters

   - **x** – symbol string

   - **v** – pyomo object

   - **used** – dictionary of pyomo objects with symbols as keys

   Returns Returns a unique symbol. If x was not in used keys, returns x, otherwise adds exponents to make it unique.

`idaes.core.util.expr_doc.`**`document_constraints`**(*comp*, *doc=True*, *descend_into=True*)
   Provides nicely formatted constraint documetntation in markdown format, assuming the $$latex math$$ and $latex math$ syntax is supported.

   Parameters

   - **comp** – A Pyomo component to document in {_ConstraintData, _ExpressionData, _BlockData}.

   - **doc** – True adds a documentation table for each constraint or expression. Due to the way symbols are semi-automatiaclly generated, the exact symbol definitions may be unique to each constraint or expression, if unique LaTeX symbols were not provided everywhere in a block.

   - **descend_into** – If True, look in subblocks for constraints.

   Returns A string in markdown format with equations in LaTeX form.

`idaes.core.util.expr_doc.`**`ipython_document_constraints`**(*comp*, *doc=True*, *descend_into=True*)
   See document_constraints, this just directly displays the markdown instead of returning a string.

`idaes.core.util.expr_doc.`**`sympify_expression`**(*expr*)
   Converts Pyomo expressions to sympy expressions. This is based on the function of the same name in pyomo.core.base.symbolic. The difference between this and the Pymomo is that this one checks if the expr argument is a named expression and expands it anyway. This allows named expressions to only be expanded if they are the top level object.

`idaes.core.util.expr_doc.`**`to_latex`**(*expr*)

> Return a sympy expression for the given Pyomo expression

> > **Parameters** **`expr`** (*Expression*) – Pyomo expression

> > **Returns**

> > > **keys: sympy_expr, a sympy expression; where, markdown string** with documentation table; latex_expr, a LaTeX string representation of the expression.

> > **Return type** ([dict](#))

idaes.core.util.homotopy module

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

idaes.core.util.initialization module

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.`**`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 StateBlock 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*, ***kwds*)

> 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

> > > • **`kwds`** – a dict of argumnets to be passed to the solver

> > **Returns** A Pyomo solver results object

idaes.core.util.math module

This module contains utility functions for mathematical operators of use in equation oriented models.

`idaes.core.util.math.`**`smooth_abs`**`(a, eps=0.0001)`
General function for creating an expression for a smooth minimum or maximum.

$$|a| = sqrt(a^2 + eps^2)$$

    Parameters

- **a** – term to get absolute value from (Pyomo component, float or int)

- **eps** – smoothing parameter (Param, float or int) (default=1e-4)

    Returns An expression for the smoothed absolute value operation.

`idaes.core.util.math.`**`smooth_max`**`(a, b, eps=0.0001)`
Smooth maximum operator, using smooth_abs operator.

$$max(a, b) = 0.5 * (a + b + |a - b|)$$

    Parameters

- **a** – first term in max function

- **b** – second term in max function

- **eps** – smoothing parameter (Param or float, default = 1e-4)

    Returns An expression for the smoothed maximum operation.

`idaes.core.util.math.`**`smooth_min`**`(a, b, eps=0.0001)`
Smooth minimum operator, using smooth_abs operator.

$$max(a, b) = 0.5 * (a + b - |a - b|)$$

    Parameters

- **a** – first term in min function

- **b** – second term in min function

- **eps** – smoothing parameter (Param or float, default = 1e-4)

    Returns An expression for the smoothed minimum operation.

`idaes.core.util.math.`**`smooth_minmax`**`(a, b, eps=0.0001, sense='max')`
General function for creating an expression for a smooth minimum or maximum. Uses the smooth_abs operator.

$$minmax(a, b) = 0.5 * (a + b + -|a - b|)$$

    Parameters

- **a** – first term in mix or max function (Pyomo component, float or int)

- **b** – second term in min or max function (Pyomo component, float or int)

- **eps** – smoothing parameter (Param, float or int) (default=1e-4)

- **sense** – 'mim' or 'max' (default = 'max')

    Returns An expression for the smoothed minimum or maximum operation.

idaes.core.util.misc module

This module contains miscellaneous utility functions for use in IDAES models.

`idaes.core.util.misc.`**`TagReference`**(*s*, *description=''*)

Create a Pyomo reference with an added description string attribute to describe the reference. The intended use for these references is to create a time-indexed reference to variables in a model corresponding to plant measurment tags.

> **Parameters**
>
> - **s** – Pyomo time slice of a variable or expression
>
> - **description** (*str*) – A description the measurment
>
> **Returns** A Pyomo Reference object with an added doc attribute

`idaes.core.util.misc.`**`add_object_reference`**(*self*, *local_name*, *remote_object*)

Method to create a reference in the local model to a remote Pyomo object. This method should only be used where Pyomo Reference objects are not suitable (such as for referencing scalar Pyomo objects where the None index is undesirable).

> **Parameters**
>
> - **local_name** – name to use for local reference (str)
>
> - **remote_object** – object to make a reference to
>
> **Returns** None

`idaes.core.util.misc.`**`copy_port_values`**(*destination*, *source*)

Copy the variable values in the source port to the destination port. The ports must containt the same variables.

> **Parameters**
>
> - **(pyomo.Port)** – Copy values from this port
>
> - **(pyomo.Port)** – Copy values to this port
>
> **Returns** None

`idaes.core.util.misc.`**`extract_data`**(*data_dict*)

General method that returns a rule to extract data from a python dictionary. This method allows the param block to have a database for a parameter but extract a subset of this data to initialize a Pyomo param object.

`idaes.core.util.misc.`**`svg_tag`**(*tags*, *svg*, *outfile=None*, *idx=None*, *tag_map=None*, *show_tags=False*, *byte_encoding='utf-8'*)

Replace text in a SVG with tag values for the model. This works by looking for text elements in the SVG with IDs that match the tags or are in tag_map.

> **Parameters**
>
> - **tags** – A dictionary where the key is the tag and the value is a Pyomo Refernce. The refernce could be indexed. In yypical IDAES applications the references would be indexed by time.
>
> - **svg** – a file pointer or a string continaing svg contents
>
> - **outfile** – a file name to save the results, if None don't save
>
> - **idx** – if None not indexed, otherwise an index in the indexing set of the reference
>
> - **tag_map** – dictionary with svg id keys and tag values, to map svg ids to tags
>
> - **show_tags** – Put tag labels of the diagram instead of numbers

- **byte_encoding** – If svg is given as a byte-array, use this encoding to convert it to a string.

Returns String for SVG

idaes.core.util.model_serializer module

Functions for saving and loading Pyomo objects to json

class idaes.core.util.model_serializer.**Counter**
This is a counter object, which is an easy way to pass an interger pointer around between methods.

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.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.component.ComponentData'>, ('active',))), skip_classes=(<class 'pyomo.core.base.external.ExternalFunction'>, <class 'pyomo.core.base.sets.Set'>, <class 'pyomo.network.port.Port'>, <class 'pyomo.core.base.expression.Expression'>, <class 'pyomo.core.base.rangeset.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 tupple) containing the following elements: (1) class (compared using isinstance) (2) attribute list or None, an emptry 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 obejct 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 excpetion 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 regaurdless of this argument.

- **suffix** – If True store suffixes and component ids. If false, don't store suffixes.

- **suffix_filter** – None to store all siffixes 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*)
    Retur a StoreSpec object to store variable values, if variables are fixed and if components are active.

    Parameters only_fixed – Only load fixed variable values

idaes.core.util.model_serializer.**component_data_from_dict**(*sd*, *o*, *wts*)
    Component data to a dict.

idaes.core.util.model_serializer.**component_data_to_dict**(*o*, *wts*)
    Component data to a dict.

idaes.core.util.model_serializer.**from_json**(*o*, *sd=None*, *fname=None*, *s=None*, *wts=None*, *gz=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-compoent of o. If the saved state contains extra information, it is ignored. If the save state doesn't contain an enetry 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
> >
> > - **wts** – 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 perfomance 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

idaes.core.util.model_serializer.**to_json**(*o*, *fname=None*, *human_read=False*, *wts=None*, *metadata={}*, *gz=None*, *return_dict=False*, *return_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 possilbe
> >
> > - **metadata** – A dictionary of addtional metadata to add.
> >
> > - **wts** – is What To Save, this is a StoreSpec object that specifies what object types and attributes to save. If None, the default is used which saves the state of the compelte model state.
> >
> > - **metadata** – addtional 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.

idaes.core.util.model_statistics module

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 deacitvated Block in a model.

        Parameters block – model to be studied

        Returns A ComponentSet including any Var components which belong to a deacitvated Block but appear in an activate 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 indeactivated 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.**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.

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*)
　　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

> **Returns** A ComponentSet including all Constraint components with a residual greater than tol which appear in block

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 deacitvated Block in a model.

>   **Parameters** **block** – model to be studied
>
>   **Returns** Number of Var components which belong to a deacitvated 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 activagted 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.`**`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

`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 inequality 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 Objective 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 Objective 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)

> Generator which returns 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** 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*)

> 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)

> > **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

idaes.core.util.plot module

Convenience plotting functions for time-dependent variables.

`idaes.core.util.plot.`**`plot_dynamic`**(*time*, *y*, *ylabel*, *xlabel='time (s)'*, *title=None*, *legend=None*)

> Plot time dependent variables with pyplot.

> > **Parameters**

> > > • **`time`** (`ContinuousSet or list-like`) – Time index set

> > > • **`y`** (`list-like of list-likes of Var, Expression, Reference, or` `float`) – List of quantities to plot (multiple quantities can be plotted). Each quantity in the list should be indexed only by time. If you want to plot something that is not indexed only by time, you can create a Pyomo Reference with the correct indexing.

> > > • **`ylabel`** (`str`) – Y-axis label, required

> > > • **`xlabel`** (`str`) – X-axis label, default = 'time (s)'

> > > • **`title`** (`str or None`) – Plot title, default = None

> > > • **`legend`** (`list-like of str or None`) – Legend string for each y, default = None

> > **Returns** None

`idaes.core.util.plot.`**`stitch_dynamic`**(**args*)

> Combine time-indexed Pyomo component values from different models into one combined time set. This allows you to use multiple models to simulate sections of the time domain, and plot them all together.

> > **Parameters arguments ()** (`Positional`) – Multiple Pyomo components indexed by time, or time sets

> > **Returns**

> > > **(list) with the time indexed Pyomo compoent values concatonated for** plotting

idaes.core.util.tables module

`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 StateBlocks.

> > > - **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*)

> 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 srings`) – A list of strings that will be used as column headings. If None the attribute names will be used.

> > **Returns** A Pandas dataframe containing a data table

> > **Return type** (DataFrame)

`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.testing module

This module contains utility functions for use in testing IDAES models.

class idaes.core.util.testing.**PhysicalParameterTestBlock**(**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 ProcessBlockData config
>
>   **Keys**
>
>   > **default_arguments** Default arguments to use with Property Package
>
> - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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** (PhysicalParameterTestBlock) New instance

class idaes.core.util.testing.**RBlockBase**(**args*, ***kwargs*)

> **initialize**(*outlvl=0*, *optarg=None*, *solver=None*)
> This is a default initialization routine for ReactionBlocks to ensure that a routine is present. All Reaction-BlockData classes should overload this method with one suited to the particular reaction package
>
> > **Parameters None** –
> >
> > **Returns** None

class idaes.core.util.testing.**ReactionBlock**(**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 ProcessBlockData config
>
>   Keys
>
> - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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** (ReactionBlock) New instance

class idaes.core.util.testing.**ReactionBlockData**(*component*)

> **build**()
> General build method for PropertyBlockDatas. 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.

class idaes.core.util.testing.**ReactionParameterTestBlock**(**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 ProcessBlockData config
>
>   **Keys**
>
>   > **property_package** Reference to associated PropertyPackageParameter object
>   >
>   > **default_arguments** Default arguments to use with Property Package
>
> - **initialize** (`dict`) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (ReactionParameterTestBlock) New instance

class idaes.core.util.testing.**SBlockBase**(**args*, ***kwargs*)

initialize(*outlvl=0*, *optarg=None*, *solver=None*, *hold_state=False*, ***state_args*)
> 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

class idaes.core.util.testing.**StateTestBlockData**(*component*)

build()
> General build method for StateBlockDatas.
>
> > **Parameters** **None** –
> >
> > **Returns** None

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(*p*)
> Method which returns a valid expression for enthalpy density to use in the energy balances.

get_enthalpy_flow_terms(*p*)
> Method which returns a valid expression for enthalpy flow to use in the energy balances.

get_material_density_terms(*p*, *j*)
> Method which returns a valid expression for material density to use in the material balances .

get_material_flow_basis()
> Method which returns an Enum indicating the basis of the material flow term.

get_material_flow_terms(*p*, *j*)
>   Method which returns a valid expression for material flow to use in the material balances.

class idaes.core.util.testing.**TestStateBlock**(**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 ProcessBlockData config
>
>   Keys
>
>   - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
>   **Returns** (TestStateBlock) New instance

idaes.core.util.testing.**get_default_solver**()
>   Tries to set-up the default solver for testing, and returns None if not available

idaes.core.util.unit_costing module

idaes.core.util.unit_costing.**hx_costing**(*self*, *hx_type='U-tube'*, *FM='stain_steel'*, *FL='12ft'*)
>   Heat exchanger costing method Source: 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 22.2 Cost Indexes and Capital Investment
>
>   This method computes the purchase cost (CP) for a shell and tube heat exchanger (Eq. 22.43), the model computes the base cost (CB for 4 type of heat exchangers, such as floating head, fixed head, U-tube, and Kettle vaporizer), construction material factor (FM), pressure design factor (FP), and tube length correction factor (FL), using CE base cost index of 500.
>
>   Cp = FP*FM*FL*CB
>
>   **Parameters**
>
>   - **hx_type** – 'floating_head', 'fixed_head', 'U-tube'*, 'Kettle_vap'
>
>   - **factor** (*material*) – 'stain_steel'*, 'carb_steel'
>
>   - **length** (*tube*) – '8ft'*, '12ft', '16ft', '20ft'
>
>   - **--> default options** (***) –

Submodules

idaes.core.control_volume0d module

Base class for control volumes

---

class idaes.core.control_volume0d.**ControlVolume0DBlock**(*args*, ***kwargs*)
    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 encompases 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 ProcessBlockData 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 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.}

        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 construciton.}

- **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 overide 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 momentum 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:* *1>*, *has_phase_equilibrium=None*)
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, *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

- **– 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=6*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)
> Initialization routine for 0D 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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

> > > • **optarg** – solver options dictionary object (default=None)

> > > • **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')

> > > • **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=6*)
> 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

idaes.core.control_volume1d module

Base class for control volumes

class idaes.core.control_volume1d.**ControlVolume1DBlock**(**args*, ***kwargs*)
> 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 encompases 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 ProcessBlockData 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 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.}

    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 construciton.}

    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 domian, 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 transformating 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 Block-Data 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 overide the default behavior 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.

      - **–** – a new ContinuousSet if length_domain is not provided (default = [0.0, 1.0]).

      - **– 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*, *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

- **– 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=6*, *optarg=None*, *solver='ipopt'*, *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 * 0 = Use default idaes.init logger
     setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for
     each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final
     initialization status and exceptions * 6 = No output

   - **optarg** – solver options dictionary object (default=None)

   - **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')

   - **hold_state** – flag indicating whether the initialization routine should unfix any state
     variables fixed during initialization, **default** - True. **Valid values: True** - states vari-
     ables 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=6*)
    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.

idaes.core.control_volume_base module

Base class for control volumes

class idaes.core.control_volume_base.**ControlVolume**(**args*, ***kwargs*)
    This class is not usually used directly. Use ControlVolume0DBlock or ControlVolume1DBlock instead.

    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, **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 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.}

> > **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 construciton.}
>
> - **initialize** (`dict`) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (ControlVolume) New instance

class idaes.core.control_volume_base.**ControlVolumeBlockData**(*component*)

> The ControlVolumeBlockData Class forms the base class for all IDAES ControlVolume models. The purpose of this class is to automate the tasks common to all control volume blockss and ensure that the necessary attributes of a control volume block are present.
>
> The most signfiicant role of the ControlVolumeBlockData class is to set up the construction arguments for the control volume block, automatically link to the time domain of the parent block, and to get the information about the property and reaction packages.
>
> **add_energy_balances**(*balance_type=<EnergyBalanceType.useDefault: -1>*, ***kwargs*)
>
> > General method for adding energy balances to a control volume. This method makes calls to specialised sub-methods for each type of energy balance.
> >
> > **Parameters**
> >
> > - **balance_type** (`EnergyBalanceType`) – Enum indicating which type of energy balance should be constructed.
> >
> > - **has_heat_of_reaction** (`bool`) – whether terms for heat of reaction should be included in energy balance
> >
> > - **has_heat_transfer** (`bool`) – whether generic heat transfer terms should be included in energy balances
> >
> > - **has_work_transfer** (`bool`) – whether generic mass transfer terms should be included in energy balances
> >
> > - **custom_term** (`Expression`) – a Pyomo Expression representing custom terms to be included in energy balances
> >
> > **Returns** Constraint objects constructed by sub-method
>
> **add_geometry**(**args*, ***kwargs*)
>
> > Method for defining the geometry of the control volume.
> >
> > See specific control volume documentation for details.
>
> **add_material_balances**(*balance_type=<MaterialBalanceType.useDefault: -1>*, ***kwargs*)
>
> > General method for adding material balances to a control volume. This method makes calls to specialised sub-methods for each type of material balance.

Parameters

- **– MaterialBalanceType Enum indicating which type of**
  (`balance_type`) – material balance should be constructed.

- **– 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.

- **– a Pyomo Expression representing custom terms to** – be included in
  material balances on a mass basis.

Returns Constraint objects constructed by sub-method

add_momentum_balances(*balance_type=<MomentumBalanceType.pressureTotal: 1>*, ***kwargs*)
General method for adding momentum balances to a control volume. This method makes calls to specialised sub-methods for each type of momentum balance.

Parameters

- **balance_type** (`MomentumBalanceType`) – Enum indicating which type of momentum balance should be constructed. Default = MomentumBalanceType.pressureTotal.

- **has_pressure_change** ([`bool`]) – whether default generation terms for pressure change should be included in momentum balances

- **custom_term** (`Expression`) – a Pyomo Expression representing custom terms to be included in momentum balances

Returns Constraint objects constructed by sub-method

add_phase_component_balances(**args*, ***kwargs*)
Method for adding material balances indexed by phase and component to the control volume.

See specific control volume documentation for details.

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(*args*, **kwargs*)
Method for adding ReactionBlocks to the control volume.

See specific control volume documentation for details.

add_state_blocks(*args*, **kwargs*)
Method for adding StateBlocks to the control volume.

See specific control volume documentation for details.

add_total_component_balances(*args*, **kwargs*)
Method for adding material balances indexed by component to the control volume.

See specific control volume documentation for details.

add_total_element_balances(*args*, **kwargs*)
Method for adding total elemental material balances indexed to the control volume.

See specific control volume documentation for details.

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(*args*, **kwargs*)
Method for adding a total enthalpy balance to the control volume.

See specific control volume documentation for details.

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(*args*, **kwargs*)
Method for adding a total pressure balance to the control volume.

See specific control volume documentation for details.

build()
General build method for Control Volumes blocks. This method calls a number of sub-methods which automate the construction of expected attributes of all ControlVolume blocks.

Inheriting models should call *super().build*.

> **Parameters None** –

> **Returns** None

class idaes.core.control_volume_base.**EnergyBalanceType**
An enumeration.

class idaes.core.control_volume_base.**FlowDirection**
An enumeration.

class idaes.core.control_volume_base.**MaterialBalanceType**
An enumeration.

class idaes.core.control_volume_base.**MomentumBalanceType**

An enumeration.

idaes.core.flowsheet_model module

This module contains the base class for constructing flowsheet models in the IDAES modeling framework.

class idaes.core.flowsheet_model.**FlowsheetBlock**(*args*, ***kwargs*)

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].
>   >
>   > **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (FlowsheetBlock) New instance

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 signfiicant 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

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

serialize(*file_base_name*, *overwrite=False*)
> Serializes the flowsheet and saves it to a file that can be read by the idaes-model-vis jupyter lab extension.

> > **Parameters** **file_base_name** – The file prefix to the .idaes.vis file produced.

> The file is created/saved in the directory that you ran from Jupyter Lab. :param overwrite: Boolean to overwrite an existing file_base_name.idaes.vis. If True, the existing file with the same file_base_name will be overwritten. This will cause you to lose any saved layout. If False and there is an existing file with that file_base_name, you will get an error message stating that you cannot save a file to the file_base_name (and therefore overwriting the saved layout). If there is not an existing file with that file_base_name then it saves as normal. Defaults to False. :return: 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

idaes.core.process_base module

Base for IDAES process model objects.

class idaes.core.process_base.**ProcessBaseBlock**(**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 ProcessBlockData config

>   Keys

> - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (ProcessBaseBlock) New instance

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 sepecified 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 implimented 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
>
> **unfix_initial_conditions**()
>
> > This method unfixed the initial conditions for dynamic models.
> >
> > > **Parameters None** –
> > >
> > > **Returns :** None

idaes.core.process_block module

The process_block module simplifies inheritance of Pyomo blocks. The main reason to subclass a Pyomo block is to create a block that comes with pre-defined model equations. This is used in the IDAES modeling framework to create modular process model blocks.

class idaes.core.process_block.**ProcessBlock**(**args*, ***kwargs*)

> 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 Block-Data 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.

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 requirment 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

idaes.core.property_base module

This module contains classes for property blocks and property parameter blocks.

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 PropertyParameterBlocks. Inheriting models should call super().build.
>
>> **Parameters** **None** –
>>
>> **Returns** None

class idaes.core.property_base.**StateBlock**(**args*, ***kwargs*)
> 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 StateBlock-Data 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

class idaes.core.property_base.**StateBlockData**(*component*)
> 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 StateBlockDatas.
>>
>>> **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.

idaes.core.property_meta module

These classes handle the metadata aspects of classes representing property packages.

Implementors of property packages need to do the following:

1. Create a new class that inherits from *idaes.core.property_base.PhysicalParameterBlock*, which in turn inherits from HasPropertyClassMetadata, in this module.

2. In that class, implement the *define_metadata()* method, inherited from HasPropertyClassMetadata. This method is called automatically, once, when the *get_metadata()* method is first invoked. An empty metadata object (an instance of PropertyClassMetadata) will be passed in, which the method should populate with information about properties and default units.

Example:

```
from idaes.core.property_base import PhysicalParameterBlock

class MyPropParams(PhysicalParameterBlock):

    @classmethod
    def define_metadata(cls, meta):
        meta.add_default_units({foo.U.TIME: 'fortnights',
```

(continues on next page)

```
                                  foo.U.MASS: 'stones'})
        meta.add_properties({'under_sea': {'units': 'leagues'},
                             'tentacle_size': {'units': 'yards'}})
        meta.add_required_properties({'under_sea': 'leagues',
                             'tentacle_size': 'yards'})

    # Also, of course, implement the non-metadata methods that
    # do the work of the class.
```

class idaes.core.property_meta.**HasPropertyClassMetadata**

Interface for classes that have PropertyClassMetadata.

> **classmethod define_metadata**(*pcm*)
>
> 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
>
> **classmethod get_metadata**()
>
> Get property parameter metadata.
>
> If the metadata is not defined, this will instantiate a new metadata object and call *define_metadata()* to set it up.
>
> If the metadata is already defined, it will be simply returned.
>
> > **Returns** The metadata
> >
> > **Return type** PropertyClassMetadata

class idaes.core.property_meta.**PropertyClassMetadata**

Container for metadata about the property class, which includes default units and properties.

Example usage:

```
foo = PropertyClassMetadata()
foo.add_default_units({foo.U.TIME: 'fortnights',
                       foo.U.MASS: 'stones'})
foo.add_properties({'under_sea': {'units': 'leagues'},
                    'tentacle_size': {'units': 'yards'}})
foo.add_required_properties({'under_sea': 'leagues',
                             'tentacle_size': 'yards'})
```

> **U**
>
> Alias for class enumerating supported/known unit types
>
> alias of UnitNames
>
> **add_default_units**(*u*)
>
> Add a dict with keys for the quantities used in the property package (as strings) and values of their default units as strings.
>
> The quantities used by the framework are in constants defined in UnitNames, aliased here in the class attribute *U*.
>
> > **Parameters u** (*dict*) – Key=property, Value=units
> >
> > **Returns** None

add_properties(*p*)
> Add properties to the metadata.
>
> For each property, the value should be another dict which may contain the following keys:
>
> - **'method': (required) the name of a method to construct the** property as a str, or None if the property will be constructed by default.
>
> - 'units': (optional) units of measurement for the property.
>
> > **Parameters p** (*dict*) – Key=property, Value=PropertyMetadata or equiv. dict
> >
> > **Returns** None

add_required_properties(*p*)
> Add required properties to the metadata.
>
> For each property, the value should be the expected units of measurement for the property.
>
> > **Parameters p** (*dict*) – Key=property, Value=units
> >
> > **Returns** None

class idaes.core.property_meta.**PropertyMetadata**(*name=None*, *method=None*, *units=None*)
> Container for property parameter metadata.
>
> Instances of this class are exactly dictionaries, with the only difference being some guidance on the values expected in the dictionary from the constructor.

class idaes.core.property_meta.**UnitNames**
> Names for recognized units.

idaes.core.reaction_base module

This module contains classes for reaction blocks and reaction parameter blocks.

class idaes.core.reaction_base.**ReactionBlockBase**(**args*, ***kwargs*)
> 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

class idaes.core.reaction_base.**ReactionBlockDataBase**(*component*)
> This is the base class for reaction block data objects. These are blocks that contain the Pyomo components associated with calculating a set of reacion properties for a given material.

build()
> General build method for PropertyBlockDatas. 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.

class idaes.core.reaction_base.**ReactionParameterBlock**(*component*)

> 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

idaes.core.unit_model module

Base class for unit models

class idaes.core.unit_model.**UnitModelBlock**(**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 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:** { **True** - construct holdup terms, **False** - do not construct holdup terms}
>
> - **initialize** (`dict`) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (UnitModelBlock) New instance

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 to use for Port object** (`name`) –
> > >
> > > - **= an instance of a ControlVolume or StateBlock to use as the** (`block`) – 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 string for Port object** (*doc*) –

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 to use for Port object** (*name*) –

- **= an instance of a ControlVolume or StateBlock to use as
  the** (*block*) – 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 string for Port object** (*doc*) –

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
name = name to use for Port object.: :keyword block = an instance of a StateBlock to use as the source to:
populate the Port object :keyword doc = doc string for Port object:

Returns A Pyomo Port object and associated components.

build ()

General build method for UnitModelBlockData. This method calls a number of sub-methods which auto-
mate the construction of expected attributes of unit models.

Inheriting models should call *super().build*.

Parameters None –

Returns None

initialize (*state_args=None*, *outlvl=6*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)

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 * 0 = Use default idaes.init logger
  setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for
  each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final
  initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

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

idaes.dmf package

IDAES Data Management Framework (DMF)

The DMF lets you save, search, and retrieve provenance related to your models.

Subpackages

idaes.dmf.ui package

Submodules

idaes.dmf.ui.flowsheet_serializer module

exception idaes.dmf.ui.flowsheet_serializer.**FileBaseNameExistsError**

idaes.dmf.ui.icon_mapping module

idaes.dmf.ui.link_position_mapping module

Submodules

idaes.dmf.cli module

Command Line Interface for IDAES DMF.

Uses "Click" to handle command-line parsing and dispatch.

class idaes.dmf.cli.**AliasedGroup**(*aliases=None*, ***attrs*)

> Improved click.Group that will accept unique prefixes for the commands, as well as a set of aliases.
>
> For example, the following code will create *mycommand* as a group, and alias the subcommand "info" to invoke the subcommand "status". Any unique prefix of "info" (not conflicting with other subcommands or aliases) or "status" will work, e.g. "inf" or "stat":

```
@click.group(cls=AliasedGroup, aliases={"info": "status"})
def mycommand():
    pass
```

get_command(*ctx*, *cmd_name*)

Given a context and a command name, this returns a `Command` object if it exists or returns *None*.

class idaes.dmf.cli.**Code**

Return codes from the CLI.

class idaes.dmf.cli.**URLType**

Click type for URLs.

convert(*value*, *param*, *ctx*)

Converts the value. This is not invoked for values that are *None* (the missing value).

idaes.dmf.codesearch module

Search through the code and index static information in the DMF.

class idaes.dmf.codesearch.**ModuleClassWalker**(*from_path=None*, *from_pkg=None*, *class_expr=None*, *parent_class=None*, *suppress_warnings=False*, *exclude_testdirs=True*, *exclude_tests=True*, *exclude_init=True*, *exclude_setup=True*, *exclude_dirs=None*)

Walk modules from a given root (e.g. 'idaes'), and visit all classes in those modules whose name matches a given pattern.

Example usage:

```
walker = ModuleClassWalker(from_pkg=idaes,
                           class_expr='_PropertyParameter.*')

walker.walk(PrintMetadataVisitor())   # see below
```

walk(*visitor*)

Interface for walkers.

> **Parameters** **visitor** (`Visitor`) – Class whose *visit* method will be called for each item.

> **Returns** None

class idaes.dmf.codesearch.**PrintPropertyMetadataVisitor**

visit_metadata(*obj*, *meta*)

Print the module and class of the object, and then the metadata dict, to standard output.

class idaes.dmf.codesearch.**PropertyMetadataVisitor**

Visit something implementing `HasPropertyClassMetadata` and pass that metadata, as a dict, to the *visit_metadata()* method, which should be implemented by the subclass.

visit(*obj*)

Visit one object.

> **Parameters** **obj** (*idaes.core.property_base.HasPropertyClassMetadata*) – The object

> **Returns** True if visit succeeded, else False

visit_metadata(*obj*, *meta*)

Do something with the metadata.

> **Parameters**

- **obj** (*object*) – Object from which metadata was pulled, for context.

- **meta** (*idaes.core.property_base.PropertyClassMetadata*) – The metadata

        Returns None

class idaes.dmf.codesearch.**Visitor**

   Interface for the 'visitor' class passed to Walker subclasses' *walk()* method.

   visit(*obj*)
       Visit one object.

           Parameters obj (*object*) – Some object to operate on.

           Returns True if visit succeeded, else False

idaes.dmf.commands module

Perform all logic, input, output of commands that is particular to the CLI.

Call functions defined in 'api' module to handle logic that is common to the API and CLI.

idaes.dmf.commands.**init_conf**(*workspace*)
   Initialize the workspace.

idaes.dmf.commands.**list_resources**(*path*, *long_format=None*, *relations=False*)
   List resources in a given DMF workspace.

       Parameters

               - **path** (*str*) – Path to the workspace

               - **long_format** (*bool*) – List in long format flag

               - **relations** (*bool*) – Show relationships, in long format

       Returns None

idaes.dmf.commands.**list_workspaces**(*root*, *stream=None*)
   List workspaces found from a given root path.

       Parameters

               - **root** – root path

               - **stream** – Output stream (must have .write() method)

idaes.dmf.commands.**workspace_import**(*path*, *patterns*, *exit_on_error*)
   Import files into workspace.

       Parameters

               - **path** (*str*) – Target workspace directory

               - **patterns** (*list*) – List of Unix-style glob for files to import. Files are expected to be resource JSON or a Jupyter Notebook.

               - **exit_on_error** (*bool*) – If False, continue trying to import resources even if one or more fail.

       Returns Number of things imported

       Return type int

       Raises BadResourceError, if there is a problem

idaes.dmf.commands.**workspace_init**(*dirname*, *metadata*)
>   Initialize from root at *dirname*, set environment variable for other commands, and parse config file.

idaes.dmf.dmfbase module

Data Management Framework

class idaes.dmf.dmfbase.**DMF**(*path=''*, *name=None*, *desc=None*, *create=False*, *save_path=False*,
>   ***ws_kwargs*)

>   Data Management Framework (DMF).

>   Expected usage is to instantiate this class, once, and then use it for storing, searching, and retrieving resources that are required for the given analysis.

>   For details on the configuration files used by the DMF, see documentation for *DMFConfig* (global configuration) and *idaes.dmf.workspace.Workspace*.

>   **add**(*rsrc*)
>   >   Add a resource and associated files.

>   >   If the resource has 'datafiles', there are some special values that cause those files to be copied and possibly the original removed at this point. There are attributes *do_copy* and *is_tmp* on the resource, and also potentially keys of the same name in the datafiles themselves. If present, the datafile key/value pairs will override the attributes in the resource. For *do_copy*, the original file will be copied into the DMF workspace. If *do_copy* is True, then if *is_tmp* is also True the original file will be removed (after the copy is made, of course).

>   >   **Parameters rsrc** (resource.Resource) – The resource

>   >   **Returns** (str) Resource ID

>   >   **Raises** DMFError, DuplicateResourceError

>   **fetch_one**(*rid*, *id_only=False*)
>   >   Fetch one resource, from its identifier.

>   >   **Parameters**

>   >   - **rid** (*str*) – Resource identifier

>   >   - **id_only** (*bool*) – If true, return only the identifier of each resource; otherwise a Resource object is returned.

>   >   **Returns** (resource.Resource) The found resource, or None if no match

>   **find**(*filter_dict=None*, *name=None*, *id_only=False*, *re_flags=0*)
>   >   Find and return resources matching the filter.

>   >   The filter syntax is a subset of the MongoDB filter syntax. This means that it is represented as a dictionary, where each key is an attribute or nested attribute name, and each value is the value against which to match. There are six possible types of values:

>   >   1. scalar string or number (int, float): Match resources that have this exact value for the given attribute.

>   >   2. special scalars "@<value>":

>   >   - "@true"/"@false": boolean (bare True/False will test existence)

>   >   3. date, as datetime.datetime or pendulum.Pendulum instance: Match resources that have this exact date for the given attribute.

4. list: Match resources that have a list value for this attribute, and for which any of the values in the provided list are in the resource's corresponding value. If a '!' is appended to the key name, then this will be interpreted as a directive to only match resources for which *all* values in the provided list are present.

5. dict: This is an inequality, with one or more key/value pairs. The key is the type of inequality and the value is the numeric value for that range. All keys begin with '$'. The possible inequalities are:

   - "$lt": Less than (<)
   - "$le": Less than or equal (<=)
   - "$gt": Greater than (>)
   - "$ge": Greater than or equal (>=)
   - "$ne": Not equal to (!=)

6. Boolean True means does the field exist, and False means does it *not* exist.

7. Regular expression, string "~<expr>" and *re_flags* for flags (understood: re.IGNORECASE)

   Parameters

   - **filter_dict** (`dict`) – Search filter.
   - **name** (`str`) – If present, add {'aliases': [<name>]} to filter_dict. This is syntactic sugar for a common case.
   - **id_only** (`bool`) – If true, return only the identifier of each resource; otherwise a Resource object is returned.
   - **re_flags** (`int`) – Flags for regex filters

   Returns (list of int|Resource) Depending on the value of *id_only*.

find_by_id(*identifier: str*, *id_only=False*) → Generator[T_co, T_contra, V_co]
    Find resources by their identifier or identifier prefix.

find_related(*rsrc*, *filter_dict=None*, *maxdepth=0*, *meta=None*, *outgoing=True*)
    Find related resources.

   Parameters

   - **rsrc** (`resource.Resource`) – Resource starting point
   - **filter_dict** (`dict`) – See parameter of same name in `find()`.
   - **maxdepth** (`int`) – Maximum depth of search (starts at 1)
   - **meta** (`List[str]`) – Metadata fields to extract for meta part
   - **outgoing** (`bool`) – If True, look at outgoing relations. Otherwise look at incoming relations. e.g. if A 'uses' B and if True, would find B starting from A. If False, would find A starting from B.

   Returns Generates triples (depth, Triple, meta), where the depth is an integer (starting at 1), the Triple is a simple namedtuple wrapping (subject, object, predicate), and *meta* is a dict of metadata for the endpoint of the relation (the object if outgoing=True, the subject if outgoing=False) for the fields provided in the *meta* parameter.

   Raises `NoSuchResourceError` – if the starting resource is not found

remove (*identifier=None*, *filter_dict=None*, *update_relations=True*)

Remove one or more resources, from its identifier or a filter. Unless told otherwise, this method will scan the DB and remove all relations that involve this resource.

> **Parameters**
>
> * **identifier** (*str*) – Identifier for a resource.
>
> * **filter_dict** (*dict*) – Filter to use instead of identifier
>
> * **update_relations** (*bool*) – If True (the default), scan the DB and remove all relations that involve this identifier.

update (*rsrc*, *sync_relations=False*, *upsert=False*)

Update/insert stored resource.

> **Parameters**
>
> * **rsrc** (*resource.Resource*) – Resource instance
>
> * **sync_relations** (*bool*) – If True, and if resource exists in the DB, then the "relations" attribute of the provided resource will be changed to the stored value.
>
> * **upsert** (*bool*) – If true, and the resource is not in the DMF, then insert it. If false, and the resource is not in the DMF, then do nothing.
>
> **Returns**
>
> > **True if the resource was updated or added, False if nothing** was done.
>
> **Return type** bool
>
> **Raises** errors.DMFError – If the input resource was invalid.

class idaes.dmf.dmfbase.**DMFConfig** (*defaults=None*)

Global DMF configuration.

Every time you create an instance of the *DMF* or run a dmf command on the command-line, the library opens the global DMF configuration file to figure out the default workspace (and, eventually, other values).

The default location for this configuration file is "~/.dmf", i.e. the file named ".dmf" in the user's home directory. This can be modified programmatically by changing the "filename" attribute of this class.

The contents of the configuration are formatted as YAML with the following keys defined:

> **workspace** Path to the default workspace directory.

idaes.dmf.errors module

Exception classes.

exception idaes.dmf.errors.**AlamoDisabledError**

exception idaes.dmf.errors.**AlamoError** (*msg*)

exception idaes.dmf.errors.**BadResourceError**

exception idaes.dmf.errors.**CommandError** (*command*, *operation*, *details*)

exception idaes.dmf.errors.**DMFError** (*detailed_error='No details'*)

exception idaes.dmf.errors.**DataFormatError** (*dtype*, *err*)

exception idaes.dmf.errors.**DmfError**

exception idaes.dmf.errors.**DuplicateResourceError** (*op*, *id_*)

exception idaes.dmf.errors.**FileError**

exception idaes.dmf.errors.**InvalidRelationError**(*subj*, *pred*, *obj*)

exception idaes.dmf.errors.**ModuleFormatError**(*module_name*, *type_*, *what*)

exception idaes.dmf.errors.**NoSuchResourceError**(*name=None*, *id_=None*)

exception idaes.dmf.errors.**ParseError**

exception idaes.dmf.errors.**ResourceError**

exception idaes.dmf.errors.**SearchError**(*spec*, *problem*)

exception idaes.dmf.errors.**WorkspaceCannotCreateError**(*path*)

exception idaes.dmf.errors.**WorkspaceConfMissingField**(*path*, *name*, *desc*)

exception idaes.dmf.errors.**WorkspaceConfNotFoundError**(*path*)

exception idaes.dmf.errors.**WorkspaceError**(*detailed_error='No details'*)

exception idaes.dmf.errors.**WorkspaceNotFoundError**(*from_dir*)

idaes.dmf.experiment module

The 'experiment' is a root container for a coherent set of 'resources'.

class idaes.dmf.experiment.**Experiment**(*dmf*, ***kwargs*)

An experiment is a way of grouping resources in a way that makes sense to the user.

It is also a useful unit for passing as an argument to functions, since it has a standard 'slot' for the DMF instance that created it.

add(*rsrc*)

Add a resource to an experiment.

This does two things:

1. Establishes an "experiment" type of relationship between the new resource and the experiment.

2. Adds the resource to the DMF

> **Parameters rsrc** (`resource.Resource`) – The resource to add.

> **Returns** Added (input) resource, for chaining calls.

> **Return type** *resource.Resource*

copy(*new_id=True*, ***kwargs*)

Get a copy of this experiment. The returned object will have been added to the DMF.

Parameters

- **new_id** (`bool`) – If True, generate a new unique ID for the copy.

- **kwargs** – Values to set in new instance after copying.

Returns

A (mostly deep) copy.

Note that the DMF instance is just a reference to the same object as in the original, and they will share state.

Return type *Experiment*

link (*subj*, *predicate='contains'*, *obj=None*)
    Add and update relation triple in DMF.

> **Parameters**
>
> - **subj** (`resource.Resource`) – Subject
>
> - **predicate** (`str`) – Predicate
>
> - **obj** (`resource.Resource`) – Object
>
> **Returns** None

remove ()
    Remove this experiment from the associated DMF instance.

update ()
    Update experiment to current values.

idaes.dmf.help module

Find documentation for modules and classes in the generated Sphinx documentation and return its location.

idaes.dmf.help.**find_html_docs** (*dmf*, *obj=None*, *obj_name=None*, ***kw*)
    Get one or more files with HTML documentation for the given object, in paths referred to by the dmf instance.

idaes.dmf.magics module

Jupyter magics for the DMF.

exception idaes.dmf.magics.**DMFMagicError** (*errmsg*, *usermsg=None*)

class idaes.dmf.magics.**DmfMagics** (*shell*)
    Implement "magic" commands in Jupyter/IPython for interacting with the DMF and IDAES more generally.

    In order to allow easier testing, the functionality is broken into two classes. This class has the decorated method(s) for invoking the 'magics', and *DmfMagicsImpl* has the state and functionality.

    dmf (*line*)
        DMF outer command.

        Example:

        ```
        %dmf <subcommand> [subcommand args..]
        ```

class idaes.dmf.magics.**DmfMagicsImpl** (*shell*)
    State and implementation called by DmfMagics.

    On failure of any method, a *DMFMagicError* is raised, that should be handled by the line or cell magic that invoked it.

    dmf (*line*)
        DMF outer command

    dmf_help (**names*)
        Provide help on IDAES objects and classes. Invoking with no arguments gives general help. Invoking with one argument looks for help in the docs on the given object or class. *Arguments*: *[name]*.

    dmf_info (**topics*)
        Provide information about DMF current state. *Arguments*: none

---

> > Parameters **topics** (*(List[str])*) – List of topics
>
> > Returns None

dmf_init(*path*, **extra*)

> Initialize DMF (do this before most other commands). *Arguments*: *path ["create"]*

> > Parameters

> > > - **path** (*str*) – Full path to DMF home
> > >
> > > - **extra** (*str*) – Extra tokens. If 'create', then try to create the path if it is not found.

dmf_list()

> List resources in the current workspace. *Arguments*: none.

dmf_workspaces(**paths*)

> List DMF workspaces. Optionally takes one or more paths to use as a starting point. By default, start from current directory. *Arguments*: *[paths..]*

> > Parameters **paths** (*List[str]*) – Paths to search, use "." by default

idaes.dmf.magics.**register**()

> Register with IPython on import (once).

idaes.dmf.model_data module

This module contains functions to read and manage data for use in parameter esitmation, data reconciliation, and validation.

idaes.dmf.model_data.**read_data**(*csv_file*, *csv_file_metadata*, *model=None*, *rename_mapper=None*, *unit_system=None*, *ambient_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.

> > Parameters

> > > - **csv_file** (*str*) – Path of file to read
> > >
> > > - **csv_file_metadata** (*str*) – Path of csv file to read column metadata from
> > >
> > > - **model** (*ConcreteModel*) – Optional model to map tags to
> > >
> > > - **rename_mapper** (*function*) – Optional function to rename tags
> > >
> > > - **unit_system** (*str*) – Optional system of units to atempt 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

> **A Pandas data frame with tags in columns and rows indexed** by time.
>
> **(dict): Column metadata, units of measure, description, and model** mapping information.
>
> **Return type** (DataFrame)

`idaes.dmf.model_data.`**`unit_convert`**(*x*, *frm*, *to=None*, *system=None*, *unit_string_map={}*, *ignore_units=[]*, *gauge_pressures={}*, *ambient_pressure=1.0*, *ambient_pressure_unit='atm'*)

Convert the quantity x to a different set of units. X can be a numpy array or pandas series. The from unit is translated into a string that pint can recognize by first looking in unit_string_map then looking in know aliases defined in this file. If it is neither place it will be given to pint as-is. This translation of the unit is done so that data can be read in with the original provided units.

> **Parameters**
>
> - **x** (*float, numpy.array, pandas.series*) – quantity to convert
>
> - **frm** (*str*) – original unit string
>
> - **to** (*str*) – new unit string, or specify "system"
>
> - **system** (*str*) – unit system to covert to, or specify "to"
>
> - **unit_string_map** (*dict*) – keys are unit strings and values are corresponding strings that pint can recognize. This only applies to the from string.
>
> - **ignore_units** (*list, or tuple*) – units to not convert
>
> - **gauge_pressures** (*dict*) – keys are units strings to be considered gauge pressures and the values are corresponding absolute pressure units
>
> - **ambient_pressure** (*float, numpy.array, pandas.series*) – pressure to add to gauge pressure to convert it to absolute pressure. The default is 1. The unit is atm by default, but can be changed with the ambient_pressure_unit argument.
>
> - **ambient_pressure_unit** (*str*) – Unit for ambient pressure, default is atm, and should be a unit recognized by pint
>
> **Returns** quantity and unit string
>
> **Return type** (tuple)

idaes.dmf.propdata module

Property data types.

Ability to import, etc. from text files is part of the methods in the type.

Import property database from textfile(s): * See *PropertyData.from_csv()*, for the expected format for data. * See *PropertyMetadata()* for the expected format for metadata.

exception `idaes.dmf.propdata.`**`AddedCSVColumnError`**(*names*, *how_bad*, *column_type=''*)
    Error for :meth:PropertyData.add_csv()

class `idaes.dmf.propdata.`**`Fields`**
    Constants for fields.

class `idaes.dmf.propdata.`**`PropertyColumn`**(*name*, *data*)
    Data column for a property.

class `idaes.dmf.propdata.`**`PropertyData`**(*data*)
    Class representing property data that knows how to construct itself from a CSV file.

You can build objects from multiple CSV files as well. See the property database section of the API docs for details, or read the code in *add_csv()* and the tests in idaes_dmf.propdb.tests.test_mergecsv.

add_csv(*file_or_path*, *strict=False*)

Add to existing object from a new CSV file.

Depending on the value of the *strict* argument (see below), the new file may or may not have the same properties as the object – but it always needs to have the same number of state columns, and in the same order.

Note: Data that is "missing" because of property columns in one CSV and not the other will be filled with *float(nan)* values.

Parameters

- **file_or_path** (*file or str*) – Input file. This should be in exactly the same format as expected by :meth:from_csv().

- **strict** (*bool*) – If true, require that the columns in the input CSV match columns in this object. Otherwise, only require that *state* columns in input CSV match columns in this object. New property columns are added, and matches to existing property columns will append the data.

Raises *AddedCSVColumnError* – If the new CSV column headers are not the same as the ones in this object.

Returns (int) Number of added rows

as_arr(*states=True*)

Export property data as arrays.

Parameters states (*bool*) – If False, exclude "state" data, e.g. the ambient temperature, and only include measured property values.

Returns (values[M,N], errors[M,N]) Two arrays of floats, each with M columns having N values.

Raises ValueError if the columns are not all the same length

errors_dataframe(*states=False*)

Get errors as a dataframe.

Parameters states (*bool*) – If False, exclude state data. This is the default, because states do not normally have associated error information.

Returns Pandas dataframe for values.

Return type pd.DataFrame

Raises *ImportError* – If *pandas* or *numpy* were never successfully imported.

static from_csv(*file_or_path*, *nstates=0*)

Import the CSV data.

Expected format of the files is a header plus data rows.

Header: Index-column, Column-name(1), Error-column(1), Column-name(2), Error-column(2), .. Data: <index>, <val>, <errval>, <val>, <errval>, ..

Column-name is in the format "Name (units)"

Error-column is in the format "<type> Error", where "<type>" is the error type.

> **Parameters**
>
> > - **file_or_path** (*file-like or str*) – Input file
> >
> > - **nstates** (*int*) – Number of state columns, appearing first before property columns.
>
> **Returns** New properties instance
>
> **Return type** *PropertyData*

is_property_column(*index*)
> Whether given column is a property. See *is_state_column()*.

is_state_column(*index*)
> Whether given column is state.
>
> > **Parameters index** (*int*) – Index of column
> >
> > **Returns** (bool) State or property and the column number.
> >
> > **Raises** IndexError – No column at that index.

names(*states=True*, *properties=True*)
> Get column names.
>
> > **Parameters**
> >
> > > - **states** (*bool*) – If False, exclude "state" data, e.g. the ambient temperature, and only include measured property values.
> > >
> > > - **properties** (*bool*) – If False, exclude property data
> >
> > **Returns** List of column names.
> >
> > **Return type** list[str]

values_dataframe(*states=True*)
> Get values as a dataframe.
>
> > **Parameters states** (*bool*) – see *names()*.
> >
> > **Returns** (pd.DataFrame) Pandas dataframe for values.
> >
> > **Raises** ImportError – If *pandas* or *numpy* were never successfully imported.

class idaes.dmf.propdata.**PropertyMetadata**(*values=None*)
> Class to import property metadata.

class idaes.dmf.propdata.**PropertyTable**(*data=None*, ***kwargs*)
> Property data and metadata together (at last!)

> **classmethod load**(*file_or_path*, *validate=True*)
> > Create PropertyTable from JSON input.
> >
> > > **Parameters**
> > >
> > > > - **file_or_path** (*file or str*) – Filename or file object from which to read the JSON-formatted data.
> > > >
> > > > - **validate** (*bool*) – If true, apply validation to input JSON data.
> > >
> > > Example input:

```
{
    "meta": [
        {"datatype": "MEA",
         "info": "J. Chem. Eng. Data, 2009, Vol 54, pg. 306-310",
         "notes": "r is MEA weight fraction in aqueous soln.",
         "authors": "Amundsen, T.G., Lars, E.O., Eimer, D.A.",
         "title": "Density and Viscosity of ..."}
    ],
    "data": [
        {"name": "Viscosity Value",
         "units": "mPa-s",
         "values": [2.6, 6.2],
         "error_type": "absolute",
         "errors": [0.06, 0.004],
         "type": "property"},
        {"name": "r",
         "units": "",
         "values": [0.2, 1000],
         "type": "state"}
    ]
}
```

class idaes.dmf.propdata.**StateColumn**(*name*, *data*)

> Data column for a state.

idaes.dmf.propindex module

Index Property metadata

class idaes.dmf.propindex.**DMFVisitor**(*dmf*, *default_version=None*)

> **INDEXED_PROPERTY_TAG = 'indexed-property'**
> > Added to resource 'tags', so easier to find later
>
> **visit_metadata**(*obj*, *meta*)
> > **Called for each property class encountered during the "walk"** initiated by *index_property_metadata()*.
> >
> > > **Parameters**
> > >
> > > - **obj** (*property_base.PropertyParameterBase*) – Property class instance
> > >
> > > - **meta** (*property_base.PropertyClassMetadata*) – Associated metadata
> > >
> > > **Returns** None
> > >
> > > **Raises** AttributeError – if

idaes.dmf.propindex.**index_property_metadata**(*dmf*, *pkg=<module 'idaes' from '/home/docs/checkouts/readthedocs.org/user_builds/idaes-pse/checkouts/1.4.4/idaes/__init__.py'>*, *expr='_PropertyMetadata.*'*, *default_version='0.0.1'*, ***kwargs*)

Index all the PropertyMetadata classes in this package.

Usually the defaults will be correct, but you can modify the package explored and set of classes indexed.

When you re-index the same class (in the same module), whether or not that is a "duplicate" will depend on the version found in the containing module. If there is no version in the containing module, the default version is used (so it is always the same). If it is a duplicate, nothing is done, this is not considered an error. If a new version is added, it will be explicitly connected to the highest version of the same module/code. So, for example,

1. Starting with (a.module.ClassName version=0.1.2)

2. If you then find a new version (a.module.ClassName version=1.2.3) There will be 2 resources, and you will have the relation:

```
a.module.ClassName/1.2.3 --version---> a.module.ClassName/0.1.2
```

3. If you add another version (a.module.ClassName version=1.2.4), you will have two relations:

```
a.module.ClassName/1.2.3 --version---> a.module.ClassName/0.1.2
a.module.ClassName/1.2.4 --version---> a.module.ClassName/1.2.3
```

> **Parameters**
>
> - **dmf** (`idaes.dmf.DMF`) – Data Management Framework instance in which to record the found metadata.
> - **pkg** (`module`) – Root module (i.e. package root) from which to find the classes containing metadata.
> - **expr** (`str`) – Regular expression pattern for the names of the classes in which to look for metadata.
> - **default_version** (`str`) – Default version to use for modules with no explicit version.
> - **kwargs** – Other keyword arguments passed to `codesearch.ModuleClassWalker`.
>
> **Returns**
>
> > **Class that walked through the modules.** You can call *.get_indexed_classes()* to see the list of classes walked, or *.walk()* to walk the modules again.
>
> **Return type** *codesearch.ModuleClassWalker*
>
> **Raises**
>
> - This instantiated a *DMFVisitor* and calls its *walk()* method to
> - walk/visit each found class, so any exception raised by the constructor
> - or *DMFVisitor.visit_metadata()*.

idaes.dmf.resource module

Resource representaitons.

class idaes.dmf.resource.**CodeImporter**(*path*, *language*, ***kwargs*)

class idaes.dmf.resource.**Dict**(**args*, ***kwargs*)
    Subclass of dict that has a 'dirty' bit.

class idaes.dmf.resource.**FileImporter**(*path: pathlib.Path*, *do_copy: bool = None*)

class idaes.dmf.resource.**JsonFileImporter**(*path: pathlib.Path*, *do_copy: bool = None*)

class idaes.dmf.resource.**JupyterNotebookImporter**(*path: pathlib.Path*, *do_copy: bool =*
                                                                                    None)

idaes.dmf.resource.**PR_DERIVED = 'derived'**
>   Constants for relation predicates

class idaes.dmf.resource.**ProgLangExt**
>   Helper class to map from file extensions to names of the programming language.

idaes.dmf.resource.**RESOURCE_TYPES = {'code', 'data', 'experiment', 'flowsheet', 'json', 'no**
>   Constants for resource 'types'

class idaes.dmf.resource.**Resource**(*value: dict = None*, *type_: str = None*)
>   Core object for the Data Management Framework.

>   **ID_FIELD = 'id_'**
>   >   Identifier field name constant

>   **ID_LENGTH = 32**
>   >   Full-length of identifier

>   **exception InferResourceTypeError**

>   **exception LoadResourceError**(*inferred_type*, *msg*)

>   **TYPE_FIELD = 'type'**
>   >   Resource type field name constant

>   **data**
>   >   Get JSON data for this resource.

>   **classmethod from_file**(*path: str*, *as_type: str = None*, *strict: bool = True*, *do_copy: bool = True*) → idaes.dmf.resource.Resource
>   >   Import resource from a file.

>   >   **Parameters**

>   >   - **path** – File path

>   >   - **as_type** – Resource type. If None/empty, then inferred from path.

>   >   - **strict** – If True, fail when file extension and contents don't match. If False, always fall through to generic resource.

>   >   - **do_copy** – If True (the default), copy the files; else do not

>   >   **Raises**

>   >   - *InferResourceTypeError* – if resource type does not match inferred/specified

>   >   - *LoadResourceError* – if resource import failed

>   **get_datafiles**(*mode='r'*)
>   >   Generate readable file objects for 'datafiles' in resource.

>   >   **Parameters mode** (*str*) – Mode for open()

>   >   **Returns** Generates file objects.

>   >   **Return type** generator

>   **id**
>   >   Get resource identifier.

>   **name**
>   >   Get resource name (first alias).

>   **type**
>   >   Get resource type.

class idaes.dmf.resource.**ResourceImporter**(*path: pathlib.Path*, *do_copy: bool = None*)
    Base class for Resource importers.

    create() → idaes.dmf.resource.Resource
        Factory method.

class idaes.dmf.resource.**SerializedResourceImporter**(*path*, *parsed*, ***kwargs*)

idaes.dmf.resource.**TY_CODE = 'code'**
    Resource type for source code

idaes.dmf.resource.**TY_DATA = 'data'**
    Resource type for generic data

idaes.dmf.resource.**TY_EXPERIMENT = 'experiment'**
    Resource type for experiments

idaes.dmf.resource.**TY_FLOWSHEET = 'flowsheet'**
    Resource type for a process flowsheet

idaes.dmf.resource.**TY_JSON = 'json'**
    Resource type for JSON data

idaes.dmf.resource.**TY_NOTEBOOK = 'notebook'**
    Resource type for a Jupyter Notebook

idaes.dmf.resource.**TY_OTHER = 'other'**
    Resource type for unspecified type of resource

idaes.dmf.resource.**TY_PROPERTY = 'propertydb'**
    Resource type for property data

idaes.dmf.resource.**TY_RESOURCE_JSON = 'resource_json'**
    Resource type for a JSON serialized resource

idaes.dmf.resource.**TY_SURRMOD = 'surrogate_model'**
    Resource type for a surrogate model

idaes.dmf.resource.**TY_TABULAR = 'tabular_data'**
    Resource type for tabular data

class idaes.dmf.resource.**TidyUnitData**(*data: dict = None*, *variables: List[T] = None*, *units:*
                                                              List[T] = None, *observations: List[T] = None*)
    Handle "tidy data" with per-column units.

    This can be used to convert from a simple dictionary/json representation like this:

```
{
  "variables": ["compound", "pressure"],
  "units": [null|None, "Pa"],
  "observations": [
    ["benzene", 4890000.0],
    ...etc..
  ]
}
```

    into a pandas DataFrame. A convenience method is provided for returning the data in a format easily dealt with
    when creating unit block parameters. Note that the keys in the preceding dictionary match the names of the
    parameters in the constructor (so you can pass this directly in as '**arg').

    units
        Units for each column, None where no units are defined

---

> **Type** list

table
> The observation data
>
> > **Type** pandas.DataFrame

param_data
> Data in a form easily consumed by unit block params.
>
> The dictionary returned is like { (key1, key2, ..): value }, where the keys are values from all columns except the last, and the value is the last column.

class idaes.dmf.resource.**Triple**(*subject*, *predicate*, *object*)
> Provide attribute access to an RDF subject, predicate, object triple

> **object**
> > Alias for field number 2

> **predicate**
> > Alias for field number 1

> **subject**
> > Alias for field number 0

idaes.dmf.resource.**create_relation**(*rel*)
> Create a relationship between two Resource instances.
>
> Relations are stored in both the *subject* and *object* resources, in the following way:

```
If R = (subject)S, (predicate)P, and (object)O
then store the following:
  In S.relations: {predicate: P, identifier:O.id, role:subject}
  In O.relations: {predicate: P, identifier:S.id, role:object}
```

> > **Parameters rel** (`Triple`) – Relation triple. The 'subject' and 'object' parts should be *Resource*, and the 'predicate' should be a simple string.
> >
> > **Returns** None
> >
> > **Raises** `ValueError` – if this relation already exists in the subject or object resource, or the predicate is not in the list of valid ones in RELATION_PREDICATES

idaes.dmf.resource.**create_relation_args**(**args*)
> Syntactic sugar to take 3 args instead of a Triple.

idaes.dmf.resource.**date_float**(*value*)
> Convert a date to a floating point seconds since the UNIX epoch.

idaes.dmf.resource.**identifier_str**(*value=None*, *allow_prefix=False*)
> Generate or validate a unique identifier.
>
> If generating, you will get a UUID in hex format

```
>>> identifier_str()
'...'
```

> If validating, anything that is not 32 lowercase letters or digits will fail.

```
>>> identifier_str('A' * 32)
Traceback (most recent call last):
ValueError: Bad format for identifier "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA":
must match regular expression "[0-9a-f]{32}"
```

> **Parameters value** (`str`) – If given, validate that it is a 32-byte str If not given or None, set new random value.
>
> **Raises** ValuError, if a value is given, and it is invalid.

idaes.dmf.resource.**schema_as_yaml**()

> Export resource schema as YAML suitable for embedding into, e.g., an OpenAPI spec.

idaes.dmf.resource.**triple_from_resource_relations**(*id_*, *rrel*)

> Create a Triple from one entry in resource['relations'].
>
> **Parameters**
>
> > - **id** (`str`) – Identifier of the containing resource.
> > - **rrel** (`dict`) – Stored relation with three keys, see *create_relation()*.
>
> **Returns** A triple
>
> **Return type** *Triple*

idaes.dmf.resource.**version_list**(*value*)

> Semantic version.
>
> Three numeric identifiers, separated by a dot. Trailing non-numeric characters allowed.
>
> Inputs, string or tuple, may have less than three numeric identifiers, but internally the value will be padded with zeros to always be of length four.
>
> A leading dash or underscore in the trailing non-numeric characters is removed.
>
> Some examples of valid inputs and how they translate to 4-part versions:

```
>>> version_list('1')
[1, 0, 0, '']
>>> version_list('1.1')
[1, 1, 0, '']
>>> version_list('1a')
[1, 0, 0, 'a']
>>> version_list('1.12.1')
[1, 12, 1, '']
>>> version_list('1.12.13-1')
[1, 12, 13, '1']
```

> Some examples of invalid inputs:

```
>>> for bad_input in ('rc3',        # too short
...                   '1.a.1.',     # non-number in middle
...                   '1.12.13.x'   # too long
...       ):
...       try:
...           version_list(bad_input)
...       except ValueError:
...           print(f"failed: {bad_input}")
...
```

```
failed: rc3
failed: 1.a.1.
failed: 1.12.13.x
```

> **Returns** [major:int, minor:int, debug:int, release-type:str]
>
> **Return type** list

idaes.dmf.resourcedb module

Resource database.

class idaes.dmf.resourcedb.**ResourceDB**(*dbfile=None*, *connection=None*)

> A database interface to all the resources within a given DMF workspace.
>
> **delete**(*id_=None*, *idlist=None*, *filter_dict=None*, *internal_ids=False*)
>
> > Delete one or more resources with given identifiers.
> >
> > **Parameters**
> >
> > - **id** (*Union[str, int]*) – If given, delete this id.
> > - **idlist** (*list*) – If given, delete ids in this list
> > - **filter_dict** (*dict*) – If given, perform a search and delete ids it finds.
> > - **internal_ids** (*bool*) – If True, treat identifiers as numeric (internal) identifiers. Otherwise treat them as resource (string) indentifiers.
> >
> > **Returns** (list[str]) Identifiers
>
> **find**(*filter_dict*, *id_only=False*, *flags=0*)
>
> > Find and return records based on the provided filter.
> >
> > **Parameters**
> >
> > - **filter_dict** (*dict*) – Search filter. For syntax, see docs in dmf.DMF.find().
> > - **id_only** (*bool*) – If true, return only the identifier of each resource; otherwise a Resource object is returned.
> > - **flags** (*int*) – Flag values for, e.g., regex searches
> >
> > **Returns** generator of int|Resource, depending on the value of *id_only*
>
> **find_one**(**args*, ***kwargs*)
>
> > Same as *find()*, but returning only first value or None.
>
> **find_related**(*id_*, *filter_dict=None*, *outgoing=True*, *maxdepth=0*, *meta=None*)
>
> > Find all resources connected to the identified one.
> >
> > **Parameters**
> >
> > - **id** (*str*) – Unique ID of target resource.
> > - **filter_dict** (*dict*) – Filter to these resources
> > - **outgoing** –
> > - **maxdepth** –
> > - **meta** (*List[str]*) – Metadata fields to extract

> **Returns** Generator of (depth, relation, metadata)
>
> **Raises** KeyError if the resource is not found.

get(*identifier*)
> Get a resource by identifier.
>
> > **Parameters identifier** – Internal identifier
> >
> > **Returns** (Resource) A resource or None

put(*resource*)
> Put this resource into the database.
>
> > **Parameters resource** (Resource) – The resource to add
> >
> > **Returns** None
> >
> > **Raises** errors.DuplicateResourceError – If there is already a resource in the database with the same "id".

update(*id_*, *new_dict*)
> Update the identified resource with new values.
>
> > **Parameters**
> >
> > - **id** (int) – Identifier of resource to update
> > - **new_dict** (dict) – New dictionary of resource values
> >
> > **Returns** None
> >
> > **Raises**
> >
> > - ValueError – If new resource is of wrong type
> > - KeyError – If old resource is not found

idaes.dmf.surrmod module

Surrogate modeling helper classes and functions. This is used to run ALAMO on property data.

class idaes.dmf.surrmod.**SurrogateModel**(*experiment*, ***kwargs*)
> Run ALAMO to generate surrogate models.
>
> Automatically track the objects in the DMF.
>
> Example:

```
model = SurrogateModel(dmf, simulator='linsim.py')
rsrc = dmf.fetch_one(1) # get resource ID 1
data = rsrc.property_table.data
model.set_input_data(data, ['temp'], 'density')
results = model.run()
```

> **PARAM_DATA_KEY = 'parameters'**
> > Key in resource 'data' for params
>
> **run**(***kwargs*)
> > Run ALAMO.
> >
> > > **Parameters **kwargs** – Additional arguments merged with those passed to the class constructor. Any duplicate values will override the earlier ones.
> > >
> > > **Returns** The dictionary returned from alamopy.doalamo()

Return type dict

set_input_data(*data*, *x_colnames*, *z_colname*)

Set input from provided dataframe or property data.

Parameters

- **data** (`PropertyData|pandas.DataFrame`) – Input data

- **x_colnames** (`List[str]|str`) – One or more column names for parameters

- **z_colname** (`str`) – Column for response variable

Returns None

Raises KeyError – if columns are not found in data

set_input_data_np(*x*, *z*, *xlabels=None*, *zlabel='z'*)

Set input data from numpy arrays.

Parameters

- **x** (`arr`) – Numpy array with parameters

- **xlabels** (`List[str]`) – List of labels for x

- **zlabel** (`str`) – Label for z

- **z** (`arr`) – Numpy array with response variables

Returns None

set_validation_data(*data*, *x_colnames*, *z_colname*)

Set validation data from provided data.

Parameters

- **data** (`PropertyData|pandas.DataFrame`) – Input data

- **x_colnames** (`List[str]|str`) – One or more column names for parameters

- **z_colname** (`str`) – Column for response variable

Returns None

Raises KeyError – if columns are not found in data

set_validation_data_np(*x*, *z*, *xlabels=None*, *zlabel='z'*)

Set input data from numpy arrays.

Parameters

- **x** (`arr`) – Numpy array with parameters

- **xlabels** (`List[str]`) – List of labels for x

- **zlabel** (`str`) – Label for z

- **z** (`arr`) – Numpy array with response variables

Returns None

idaes.dmf.tabular module

Tabular data handling

class `idaes.dmf.tabular.`**`Column`**(*name*, *data*)
>   Generic, abstract column

class `idaes.dmf.tabular.`**`Fields`**
>   Constants for field names.

>   **`DATA_NAME = 'name'`**
>   >   Keys for data mapping

class `idaes.dmf.tabular.`**`Metadata`**(*values=None*)
>   Class to import metadata.

>   **`author`**
>   >   Publication author(s).

>   **`date`**
>   >   Publication date

>   **static `from_csv`**(*file_or_path*)
>   >   Import metadata from simple text format.

>   >   Example input:

```
Source,Han, J., Jin, J., Eimer, D.A., Melaaen, M.C.,"Density of        ␣
↪Water(1) + Monoethanolamine(2) + CO2(3) from (298.15 to 413.15) K      ␣
↪ and Surface Tension of Water(1) + Monethanolamine(2) from (           ␣
↪303.15 to 333.15)K", J. Chem. Eng. Data, 2012, Vol. 57,            pg.␣
↪1095-1103"
Retrieval,"J. Morgan, date unknown"
Notes,r is MEA weight fraction in aqueous soln. (CO2-free basis)
```

>   >   **Parameters `file_or_path`**(`str or file`) – Input file

>   >   **Returns** (PropertyMetadata) New instance

>   **`info`**
>   >   Publication venue, etc.

>   **`source`**
>   >   Full publication info.

>   **`title`**
>   >   Publication title.

class `idaes.dmf.tabular.`**`Table`**(*data=None*, *metadata=None*)
>   Tabular data and metadata together (at last!)

>   **`as_dict`**()
>   >   Represent as a Python dictionary.

>   >   **Returns** (dict) Dictionary representation

>   **dump**(*fp*, ***kwargs*)
>   >   Dump to file as JSON. Convenience method, equivalent to converting to a dict and calling `json.dump()`.

>   >   **Parameters**

>   >   - **`fp`** (`file`) – Write output to this file

>   >   - **`**kwargs`** – Keywords passed to json.dump()

>   >   **Returns** see json.dump()

dumps(***kwargs*)

> Dump to string as JSON. Convenience method, equivalent to converting to a dict and calling `json.dumps()`.
>
> > **Parameters** ****kwargs** – Keywords passed to json.dumps()
> >
> > **Returns** (str) JSON-formatted data

classmethod load(*file_or_path*, *validate=True*)

> Create from JSON input.
>
> > **Parameters**
> >
> > - **file_or_path** (*file or str*) – Filename or file object from which to read the JSON-formatted data.
> >
> > - **validate** (*bool*) – If true, apply validation to input JSON data.
>
> Example input:

```
{
    "meta": [{
        "datatype": "MEA",
        "info": "J. Chem. Eng. Data, 2009, Vol 54, pg. 3096-30100",
        "notes": "r is MEA weight fraction in aqueous soln.",
        "authors": "Amundsen, T.G., Lars, E.O., Eimer, D.A.",
        "title": "Density and Viscosity of Monoethanolamine + etc."
    }],
    "data": [
        {
            "name": "Viscosity Value",
            "units": "mPa-s",
            "values": [2.6, 6.2],
            "error_type": "absolute",
            "errors": [0.06, 0.004],
            "type": "property"
        }
    ]
}
```

class idaes.dmf.tabular.**TabularData**(*data*, *error_column=False*)

> Class representing tabular data that knows how to construct itself from a CSV file.
>
> You can build objects from multiple CSV files as well. See the property database section of the API docs for details, or read the code in add_csv() and the tests in idaes_dmf.propdb.tests.test_mergecsv.

as_arr()

> Export property data as arrays.
>
> > **Returns** (values[M,N], errors[M,N]) Two arrays of floats, each with M columns having N values.
> >
> > **Raises** ValueError if the columns are not all the same length

as_list()

> Export the data as a list.
>
> Output will be in same form as data passed to constructor.
>
> > **Returns** (list) List of dicts

errors_dataframe()
> Get errors as a dataframe.
>
> > **Returns** Pandas dataframe for values.
> >
> > **Return type** pd.DataFrame
> >
> > **Raises** ImportError – If *pandas* or *numpy* were never successfully imported.

static from_csv(*file_or_path*, *error_column=False*)
> Import the CSV data.
>
> Expected format of the files is a header plus data rows.
>
> Header: Index-column, Column-name(1), Error-column(1), Column-name(2), Error-column(2), .. Data: <index>, <val>, <errval>, <val>, <errval>, ..
>
> Column-name is in the format "Name (units)"
>
> Error-column is in the format "<type> Error", where "<type>" is the error type.
>
> > **Parameters**
> >
> > - **file_or_path** (*file-like or str*) – Input file
> >
> > - **error_column** (*bool*) – If True, look for an error column after each value column. Otherwise, all columns are assumed to be values.
> >
> > **Returns** New table of data
> >
> > **Return type** *TabularData*

get_column(*key*)
> Get an object for the given named column.
>
> > **Parameters key** (*str*) – Name of column
> >
> > **Returns** (TabularColumn) Column object.
> >
> > **Raises** KeyError – No column by that name.

get_column_index(*key*)
> Get an index for the given named column.
>
> > **Parameters key** (*str*) – Name of column
> >
> > **Returns** (int) Column number.
> >
> > **Raises** KeyError – No column by that name.

names()
> Get column names.
>
> > **Returns** List of column names.
> >
> > **Return type** list[str]

num_columns
> Number of columns in this table.
>
> A "column" is defined as data + error. So if there are two columns of data, each with an associated error column, then *num_columns* is 2 (not 4).
>
> > **Returns** Number of columns.
> >
> > **Return type** int

num_rows
> Number of rows in this table.
>
> obj.num_rows is a synonym for len(obj)
>
>> **Returns** Number of rows.
>>
>> **Return type** int

values_dataframe()
> Get values as a dataframe.
>
>> **Returns** (pd.DataFrame) Pandas dataframe for values.
>>
>> **Raises** ImportError – If *pandas* or *numpy* were never successfully imported.

class idaes.dmf.tabular.**TabularObject**
> Abstract Property data class.

as_dict()
> Return Python dict representation.

idaes.dmf.userapi module

Data Management Framework high-level functions.

idaes.dmf.userapi.**find_property_packages**(*dmf*, *properties=None*)
> Find all property packages matching provided criteria.
>
> Return the matching packages as a generator.
>
>> **Parameters**
>>
>> - **dmf** (DMF) – Data Management Framework instance
>> - **properties** (*List[str]*) – Names of properties that must be present in the returned packages.
>>
>> **Returns**
>>
>>> **Each object has the property** data (properties and default units) in its *.data* attribute.
>>
>> **Return type** Generator[*idaes.dmf.resource.Resource*]

idaes.dmf.userapi.**get_workspace**(*path=''*, *name=None*, *desc=None*, *create=False*, *errs=None*,
> ***kwargs*)
> Create or load a DMF workspace.
>
> If the DMF constructor, throws an exception, this catches it and prints the error to the provided stream (or stdout).
>
> See DMF for details on arguments.
>
>> **Parameters**
>>
>> - **path** (*str*) – Path to workspace.
>> - **name** (*str*) – Name to be used for workspace.
>> - **desc** (*str*) – Longer description of workspace.
>> - **create** (*bool*) – If the path to the workspace does not exist, this controls whether to create it.
>> - **errs** (*object*) – Stream for errors, stdout is used if None
>>
>> **Returns** New instance, or None if it failed.

> **Return type** *DMF*

idaes.dmf.util module

Utility functions.

class idaes.dmf.util.**ColorTerm**(*enabled=True*)

 For colorized printing, a very simple wrapper that allows colorama objects, or nothing, to be used.

 class EmptyStr

  Return an empty string on any attribute requested.

class idaes.dmf.util.**TempDir**(**args*)

 Simple context manager for mkdtemp().

idaes.dmf.util.**datetime_timestamp**(*v*)

 Get numeric timestamp. This will work under both Python 2 and 3.

  Parameters v (`datetime.datetime`) – Date/time value

  Returns (float) Floating point timestamp

idaes.dmf.util.**get_file**(*file_or_path*, *mode='r'*)

 Open a file for reading, or simply return the file object.

idaes.dmf.util.**get_module_author**(*mod*)

 Find and return the module author.

  Parameters mod (`module`) – Python module

  Returns (str) Author string or None if not found

  Raises nothing

idaes.dmf.util.**get_module_version**(*mod*)

 Find and return the module version.

 Version must look like a semantic version with <a>.<b>.<c> parts; there can be arbitrary extra stuff after the <c>. For example:

```
1.0.12
0.3.6
1.2.3-alpha-rel0
```

  Parameters mod (`module`) – Python module

  Returns (str) Version string or None if not found

  Raises ValueError if version is found but not valid

idaes.dmf.util.**is_jupyter_notebook**(*filename*, *check_contents=True*)

 See if this is a Jupyter notebook.

idaes.dmf.util.**is_python**(*filename*)

 See if this is a Python file. Do *not* import the source code.

idaes.dmf.util.**is_resource_json**(*filename*, *max_bytes=1000000.0*)

 Is this file a JSON Resource?

  Parameters

   • **filename** (`str`) – Full path to file

- **max_bytes** (*int*) – Max. allowable size. Since we try to parse the file, this saves potential DoS issues. Large files are a bad idea anyways, since this is metadata and may be stored somewhere with a record size limit (like MongoDB).

> **Returns** (bool) Whether it's a resource JSON file.

idaes.dmf.util.**mkdir_p**(*path*, **args*)
> Try to create all non-existent components of a path.

> **Parameters**

> - **path** (*str*) – Path to create

> - **args** – Other arguments for *os.mkdir()*.

> **Returns** None

> **Raises** os.error – Raised from *os.mkdir()*

idaes.dmf.util.**uuid_prefix_len**(*uuids*, *step=4*, *maxlen=32*)
> Get smallest multiple of *step* len prefix that gives unique values.

> The algorithm is not fancy, but good enough: build *sets* of the ids at increasing prefix lengths until the set has all ids (no duplicates). Experimentally this takes ~.1ms for 1000 duplicate ids (the worst case).

idaes.dmf.workspace module

Workspace classes and functions.

class idaes.dmf.workspace.**Fields**
> Workspace configuration fields.

class idaes.dmf.workspace.**Workspace**(*path*, *create=False*, *add_defaults=False*, *html_paths=None*)
> DMF Workspace.

> In essence, a workspace is some information at the root of a directory tree, a database (currently file-based, so also in the directory tree) of *Resources*, and a set of files associated with these resources.

> **Workspace Configuration**

> When the DMF is initialized, the workspace is given as a path to a directory. In that directory is a special file named config.yaml, that contains metadata about the workspace. The very existence of a file by that name is taken by the DMF code as an indication that the containing directory is a DMF workspace:

```
/path/to/dmf: Root DMF directory
 |
 +- config.yaml: Configuration file
 +- resourcedb.json: Resource metadata "database" (uses TinyDB)
 +- files: Data files for all resources
```

> The configuration file is a YAML formatted file

> The configuration file defines the following key/value pairs:

> **_id** Unique identifier for the workspace. This is auto-generated by the library, of course.

> **name** Short name for the workspace.

> **description** Possibly longer text describing the workspace.

> **created** Date at which the workspace was created, as string in the ISO8601 format.

> **modified** Date at which the workspace was last modified, as string in the ISO8601 format.

htmldocs Full path to the location of the built (not source) Sphinx HTML documentation for the *idaes_dmf* package. See DMF Help Configuration for more details.

There are many different possible "styles" of formatting a list of values in YAML, but we prefer the simple block-indented style, where the key is on its own line and the values are each indented with a dash:

```
_id: fe5372a7e51d498fb377da49704874eb
created: '2018-07-16 11:10:44'
description: A bottomless trashcan
modified: '2018-07-16 11:10:44'
name: Oscar the Grouch's Home
htmldocs:
- '{dmf_root}/doc/build/html/dmf'
- '{dmf_root}/doc/build/html/models'
```

Any paths in the workspace configuration, e.g., for the "htmldocs", can use two special variables that will take on values relative to the workspace location. This avoids hardcoded paths and makes the workspace more portable across environments. `{ws_root}` will be replaces with the path to the workspace directory, and `{dmf_root}` will be replaced with the path to the (installed) DMF package.

The *config.yaml* file will allow keys and values it does not know about. These will be accessible, loaded into a Python dictionary, via the `meta` attribute on the *Workspace* instance. This may be useful for passing additional user-defined information into the DMF at startup.

CONF_CREATED = 'created'
    Configuration field for created date

CONF_DESC = 'description'
    Configuration field for description

CONF_MODIFIED = 'modified'
    Configuration field for modified date

CONF_NAME = 'name'
    Configuration field for name

ID_FIELD = '_id'
    Name of ID field

WORKSPACE_CONFIG = 'config.yaml'
    Name of configuration file placed in WORKSPACE_DIR

configuration_file
    Configuration file path.

get_doc_paths()
    Get paths to generated HTML Sphinx docs.

> **Returns** (list) Paths or empty list if not found.

meta
    Get metadata.

    This reads and parses the configuration. Therefore, one way to force a config refresh is to simply refer to this property, e.g.:

```
dmf = DMF(path='my-workspace')
#  ... do stuff that alters the config ...
dmf.meta  # re-read/parse the config
```

> **Returns** (dict) Metadata for this workspace.

root
> Root path for this workspace. This is the path containing the configuration file.

set_doc_paths(*paths: List[str], replace: bool = False*)
> Set paths to generated HTML Sphinx docs.

> > **Parameters**
> >
> > - **paths** – New paths to add.
> >
> > - **replace** – If True, replace any existing paths. Otherwise merge new paths with existing ones.

set_meta(*values*, *remove=None*)
> Update metadata with new values.

> > **Parameters**
> >
> > - **values** ([*dict*](#)) – Values to add or change
> >
> > - **remove** ([*list*](#)) – Keys of values to remove.

wsid
> Get workspace identifier (from config file).

> > **Returns** Unique identifier.

> > **Return type** str

idaes.dmf.workspace.**find_workspaces**(*root*)
> Find workspaces at or below 'root'.

> > **Parameters root** ([*str*](#)) – Path to start at

> > **Returns** paths, which are all workspace roots.

> > **Return type** List[str]

idaes.dynamic package

Submodules

idaes.dynamic.pid_controller module

PID controller block

class idaes.dynamic.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

> **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns:** (PIDBlock) New instance

class idaes.dynamic.pid_controller.**PIDBlockData**(*component*)

> **build**()
>> Build the PID block

idaes.examples package

Subpackages

idaes.examples.alamo_python package

Submodules

idaes.examples.alamo_python.MEAPropertyRegression module

Property regression models

Ability to use DMF csv files to regress important parameters for literature based models and ALAMO models.

Tracking property metadata and new model metadata * See :meth:'PropertyData.from_csv', for expected format for data. * See :meth:'PropertyMetadata()' for the expected format for metadata.

Author: Marissa Engle

idaes.examples.alamo_python.ackley module

idaes.examples.alamo_python.branin module

idaes.examples.alamo_python.camel6 module

idaes.examples.alamo_python.examples module

idaes.examples.helmet_python package

Submodules

idaes.examples.helmet_python.H2OSetup module

idaes.examples.power_generation package

Subpackages

idaes.examples.power_generation.supercritical_steam_cycle package

Submodules

idaes.examples.power_generation.supercritical_steam_cycle.supercritical_steam_cycle module

This is an example supercritical pulverized coal (SCPC) power plant steam cycle model. This model doesn't represent any specific power plant, but it represents what could be considered a typical SCPC plant, producing around 620 MW gross. 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).

idaes.examples.power_generation.supercritical_steam_cycle.supercritical_steam_cycle.**create_**
> Create the flowsheet and add unit models. Fixing model inputs is done in a separate function to try to keep this fairly clean and easy to follow.

> > **Parameters None** –

> > **Returns** (ConcreteModel) Steam cycle model

idaes.examples.power_generation.supercritical_steam_cycle.supercritical_steam_cycle.**initial**

Initialize a mode from create_model(), set model inputs before initializing.

> > **Parameters**

> > > • **m** (*ConcreteModel*) – A Pyomo model from create_model()

> > > • **fileinput** (*str|None*) – File to load initialized model state from. If a file is supplied skip initialization routine. If None, initialize.

> > **Returns** A Pyomo solver object, that can be used to solve the model.

> > **Return type** solver

idaes.examples.power_generation.supercritical_steam_cycle.supercritical_steam_cycle.**main**(*in*
> *st*
> Create and initalize a model and solver

> > **Parameters None** –

> > **Returns** A tuple of a model and solver

idaes.examples.power_generation.supercritical_steam_cycle.supercritical_steam_cycle.**pfd_res**

Insert model results into the PFD and return a new SVG string, which can be displayed, further edited, or saved to a file.

> > **Parameters**

> > > • **m** (*ConcreteModel*) – A steam cycle model

- **df** (*Pandas DataFrame*) – Stream table

- **svg** (*FILE*, str, bytes*) – Origianl svg svg as either a file-like object, a string, or a byte array.

Returns SVG content.

Return type (str)

idaes.examples.power_generation.supercritical_steam_cycle.supercritical_steam_cycle.**set_mo**
Fix some variables and set values. Generally get the model ready to run in simulation mode (0 degrees of freedom).

Parameters m (*ConcreteModel*) – A Pyomo model from create_model()

Returns None

idaes.examples.properties package

Subpackages

idaes.examples.properties.Workshop_Module_2 package

Submodules

idaes.examples.properties.Workshop_Module_2.hda_ideal_VLE module

idaes.examples.properties.Workshop_Module_2.hda_reaction module

Property package for the hydrodealkylation of toluene to form benzene

class idaes.examples.properties.Workshop_Module_2.hda_reaction.**HDAReactionBlock**(**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 ProcessBlockData config

  Keys

    parameters A reference to an instance of the Reaction Parameter Block associated with this property package.

    state_block A reference to an instance of a StateBlock with which this reaction block should be associated.

    has_equilibrium Flag indicating whether equilibrium constraints should be constructed in this reaction block, **default** - True. **Valid values:** { **True** - ReactionBlock should enforce equilibrium constraints, **False** - ReactionBlock should not enforce equilibrium constraints.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (HDAReactionBlock) New instance

class idaes.examples.properties.Workshop_Module_2.hda_reaction.**HDAReactionBlockData**(*componen*
  An example reaction package for saponification of ethyl acetate

  build()
    Callable method for Block construction

  get_reaction_rate_basis()
    Method which returns an Enum indicating the basis of the reaction rate term.

class idaes.examples.properties.Workshop_Module_2.hda_reaction.**HDAReactionParameterBlock**(**a*
  ***
  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

      property_package Reference to associated PropertyPackageParameter object

      default_arguments Default arguments to use with Property Package

  - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

  > **Returns** (HDAReactionParameterBlock) New instance

class idaes.examples.properties.Workshop_Module_2.hda_reaction.**HDAReactionParameterData**(*con*
  Property Parameter Block Class

  Contains parameters and indexing sets associated with properties for superheated steam.

  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

class idaes.examples.properties.Workshop_Module_2.hda_reaction.**ReactionBlock**(**args*,
  **kwargs)
  This Class contains methods which should be applied to Reaction Blocks as a whole, rather than individual elements of indexed Reaction Blocks.

initialize(*outlvl=0, **kwargs*)
> Initialization routine for reaction package.

> **Keyword Arguments** `outlvl` – sets output level of initialization routine

> - 0 = no output (default)

> - 1 = report after each step

> **Returns** None

idaes.examples.ripe_python package

Submodules

idaes.examples.ripe_python.clc module

idaes.examples.ripe_python.crac module

idaes.examples.ripe_python.cracsim module

idaes.examples.ripe_python.isoT module

idaes.examples.ripe_python.isotsim module

idaes.examples.tutorials package

Submodules

idaes.examples.tutorials.Tutorial_1_Basic_Flowsheets module

Demonstration and test flowsheet for a dynamic flowsheet.

`idaes.examples.tutorials.Tutorial_1_Basic_Flowsheets.`**main**()
> Make the flowsheet object, fix some variables, and solve the problem

idaes.examples.tutorials.Tutorial_2_Basic_Flowsheet_Optimization module

Demonstration and test flowsheet for a dynamic flowsheet.

`idaes.examples.tutorials.Tutorial_2_Basic_Flowsheet_Optimization.`**main**()
> Make the flowsheet object, fix some variables, and solve the problem

idaes.examples.tutorials.Tutorial_3_Dynamic_Flowsheets module

Demonstration and test flowsheet for a dynamic flowsheet.

`idaes.examples.tutorials.Tutorial_3_Dynamic_Flowsheets.`**main**()
> Make the flowsheet object, fix some variables, and solve the problem

idaes.examples.workshops package

Subpackages

idaes.examples.workshops.Module_0_Welcome package

Submodules

idaes.examples.workshops.Module_0_Welcome.notebook_test_script module

idaes.examples.workshops.Module_1_Flash_Unit package

Submodules

idaes.examples.workshops.Module_1_Flash_Unit.BTX_ideal_VLE module

idaes.examples.workshops.Module_1_Flash_Unit.ideal_prop_pack_VLE module

Ideal property package with VLE calucations. Correlations to compute Cp_comp, h_comp and vapor pressure are obtained from "The properties of gases and liquids by Robert C. Reid" and "Perry's Chemical Engineers Handbook by Robert H. Perry". SI units.

class idaes.examples.workshops.Module_1_Flash_Unit.ideal_prop_pack_VLE.**IdealParameterData**(*o*
Property Parameter Block Class Contains parameters and indexing sets associated with properties for BTX system.

build()
Callable method for Block construction.

classmethod define_metadata(*obj*)
Define properties supported and units.

class idaes.examples.workshops.Module_1_Flash_Unit.ideal_prop_pack_VLE.**IdealStateBlock**(**arg*
**kw

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

  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 equi-
librium.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-
  Data 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 overide
  the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (IdealStateBlock) New instance

class idaes.examples.workshops.Module_1_Flash_Unit.ideal_prop_pack_VLE.**IdealStateBlockData**
An example property package for ideal VLE.

build ()
Callable method for Block construction.

calculate_bubble_point_pressure (*clear_components=True*)
"To compute the bubble point pressure of the mixture.

calculate_bubble_point_temperature (*clear_components=True*)
"To compute the bubble point temperature of the mixture.

calculate_dew_point_pressure (*clear_components=True*)
"To compute the dew point pressure of the mixture.

calculate_dew_point_temperature (*clear_components=True*)
"To compute the dew point temperature of the mixture.

define_state_vars ()
Define state vars.

get_enthalpy_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 ()
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.

idaes.examples.workshops.Module_1_Flash_Unit.workshoptools module

idaes.examples.workshops.Module_2_Flowsheet package

Submodules

idaes.examples.workshops.Module_2_Flowsheet.Module_2_HDA module

Demonstration of HDA flowsheet with optimization.

idaes.examples.workshops.Module_2_Flowsheet.hda_ideal_VLE module

Example ideal parameter block for the VLE calucations for a Benzene-Toluene-o-Xylene system.

class idaes.examples.workshops.Module_2_Flowsheet.hda_ideal_VLE.**HDAParameterBlock**(**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 ProcessBlockData config
>
>   **Keys**
>
>   > **default_arguments** Default arguments to use with Property Package
>
> - **initialize** (`dict`) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (HDAParameterBlock) New instance

class idaes.examples.workshops.Module_2_Flowsheet.hda_ideal_VLE.**HDAParameterData**(*component*)

> **build**()
>   Callable method for Block construction.
>
> **classmethod define_metadata**(*obj*)
>   Define properties supported and units.

class idaes.examples.workshops.Module_2_Flowsheet.hda_ideal_VLE.**IdealStateBlock**(**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 ProcessBlockData 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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (IdealStateBlock) New instance

class idaes.examples.workshops.Module_2_Flowsheet.hda_ideal_VLE.**IdealStateBlockData**(*componer*
An example property package for ideal VLE.

build()
Callable method for Block construction.

calculate_bubble_point_pressure(*clear_components=True*)
"To compute the bubble point pressure of the mixture.

calculate_bubble_point_temperature(*clear_components=True*)
"To compute the bubble point temperature of the mixture.

calculate_dew_point_pressure(*clear_components=True*)
"To compute the dew point pressure of the mixture.

calculate_dew_point_temperature(*clear_components=True*)
"To compute the dew point temperature of the mixture.

define_state_vars()
Define state vars.

get_enthalpy_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()
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.

idaes.examples.workshops.Module_2_Flowsheet.hda_reaction module

Property package for the hydrodealkylation of toluene to form benzene

class idaes.examples.workshops.Module_2_Flowsheet.hda_reaction.**HDAReactionBlock**(**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 ProcessBlockData config
>
>   **Keys**

parameters A reference to an instance of the Reaction Parameter Block associated with this property package.

state_block A reference to an instance of a StateBlock with which this reaction block should be associated.

has_equilibrium Flag indicating whether equilibrium constraints should be constructed in this reaction block, **default** - True. **Valid values:** { **True** - ReactionBlock should enforce equilibrium constraints, **False** - ReactionBlock should not enforce equilibrium constraints.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

 Returns (HDAReactionBlock) New instance

class idaes.examples.workshops.Module_2_Flowsheet.hda_reaction.**HDAReactionBlockData**(*componen*
  An example reaction package for saponification of ethyl acetate

 build()
  Callable method for Block construction

 get_reaction_rate_basis()
  Method which returns an Enum indicating the basis of the reaction rate term.

class idaes.examples.workshops.Module_2_Flowsheet.hda_reaction.**HDAReactionParameterBlock**(**a*
                                          **

 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

 property_package Reference to associated PropertyPackageParameter object

 default_arguments Default arguments to use with Property Package

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

 Returns (HDAReactionParameterBlock) New instance

class idaes.examples.workshops.Module_2_Flowsheet.hda_reaction.**HDAReactionParameterData**(*con*
  Property Parameter Block Class

 Contains parameters and indexing sets associated with properties for superheated steam.

 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

class idaes.examples.workshops.Module_2_Flowsheet.hda_reaction.**ReactionBlock**(**args*, ***kwargs*)

> This Class contains methods which should be applied to Reaction Blocks as a whole, rather than individual elements of indexed Reaction Blocks.

initialize(*outlvl=0*, ***kwargs*)

> Initialization routine for reaction package.
>
> > **Keyword Arguments outlvl** – sets output level of initialization routine
> >
> > - 0 = no output (default)
> >
> > - 1 = report after each step
> >
> > **Returns** None

idaes.examples.workshops.Module_2_Flowsheet.workshoptools module

idaes.examples.workshops.Module_3_Custom_Unit_Model package

Submodules

idaes.examples.workshops.Module_3_Custom_Unit_Model.methanol_param_VLE module

Example property package for the VLE calucations for the methanol synthesis problem from Turkay & Grossmann. The parameters and correlations are from the paper.

class idaes.examples.workshops.Module_3_Custom_Unit_Model.methanol_param_VLE.**PhysicalParame**

> **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**
>
>   > **default_arguments** Default arguments to use with Property Package
>   >
>   > **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,}
>   >
>   > **Cp** Value for the constant pressure heat capacity, **default** = 0.035 MJ/(kgmol K)

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (PhysicalParameterBlock) New instance

class idaes.examples.workshops.Module_3_Custom_Unit_Model.methanol_param_VLE.**PhysicalParame**
Property Parameter Block Class.

build()
Callable method for Block construction.

classmethod define_metadata(*obj*)
Define properties supported and units.

idaes.examples.workshops.Module_3_Custom_Unit_Model.methanol_state_block_VLE module

Property package for ideal VLE calucations for the methanol synthesis problem. Correlations from Turkay and Grossmann paper. See Latex files for details.

class idaes.examples.workshops.Module_3_Custom_Unit_Model.methanol_state_block_VLE.**IdealSta**

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

  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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (IdealStateBlock) New instance

class idaes.examples.workshops.Module_3_Custom_Unit_Model.methanol_state_block_VLE.**StateBlo**
An example property package for ideal VLE.

build()
    Callable method for Block construction.

define_state_vars()
    Define state vars.

get_enthalpy_density_terms(*p*)
    Create enthalpy density terms.

get_enthalpy_flow_terms(*p*)
    Create enthalpy flow terms [MJ/s].

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.

idaes.functions package

idaes.plugins package

Submodules

idaes.plugins.variable_replace module

class idaes.plugins.variable_replace.**ReplaceVariables**(***kwds*)
    Replace variables in a model or block with other variables.

    Keyword arguments below are specified for the apply_to(instance, **kwargs) method.

        Keyword Arguments substitute – List-like of tuples where the first item in a tuple is a Pyomo variable to be replaced and the second item in the tuple is a Pyomo variable to replace it with. This transformation is not reversible.

idaes.property_models package

Subpackages

idaes.property_models.activity_coeff_models package

Submodules

idaes.property_models.activity_coeff_models.BTX_activity_coeff_VLE module

Example property package for the VLE calucations for a Benzene-Toluene-o-Xylene system. If using the activity coefficient models (NRTL or Wilson), the user is expected to provide the paramters necessary for these models. Please

note that these parameters are declared as variables here to allow for use in a parameter estimation problem if the VLE data is available.

class idaes.property_models.activity_coeff_models.BTX_activity_coeff_VLE.**BTXParameterBlock**

> **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**
>>> **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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (BTXParameterBlock) New instance

class idaes.property_models.activity_coeff_models.BTX_activity_coeff_VLE.**BTXParameterData**(

> **build**()
>> Callable method for Block construction.

idaes.property_models.activity_coeff_models.activity_coeff_prop_pack module

Ideal gas + Ideal/Non-ideal liquid property package.

VLE calucations assuming an ideal gas for the gas phase. For the liquid phase, options include ideal liquid or non-ideal liquid using an activity coefficient model; options include Non Random Two Liquid Model (NRTL) or the Wilson model to compute the activity coefficient. This property package supports the following combinations for gas-liquid mixtures: 1. Ideal (vapor) - Ideal (liquid) 2. Ideal (vapor) - NRTL (liquid) 3. Ideal (vapor) - Wilson (liquid)

This property package currently supports the flow_mol, temperature, pressure and mole_frac_comp as state variables (mole basis). Support for other combinations will be available in the future.

Please note that the parameters required to compute the activity coefficient for the component needs to be provided by the user in the parameter block or can be estimated by the user if VLE data is available. Please see the documentation for more details.

SI units.

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.

class idaes.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffPa**

> **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**
>
>   **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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (ActivityCoeffParameterBlock) New instance

class idaes.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffPa**
Property Parameter Block Class Contains parameters and indexing sets associated with properties for BTX system.

build()
Callable method for Block construction.

classmethod define_metadata(*obj*)
Define properties supported and units.

---

class idaes.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffSt**

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

    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`) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

    Returns (ActivityCoeffStateBlock) New instance

class idaes.property_models.activity_coeff_models.activity_coeff_prop_pack.**ActivityCoeffSt**
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.

idaes.property_models.activity_coeff_models.methane_combustion_ideal module

Example property package for the VLE calucations for a Benzene-Toluene-o-Xylene system. If using the activity coefficient models (NRTL or Wilson), the user is expected to provide the paramters necessary for these models. Please note that these parameters are declared as variables here to allow for use in a parameter estimation problem if the VLE data is available.

class idaes.property_models.activity_coeff_models.methane_combustion_ideal.**MethaneParameter**

> **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**
> > >
> > > > **default_arguments** Default arguments to use with Property Package
> > > >
> > > > **activity_coeff_model** Methane combustion supports ideal gas only
> > > >
> > > > **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** Methane combustion supports ideal gas only
> >
> > - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> > **Returns** (MethaneParameterBlock) New instance

class idaes.property_models.activity_coeff_models.methane_combustion_ideal.**MethaneParameter**

> **build**()
>   Callable method for Block construction.

idaes.property_models.cubic_eos package

Submodules

idaes.property_models.cubic_eos.BT_PR module

Example Peng-Robinson parameter block for the VLE calucations for a benzene-toluene system.

Unless otherwise noted, parameters are from: "The Properties of Gases and Liquids, 4th Edition", Reid, Prausnitz and Poling, McGraw-Hill, 1987

class idaes.property_models.cubic_eos.BT_PR.**BTParameterBlock**(**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 ProcessBlockData config

    Keys

    default_arguments Default arguments to use with Property Package

    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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

    Returns (BTParameterBlock) New instance

class idaes.property_models.cubic_eos.BT_PR.**BTParameterData**(*component*)

> **build**()
>   Callable method for Block construction.

idaes.property_models.cubic_eos.cubic_prop_pack module

General Cubic Equation of State property package with VLE calucations. Cubic formulation and pure component property correlations from: "The Properties of Gases and Liquids, 4th Edition", Reid, Prausnitz and Poling, McGraw-Hill, 1987

Smooth Vapor-Liquid Equilibrium formulation from: "A Smooth, Square Flash Formulation for Equation-Oriented Flowsheet Optimization", Burgard et al., Proceedings of the 13 the International Symposium on Process Systems Engineering – PSE 2018, July 1-5, 2018, San Diego

All results have been cross-referenced against other sources.

class idaes.property_models.cubic_eos.cubic_prop_pack.**CubicEoS**
    An enumeration.

class idaes.property_models.cubic_eos.cubic_prop_pack.**CubicParameterBlock**(**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 ProcessBlockData config

    Keys

    default_arguments Default arguments to use with Property Package

> > > **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`) – ProcessBlockData config for individual elements. Keys are Block-Data 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** (CubicParameterBlock) New instance

class idaes.property_models.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.property_models.cubic_eos.cubic_prop_pack.**CubicStateBlock**(**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 ProcessBlockData 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`) – ProcessBlockData config for individual elements. Keys are Block-Data 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.property_models.cubic_eos.cubic_prop_pack.**CubicStateBlockData**(*component*)

> 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.

idaes.property_models.cubic_eos.cubic_prop_pack.**cubic_roots_available**()
:   Make sure the compiled cubic root functions are available. Yes, in Windows the .so extention is still used.

idaes.property_models.examples package

Submodules

idaes.property_models.examples.saponification_reactions module

Example property package for the saponification of Ethyl Acetate with NaOH Assumes dilute solutions with properties of H2O.

class idaes.property_models.examples.saponification_reactions.**ReactionBlock**(**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 ProcessBlockData config
>
>   **Keys**
>
>   **parameters** A reference to an instance of the Reaction Parameter Block associated with this property package.
>
>   **state_block** A reference to an instance of a StateBlock with which this reaction block should be associated.
>
>   **has_equilibrium** Flag indicating whether equilibrium constraints should be constructed in this reaction block, **default** - True. **Valid values:** { **True** - ReactionBlock should

enforce equilibrium constraints, **False** - ReactionBlock should not enforce equilibrium constraints.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

   Returns (ReactionBlock) New instance

class idaes.property_models.examples.saponification_reactions.**ReactionBlockData**(*component*)
   An example reaction package for saponification of ethyl acetate

   build()
      Callable method for Block construction

   get_reaction_rate_basis()
      Method which returns an Enum indicating the basis of the reaction rate term.

class idaes.property_models.examples.saponification_reactions.**ReactionParameterData**(*componer*
   Property Parameter Block Class

   Contains parameters and indexing sets associated with properties for superheated steam.

   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

class idaes.property_models.examples.saponification_reactions.**SaponificationReactionParamet**

   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

         property_package Reference to associated PropertyPackageParameter object

         default_arguments Default arguments to use with Property Package

   - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 (SaponificationReactionParameterBlock) New instance

idaes.property_models.examples.saponification_thermo module

Example property package for the saponification of Ethyl Acetate with NaOH Assumes dilute solutions with properties of H2O.

class idaes.property_models.examples.saponification_thermo.**PhysicalParameterData**(*component*)
    Property Parameter Block Class

    Contains parameters and indexing sets associated with properties for superheated steam.

    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

class idaes.property_models.examples.saponification_thermo.**SaponificationParameterBlock**(**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 ProcessBlockData config

                Keys

                    default_arguments Default arguments to use with Property Package

            • **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

        Returns (SaponificationParameterBlock) New instance

class idaes.property_models.examples.saponification_thermo.**SaponificationStateBlock**(**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 ProcessBlockData 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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (SaponificationStateBlock) New instance

class idaes.property_models.examples.saponification_thermo.**SaponificationStateBlockData**(*con*
 An example property package for properties for saponification of ethyl acetate

> **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**()
> 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**(*p*)
> Method which returns a valid expression for enthalpy density to use in the energy balances.

> **get_enthalpy_flow_terms**(*p*)
> Method which returns a valid expression for enthalpy flow to use in the energy balances.

> **get_material_density_terms**(*p*, *j*)
> Method which returns a valid expression for material density to use in the material balances .

> **get_material_flow_basis**()
> Method which returns an Enum indicating the basis of the material flow term.

> **get_material_flow_terms**(*p*, *j*)
> Method which returns a valid expression for material flow to use in the material balances.

> **model_check**()
> Model checks for property block

Submodules

idaes.property_models.iapws95 module

IDAES IAPWS-95 Steam properties

Dropped all critical enhancments and non-analytic terms ment to improve accruacy near the critical point. These tend to cause singularities in the equations, and it is assumend that we will try to avoid operating very close to the critical point.

References: (some of this is only used in the C++ part)

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

class idaes.property_models.iapws95.**Iapws95ParameterBlock**(**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 ProcessBlockData 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 independely. The LG option presents as two sperate 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}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Iapws95ParameterBlock) New instance

class idaes.property_models.iapws95.**Iapws95ParameterBlockData**(*component*)

build()
General build method for PropertyParameterBlocks. Inheriting models should call super().build.

Parameters None –

Returns None

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

class idaes.property_models.iapws95.**Iapws95StateBlock**(**args*, ***kwargs*)
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 ProcessBlockData 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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Iapws95StateBlock) New instance

class idaes.property_models.iapws95.**Iapws95StateBlockData**(*component*)
This is a property package for calculating thermophysical properties of water

build(**args*)
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()
>   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(*p*)
>   Method which returns a valid expression for enthalpy density to use in the energy balances.

get_enthalpy_flow_terms(*p*)
>   Method which returns a valid expression for enthalpy flow to use in the energy balances.

get_material_density_terms(*p*, *j*)
>   Method which returns a valid expression for material density to use in the material balances .

get_material_flow_terms(*p*, *j*)
>   Method which returns a valid expression for material flow to use in the material balances.

class idaes.property_models.iapws95.**PhaseType**
>   Ways to present phases to the framework

class idaes.property_models.iapws95.**StateVars**
>   State variable set options

idaes.property_models.iapws95.**htpx**(*T*, *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 one (and only one) of arguments P and x.

>>   **Parameters**
>>   - **T** – Temperature [K] (between 200 and 3000)
>>   - **P** – Pressure [Pa] (between 1 and 1e9), None if saturated steam
>>   - **x** – Vapor fraction [mol vapor/mol total] (between 0 and 1), None if
>>   - **or subcooled** (*superheated*) –

>>   **Returns** Total molar enthalpy [J/mol].

idaes.property_models.iapws95.**iapws95_available**()
>   Make sure the compiled IAPWS-95 functions are available. Yes, in Windows the .so extention is still used.

idaes.surrogate package

Subpackages

idaes.surrogate.alamopy package

Submodules

idaes.surrogate.alamopy.allcard module

idaes.surrogate.alamopy.almconfidence module

idaes.surrogate.alamopy.almerror module

Errors

exception idaes.surrogate.alamopy.almerror.**AlamoError**

exception idaes.surrogate.alamopy.almerror.**AlamoInputError**(*msg*)

idaes.surrogate.alamopy.almpickle module

idaes.surrogate.alamopy.almplot module

Plot doalamo output including confidence intervals if they are calculated.

idaes.surrogate.alamopy.almpywriter module

idaes.surrogate.alamopy.almwriter module

idaes.surrogate.alamopy.doalamo module

Run ALAMO.

idaes.surrogate.alamopy.doalamo.**addBasisConstraints**(*groups_constraints*)
> NMT (no more than), ATL (at least), REQ (required if main group), XCL (exclude) format: group-id output_id constraint_type integer_parameter

idaes.surrogate.alamopy.doalamo.**addBasisGroup**(*type_of_function*, *input_indices=''*, *powers=''*)
> Include the checks

idaes.surrogate.alamopy.doalamo.**addBasisGroups**(*groups*)
> format: index_num Type_of_function Member_indices(indice of the input) <pow> types: LIN, LOG, EXP, SIN, COS, MONO, MULTI2, MULTI3, RATIO, RBf, CUST, CONST Only one CONST One input: MONO, EXP, LOG, SIN, COS - integer from 1- NINPUTs, (-1) - all inputs multiple input: Multi2, Multi3, ratio Need <pow> for MONO, Multi2, Multi3, Ratio, (-1111) - all powers

idaes.surrogate.alamopy.doalamo.**addCustomConstraints**(*constraint_list*, ***kwargs*)
> constraint args = CRTOL, CRNINITIAL, CRMAXITER, CRNVIOL, CRNTRIALS

idaes.surrogate.alamopy.doalamo.**alamo**(*xdata*, *zdata*, ***kwargs*)
> [almmodel] = doalamo(xdata,zdata, xvaldata, zvaldata,addopt=vals)

> > **Parameters**

> > - **xdata** (*numpy.array or* `list[real]`) –
> > - **zdata** (*numpy.array or* `list[real]`) –
> > - **kwargs** –

> > > **Additional options may be specified and will be applied** to the .alm

> > > – example - monomialpower=(1,2,3,4)
> > > – xlabels : labels given to input variables
> > > – zlabels : labels given to outputs

- xval : validaiton data for alamo

- zval : response validation data for alamo

- modeler : modeler value used in alamo

- solvemip : force alamo to solve mip if gams is availible

- linfcns : 0-1 option to include linear transformations

- expfcns : 0-1 option to include exponential transformations

- logfcns : 0-1 option to include logarithmic transformations

- sinfcns : 0-1 option to include sine transformations

- cosfcns : 0-1 option to include cosine transformations

- monomialpower : list of monomial powers

- multi2power : list of binomial powers

- multi3power : list of trinomials

- ratiopower : list of ratio powers

- screener : screening method

- almname : specify a name for the .alm file

- savescratch : saves .alm and .lst

- savetrace : saves trace file

- **expandoutput** [add a key to the output dictionary for the output] (must be on for inputs(outputs?#Engle)>1)

- **almopt** [direct text appending] the option almopt=<file> will append a file to the end of the .alm and can be used to facilitate direct access to the .alm (no current checks)

- loo : leave one out evaluation

- lmo : leave many out evaluation

Returns

    An ALAMO model with the following keys

- 'model' : algebraic form of model

- 'f(model)' : a callable lambda function

- **Syntac is depended on expandout**

    syntax => almmodel['f(model)']['out'](inputs,sep,by,comma)
        almmodel['f(model)'](inputs,sep,by,comma)

- 'ssr' : SSE on training set provided

- 'R2' : R2 on training set provided

- 'ssrval' : SSE on testing set if provided

- 'R2val' : R2 on testing set if provided

Return type dict

idaes.surrogate.alamopy.doalamo.**buildSimWrapper**(*data*, *debug*)
    Builds an executable simulator to sample for data

> Parameters

>> • **data** – shared alamo data options

>> • **debug** – Additional options may be specified and will be applied to the .alm

idaes.surrogate.alamopy.doalamo.**checkForSampledData**(*data*, *debug*)
  Check to see if data has been sampled and update ndata

> Parameters

>> • **data** – shared alamo data options

>> • **debug** – Additional options may be specified and will be applied to the .alm

idaes.surrogate.alamopy.doalamo.**checkinput**(*data*, *debug*, *xdata*, *zdata*, *vargs*, *kwargs*)
  Check the input data into doalamo for errors.

> Parameters

>> • **data/debug** – shared default options for .alm file

>> • **xdata** (*numpy.array or* list[real]) –

>> • **zdata** (*numpy.array or* list[real]) –

>> • **vargs** – Validation data

>> • **kwargs** – Additional options may be specified and will be applied to the .alm

idaes.surrogate.alamopy.doalamo.**cleanFiles**(*data*, *debug*, *pywrite=False*, ***kwargs*)
  Removes intermediate files

> Parameters

>> • **data/debug** – shared default options for .alm file

>> • **vargs** – Validation data

idaes.surrogate.alamopy.doalamo.**constructXBounds**(*xdata*, *zdata*, *data*, *debug*)
  Construct xmin,xmax and zmin, zmax for alamo if none are given

> Parameters

>> • **xdata** (*numpy.array or* list[real]) –

>> • **zdata** (*numpy.array or* list[real]) –

>> • **data** – shared alamo data options

>> • **debug** – Additional options may be specified and will be applied to the .alm

idaes.surrogate.alamopy.doalamo.**doalamo**(*xdata*, *zdata*, ***kwargs*)
  Warning: doalamo is deprecated. please use alamopy.doalamo.alamo(xdata, zdata, **kwargs)

idaes.surrogate.alamopy.doalamo.**expandOutput**(*xdata*, *zdata*, *vargs*, *data*, *debug*)
  Expand output to validation metrics and labels

> Parameters

>> • **data/debug** – shared default options for .alm file

>> • **xdata** (*numpy.array or* list[real]) –

>> • **zdata** (*numpy.array or* list[real]) –

>> • **vargs** – Validation data

idaes.surrogate.alamopy.doalamo.**getTrainingData**(*xdata*, *zdata*, *data*, *debug*)
:   Structure data for training the model. Modifies data['opts']

    Args: xdata (numpy.array or list[real]) zdata (numpy.array or list[real]) data: shared alamo data options debug: Additional options may be specified and will be applied

    > to the .alm

idaes.surrogate.alamopy.doalamo.**getValidationData**(*vargs*, *data*, *debug*)
:   Structure data for validating the model. Modifies data['opts']

    Args: vargs: validation data valxdata, valzdata data: shared alamo data options debug: Additional options may be specified and will be applied

    > to the .alm

idaes.surrogate.alamopy.doalamo.**getlabels**(*data*, *debug*, *kwargs*)
:   Creates labels for data and output. Modifies data['labs']. Makes labels if no labels are given.

    Parameters

    - **data** – shared alamo data options

    - **debug** – Additional options may be specified and will be applied to the .alm

    - **vargs** – validation data valxdata, valzdata

idaes.surrogate.alamopy.doalamo.**makelabs**(*data*, *debug*, *param*)
:   Constructs labels for alamo

    Args: data: shared alamo data options debug: Additional options may be specified and will be applied

    > to the .alm

    param = 'ninputs' or 'noutputs

idaes.surrogate.alamopy.doalamo.**manageArguments**(*xdata*, *zdata*, *data*, *debug*, *kwargs*)
:   Parse additional input options The 'pargs' library is used to keep track of options a user has availible descriptions of the dictionaries data, and debug are given in shared.py Multiple keys used to make writing the .alm file easier

    Parameters

    - **xdata** (*numpy.array or list[real]*) –

    - **zdata** (*numpy.array or list[real]*) –

    - **data** – shared alamo data options

    - **debug** – Additional options may be specified and will be applied to the .alm

idaes.surrogate.alamopy.doalamo.**parseKwargs**(*data*, *debug*, *kwargs*)
:   Parse keyword arguments

    Parameters

    - **data** – shared alamo data options

    - **debug** – Additional options may be specified and will be applied to the .alm

    - **kwargs** – keyword arguments

idaes.surrogate.alamopy.doalamo.**readTraceFile**(*vargs*, *data*, *debug*)
:   Read the alamo trace file to read in the model and metrics

    Parameters

    - **data/debug** – shared default options for .alm file

- **vargs** – Validation data

idaes.surrogate.alamopy.doalamo.**setupData**(*data*, *debug*, *xdata*, *zdata*, *vargs*, *kwargs*)

> **[xdata, zdata, xvaldata, zvaldata] =** setupData(data, debug, xdata,zdata, vargs, kwargs)

> Checks inputted data and resturctures the data for the .alm file

> > Parameters

> > > - **data/debug** – shared default options for .alm file
> > > - **xdata**(*numpy.array or* *list[real]*) –
> > > - **zdata**(*numpy.array or* *list[real]*) –
> > > - **vargs** – Validation data
> > > - **kwargs** – Additional options may be specified and will be applied to the .alm

idaes.surrogate.alamopy.examples module

idaes.surrogate.alamopy.mapminmax module

idaes.surrogate.alamopy.multos module

idaes.surrogate.alamopy.multos.**catfile**(*outf*, **fname*)
> Concatenates files

idaes.surrogate.alamopy.multos.**copyfile**(*outf*, *inf*)
> Copies files

idaes.surrogate.alamopy.multos.**deletefile**(**fname*)
> Deletes files

idaes.surrogate.alamopy.multos.**has_alamo**()
> Checks for ALAMO

idaes.surrogate.alamopy.multos.**movefile**(**fname*)
> Moves files

idaes.surrogate.alamopy.remapminmax module

idaes.surrogate.alamopy.shared module

Set the alamo and gams paths here.

idaes.surrogate.alamopy.simwrapper module

idaes.surrogate.alamopy.writethis module

idaes.surrogate.helmet package

Submodules

idaes.surrogate.helmet.AncillaryEquations module

Modeling for saturated densities and vapor pressure

`idaes.surrogate.helmet.AncillaryEquations.`**`DL`**`()`
    ALAMO regression of Saturated Liquid Density

`idaes.surrogate.helmet.AncillaryEquations.`**`DV`**`()`
    ALAMO regression of saturated vapor density

`idaes.surrogate.helmet.AncillaryEquations.`**`PV`**`()`
    ALAMO regression of vapor pressure

`idaes.surrogate.helmet.AncillaryEquations.`**`getDL`**`()`
    Imports the regressed saturated liquid density function

`idaes.surrogate.helmet.AncillaryEquations.`**`getDV`**`()`
    Imports the regressed saturated vapor density function

`idaes.surrogate.helmet.AncillaryEquations.`**`getPV`**`()`
    Imports the regressed vapor pressure function

idaes.surrogate.helmet.BasisFunctions module

Basis functions for generating the multiparameter equation of state

`idaes.surrogate.helmet.BasisFunctions.`**`arBY`**`(`*D*, *T*, *Y*, *Beta*`)`
    Residual Helmholtz contribution

`idaes.surrogate.helmet.BasisFunctions.`**`d2rd`**`(`*D*, *T*`)`
    Partial derivative with respect to density twice

`idaes.surrogate.helmet.BasisFunctions.`**`d2rdRes`**`(`*D*, *T*, *Y*, *Beta*`)`
    Residual helmholtz contribution second partial derivative with respect to density

`idaes.surrogate.helmet.BasisFunctions.`**`d2rdrtRes`**`(`*D*, *T*, *Y*, *Beta*`)`
    Residual helmholtz contribution third partial derivative with respect to density(2) and temperature(1)

`idaes.surrogate.helmet.BasisFunctions.`**`d2rdt`**`(`*D*, *T*`)`
    Partial derivative with respect to density twice and temperature

`idaes.surrogate.helmet.BasisFunctions.`**`d3rd`**`(`*D*, *T*`)`
    Third partial derivative with respect to density

`idaes.surrogate.helmet.BasisFunctions.`**`d3rdRes`**`(`*D*, *T*, *Y*, *Beta*`)`
    Third partial derivative with respect to density

`idaes.surrogate.helmet.BasisFunctions.`**`d4rd`**`(`*D*, *T*`)`
    Fourth partial derivative with respect to density

`idaes.surrogate.helmet.BasisFunctions.`**`d5rd`**`(`*D*, *T*`)`
    Fifth partial derivative with respect to density

`idaes.surrogate.helmet.BasisFunctions.`**`drd`**`(`*D*, *T*`)`
    Partial derivative with respect to density

`idaes.surrogate.helmet.BasisFunctions.`**`drdRes`**`(`*D*, *T*, *Y*, *Beta*`)`
    Calculates the partial derivaties w.r.t. density Inputs:

        D - Delta T - Tau Y - index of basis Function (int or array) Beta - weighting (float or array)

`idaes.surrogate.helmet.BasisFunctions.`**`dtrdt`**(*D*, *T*)
    Second partial derivative with respect to density and temperature

`idaes.surrogate.helmet.BasisFunctions.`**`dtrdtRes`**(*D*, *T*, *Y*, *Beta*)
    Residual helmholtz contribution second partial derivative with respect to density and temperature

`idaes.surrogate.helmet.BasisFunctions.`**`formCustomBasis`**(*LemJac=False*)
    Basis Functions developed a bank of terms based on literature (Lemmon, Span, Wagner)

`idaes.surrogate.helmet.BasisFunctions.`**`getTerm`**(*Y*)
    Prints index and basis function based on Y index

`idaes.surrogate.helmet.BasisFunctions.`**`iTT`**(*D*, *T*)
    Ideal helmholtz contribution second partial derivative with respect to temperature

`idaes.surrogate.helmet.BasisFunctions.`**`idealBY`**(*D*, *T*, *Y*, *Beta*)
    Ideal Helmholtz contribution

`idaes.surrogate.helmet.BasisFunctions.`**`molData`**(*fluidData*, *Dmolecule*, *RVal*)
    Passing of the Data from the main module ::module:: MPEOSDeveloperModule

> **Parameters**
>
> - **fluidData** (`array`) – (critT, critP, critD, M, triple, acc).
> - **Dmolecule** (`str.`) – Name of the molecule of interest.
> - **RVal** (`int.`) – Gas Constant.

`idaes.surrogate.helmet.BasisFunctions.`**`rTT`**(*D*, *T*)
    Second partial derivative with respect to temperature

`idaes.surrogate.helmet.BasisFunctions.`**`rTTRes`**(*D*, *T*, *Y*, *Beta*)
    Residual helmholtz contribution second partial derivative with respect to temperature

idaes.surrogate.helmet.DataImport module

Importing thermodynamic data, specific structures for text files

`idaes.surrogate.helmet.DataImport.`**`CP`**(*molecule*, *sample=False*, *ratio=5*)
    Import isobaric heat capacity data

`idaes.surrogate.helmet.DataImport.`**`CP0`**(*molecule*)
    Import ideal isobaric heat capacity

`idaes.surrogate.helmet.DataImport.`**`CV`**(*molecule*, *sample=False*, *ratio=5*)
    Import isochoric heat capacity

`idaes.surrogate.helmet.DataImport.`**`DL`**(*molecule*)
    Import saturated liquid density

`idaes.surrogate.helmet.DataImport.`**`DV`**(*molecule*)
    Import of saturated vapor density

`idaes.surrogate.helmet.DataImport.`**`PV`**(*molecule*)
    Import saturated vapor pressure

`idaes.surrogate.helmet.DataImport.`**`PVT`**(*molecule*, *sample=False*, *ratio=5*)
    Import pressure-volume-temperature data

`idaes.surrogate.helmet.DataImport.`**`SND`**(*molecule*, *sample=False*, *ratio=5*)
    Import speed of sound data

`idaes.surrogate.helmet.DataImport.`**`molData`**(*fluidData*, *RVal*)
    Molecular data passed to the module

`idaes.surrogate.helmet.DataImport.`**`regionsOfData`**(*molecule*, *DataValues*, *PVT=False*, *CV=False*)
    Organization of data into regions

`idaes.surrogate.helmet.DataImport.`**`sampleData`**(*Regions*, *ratio*)
    Sampling of the data regions

idaes.surrogate.helmet.DataManipulation module

Calculates dimensionless data

`idaes.surrogate.helmet.DataManipulation.`**`CP`**(*x*)
    Calculate dimensionless isobaric heat capacity Inputs:

        X = [Density, Temperature, Isobaric Heat Capacity]

    Outputs: X = [Delta, Tau, CP]

`idaes.surrogate.helmet.DataManipulation.`**`CP0`**(*x*)
    Calculate dimensionless ideal isobaric heat capacity

`idaes.surrogate.helmet.DataManipulation.`**`CV`**(*x*)
    Calculate dimensionless isochoric heat capacity Inputs:

        X = [Density, Temperature, Isochoric Heat Capacity]

    Outputs: X = [Delta, Tau, CV]

`idaes.surrogate.helmet.DataManipulation.`**`DL`**(*x*)
    Calculate Theta and Delta for saturated liquid density Inputs:

        X = [Density, Temperature]

`idaes.surrogate.helmet.DataManipulation.`**`DV`**(*x*)
    Calculate Theta and Delta for saturated vapor density Inputs:

        X = [Density, Temperature]

`idaes.surrogate.helmet.DataManipulation.`**`Dsat`**(*x*)
    Calculate dimensionless terms

`idaes.surrogate.helmet.DataManipulation.`**`P`**(*x*)
    Calculate reduced density and inverse reduced temperature Return array of Delta, Tau, Pressure

`idaes.surrogate.helmet.DataManipulation.`**`PV`**(*x*)
    Calculate Theta, Tau, and Psi for saturated liquid density Inputs:

        X = [Pressure, Temperature]

`idaes.surrogate.helmet.DataManipulation.`**`PVT`**(*x*)
    Calculate dimensionless compressibility Inputs:

        X = [Pressure, Density, Temperature]

    OutputS: X = [Delta, Tau, Compressibility]

`idaes.surrogate.helmet.DataManipulation.`**`SND`**(*x*)
    Calculate dimensionless speed of sound Inputs:

        X = [Density, Temperature, Speed of Sound]

Outputs: X = [Delta, Tau, W]

`idaes.surrogate.helmet.DataManipulation.`**`molData`**(*fluidData*, *mol*, *RVal*)
   Sets up important global values

idaes.surrogate.helmet.GAMSDataWrite module

Writer of the data into the GAMS file

`idaes.surrogate.helmet.GAMSDataWrite.`**`CPdt`**(*textFile*, *DataToWrite*, *Combination=False*, *PlotData=False*)
   Imports Isobaric Heat Capacity(CP) data into the GAMS document

   Parameters

   - **`textFile`**(*str.*) – Gams File written to.

   - **`DataToWrite`**(*array.*) – Data prepared for the GAMS file.

   Returns void.

`idaes.surrogate.helmet.GAMSDataWrite.`**`CVdt`**(*textFile*, *DataToWrite*, *Combination=False*, *PlotData=False*)
   Imports Isochoric Heat Capacity (CV) data into the GAMS document

   Parameters

   - **`textFile`**(*str.*) – Gams File written to.

   - **`DataToWrite`**(*array.*) – Data prepared for the GAMS file.

   Returns void.

`idaes.surrogate.helmet.GAMSDataWrite.`**`Crit`**(*textFile*, *DataToWrite*, *Combination=False*)
   Import Critical data points

`idaes.surrogate.helmet.GAMSDataWrite.`**`InSat`**(*textFile*, *DataToWrite*, *Combination=False*)
   Import saturation density values

`idaes.surrogate.helmet.GAMSDataWrite.`**`PVTdt`**(*textFile*, *DataToWrite*, *Combination=False*, *PlotData=False*)
   Imports P-V-T data into the GAMS document, to be written in :func:'GamsWrite'

   Parameters

   - **`textFile`**(*str.*) – Gams File written to.

   - **`DataToWrite`**(*array.*) – Data prepared for the GAMS file.

   Returns void.

`idaes.surrogate.helmet.GAMSDataWrite.`**`SNDdt`**(*textFile*, *DataToWrite*, *Combination=False*, *PlotData=False*)
   Imports Speed of Sound (SND) data into the GAMS document

   Parameters

   - **`textFile`**(*str.*) – Gams File written to.

   - **`DataToWrite`**(*array.*) – Data prepared for the GAMS file.

   Returns void.

`idaes.surrogate.helmet.GAMSDataWrite.`**`writeExp`**(*textFile*, *Combination=False*)
   Writes into the GDX file the basis function parameters

idaes.surrogate.helmet.GAMSWrite module

GAMS writer for the regression

`idaes.surrogate.helmet.GAMSWrite.`**`GenerateGDXGamsFiledtlmv`**`()`
> Generates a gams file that creates a data .gdx file Creates Combination of PVT, CV, CP, and SND GAMS file. (Titles precoded in).

`idaes.surrogate.helmet.GAMSWrite.`**`GenerateGamsShell`**`()`
> Generates the multiparameter equation of state regression through gams. Imports the molecule data.gdx file Creates Combination of PVT, CV, CP, and SND GAMS file. (Title precoded in).

`idaes.surrogate.helmet.GAMSWrite.`**`closeFile`**`()`
> Closes the GAMS file

`idaes.surrogate.helmet.GAMSWrite.`**`getRunFile`**`()`
> Returns the gams file name

`idaes.surrogate.helmet.GAMSWrite.`**`getTextFile`**`()`
> Returns name of the current textFile

`idaes.surrogate.helmet.GAMSWrite.`**`molData`**`(Dfluids, Dmolecule, Ddata_name, Dterms, Dmax_time)`
> Passing of the Data from the main module ::module:: MPEOSDeveloperModule
>
> > **Parameters**
> >
> > - **Dmolecule** (`str`) – Name of molecule of interest
> > - **Ddata_name** (`str.`) – Name of Data files
> > - **Dmax_time** (`int.`) – Running time limit for the GAMS file.

`idaes.surrogate.helmet.GAMSWrite.`**`openFile`**`(data_name, ending='.gms')`
> Opens the GAMS file based on the data set
>
> > **Parameters data_name** (`str`) – name of the molecule

`idaes.surrogate.helmet.GAMSWrite.`**`setCombination`**`(isCombination)`
> Sets the regression as multiple properties

`idaes.surrogate.helmet.GAMSWrite.`**`setNumberTerms`**`(numterms)`
> Set number of basis functions allowed in regression

`idaes.surrogate.helmet.GAMSWrite.`**`writeBasisFunctions`**`()`
> Write the values of the basis function terms

`idaes.surrogate.helmet.GAMSWrite.`**`writeBoundsB`**`(props)`
> Writes bounds on the variables

`idaes.surrogate.helmet.GAMSWrite.`**`writeCalculateIntermediates`**`(props)`
> Calculate intermediate values of properties not in the regression

`idaes.surrogate.helmet.GAMSWrite.`**`writeConstants`**`(terms)`
> Writes down the ranges of the fitting Beta value and sets regular to one. param terms: number of basis functions param terms: int

`idaes.surrogate.helmet.GAMSWrite.`**`writeDerivatives`**`(props)`
> Calculate and write the important derivatives of the basis functions

`idaes.surrogate.helmet.GAMSWrite.`**`writeEquationsAndVariablesB`**`(props)`
> Writes multiple thermodynamic parameter equations and constants. :param props: Array containing the available properties. :type props: array

---

`idaes.surrogate.helmet.GAMSWrite.`**`writeGamsHeaderdtl`**(*num_points*, *terms*, *kset*, *pset*, *re-*
gions=None)

> Writes the GAMS file Header including the number of terms and data points as well as different thermodynamic properties.

> > **Parameters**

> > > • **num_points** (*int*) – number of data points

> > > • **terms** (*int*) – number of basis functions

> > > • **kset** – what is this

> > > • **pset** – what is this

`idaes.surrogate.helmet.GAMSWrite.`**`writeGamsShellFooterB`**(*data_name*, *load_in=False*)

> Writes the Gam Footer options, model, and display.

`idaes.surrogate.helmet.GAMSWrite.`**`writeGamsShellHeaderB`**(*pset*, *regions=None*)

> Writes the GAMS file Header including the number of terms and data points as well as different thermodynamic properties.

> > **Parameters**

> > > • **pset** – list of thermodynamic properties

> > > • **regions** – regions of thermodynamic property data

`idaes.surrogate.helmet.GAMSWrite.`**`writeModelB`**(*reslim*, *props*)

> Writes the GAMS model definition

`idaes.surrogate.helmet.GAMSWrite.`**`writeModelPostEvaluations`**(*props*)

> Calculates thermodynamic values after the regression

`idaes.surrogate.helmet.GAMSWrite.`**`writeObjectivesB`**(*props*)

> Writes the objective equations for the different combination of properites and constraints

idaes.surrogate.helmet.Helmet module

HELMholtz Energy Thermodynamics (HELMET)

Main capabilities of HELMET default HELMET use

`idaes.surrogate.helmet.Helmet.`**`deletefile`**(**fname*)

> Deletes files

`idaes.surrogate.helmet.Helmet.`**`getFlag`**()

> Returns flag, marks a change in the construction of the model

`idaes.surrogate.helmet.Helmet.`**`initialize`**(***kwargs*)

> filename - location of data gamsname - name of the gams file made molecule - name of the molecule/compound data_name - name of the data fluid data - [critT, critP, critD, M, triple, acentric factor] R - gas constant value

`idaes.surrogate.helmet.Helmet.`**`prepareAncillaryEquations`**(*plot=False*, *keep-*
Files=False)

> **Develops ancillary equations of state using ALAMOPY** DL - saturated liquid density DV - saturated vapor density PV - vapor pressure

> Dependent on ALAMO

`idaes.surrogate.helmet.Helmet.`**`runRegression`**(*gams=False*, *pyomo=False*)

> Runs the gdx and main regression gams file

`idaes.surrogate.helmet.Helmet.`**`setupRegression`**(*numTerms=14*, *gams=False*, *py-omo=False*)
setup gams regression

`idaes.surrogate.helmet.Helmet.`**`updateModelSettings`**()
Settings of the model based on the chemical passed to the different python methods

`idaes.surrogate.helmet.Helmet.`**`viewMultResults`**(*lstFile*, *numTerms=0*)
View mutliple results from a lst file

`idaes.surrogate.helmet.Helmet.`**`viewPropertyData`**()
Plot imported data

`idaes.surrogate.helmet.Helmet.`**`viewResults`**(*lstFile=None*, *plot=False*, *report=False*, *surface=<matplotlib.colors.LinearSegmentedColormap object>*)

Plot results from gams or pyomo lstFile - gams listing file surface - colormapping color eg. cm.coolwarm

idaes.surrogate.helmet.Plotting module

HELMET Plotting capabilities

`idaes.surrogate.helmet.Plotting.`**`HelmetSurface`**(*Y=[]*, *Beta=[]*, *show=True*, *surface=<matplotlib.colors.LinearSegmentedColormap object>*)

Plots Helmholtz Surface

`idaes.surrogate.helmet.Plotting.`**`molData`**(*fluidData*, *Dmolecule*, *RVal*)
Shared data about the molecule and ideas gas constant R

`idaes.surrogate.helmet.Plotting.`**`plotCP`**()
Plot isobaric heat capacity

`idaes.surrogate.helmet.Plotting.`**`plotCV`**()
Plot isochoric heat capacity

`idaes.surrogate.helmet.Plotting.`**`plotDL`**()
Plot saturate liquid density

`idaes.surrogate.helmet.Plotting.`**`plotDV`**()
Plot saturated vapor density

`idaes.surrogate.helmet.Plotting.`**`plotPV`**()
Plot vapor pressure

`idaes.surrogate.helmet.Plotting.`**`plotPVT`**()
Plot Pressure-Volume-Temperature data

`idaes.surrogate.helmet.Plotting.`**`plotSND`**()
Plot speed of sound data

`idaes.surrogate.helmet.Plotting.`**`sseCP`**(*CP1=[]*, *CP1Vals=[]*, *saveFig=False*, *show=True*, *report=False*)
Plots and calculates metrics for isobaric heat capacity

`idaes.surrogate.helmet.Plotting.`**`sseCV`**(*Y=[]*, *Beta=[]*, *saveFig=False*, *show=True*, *report=False*)
Plots and metrics for isochoric heat capacity

`idaes.surrogate.helmet.Plotting.`**`sseCombo`**(*lstFile=None*, *plot=False*, *report=False*, *sur-
face=<matplotlib.colors.LinearSegmentedColormap
object>*)

> Plot regressed equation and data. Calculates statistical anlaysis metrics

`idaes.surrogate.helmet.Plotting.`**`ssePVT`**(*PVT1=[]*, *PVT1Vals=[]*, *saveFig=False*,
show=True, *report=False*)

> Plots and metrics for Pressure-volume-temperature data

`idaes.surrogate.helmet.Plotting.`**`sseSND`**(*SND1=[]*, *SND1Vals=[]*, *saveFig=False*,
show=True, *report=False*)

> Plots and calculates metrics for speed of sound

`idaes.surrogate.helmet.Plotting.`**`viewAnc`**()

> Plots all the ancillary equations for saturated density and vapor pressure

`idaes.surrogate.helmet.Plotting.`**`viewData`**()

> View imported data

idaes.surrogate.helmet.parseGAMS module

Parses and prints the solutions of the multiparameter equation of state solution

`idaes.surrogate.helmet.parseGAMS.`**`getBetas`**()

> Returns the weights of the basis functions

`idaes.surrogate.helmet.parseGAMS.`**`getIndexes`**()

> Returns indexes of the basis function terms

`idaes.surrogate.helmet.parseGAMS.`**`parser`**(*filename*, *num=2*)

> Parse solution files for the muliparameter equation of state

`idaes.surrogate.helmet.parseGAMS.`**`writeEquation`**(*Y*, *Beta=None*)

> Write full multiparameter equation

`idaes.surrogate.helmet.parseGAMS.`**`writeTerm`**(*index*)

> Writes the basis function term with the given index

idaes.surrogate.pysmo package

idaes.surrogate.ripe package

Institute for the Design of Advanced Energy Systems Process Systems # Engineering Framework (IDAES PSE Framework) Copyright (c) 2018, 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. # # Please see the files COPYRIGHT.txt and LICENSE.txt for full copyright and # license information, respectively. Both files are also available online # at the URL "https://github.com/IDAES/idaes".

Submodules

idaes.surrogate.ripe.atermconstruct module

idaes.surrogate.ripe.bounds module

idaes.surrogate.ripe.checkoptions module

idaes.surrogate.ripe.confinv module

idaes.surrogate.ripe.emsampling module

idaes.surrogate.ripe.genpyomo module

idaes.surrogate.ripe.kinforms module

idaes.surrogate.ripe.main module

idaes.surrogate.ripe.mechs module

idaes.surrogate.ripe.read module

idaes.surrogate.ripe.shared module

Set the alamo and gams paths here.

idaes.surrogate.ripe.targets module

idaes.surrogate.ripe.write module

idaes.unit_models package

Subpackages

idaes.unit_models.convergence package

Subpackages

idaes.unit_models.convergence.pressure_changer package

Submodules

idaes.unit_models.convergence.pressure_changer.pressure_changer_conv_eval module

idaes.unit_models.icons package

idaes.unit_models.power_generation package

Submodules

idaes.unit_models.power_generation.IdealProp_FlueGas module

Basic property package for flue gas main assumptions:

- ideal gas

- components in flue gas: O2, N2, NO, CO2, H2O, SO2

class idaes.unit_models.power_generation.IdealProp_FlueGas.**FlueGasParameterBlock**(**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 ProcessBlockData config
>>
>>> **Keys**
>>>
>>>> **default_arguments** Default arguments to use with Property Package
>>
>> - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (FlueGasParameterBlock) New instance

class idaes.unit_models.power_generation.IdealProp_FlueGas.**FlueGasStateBlock**(**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 ProcessBlockData 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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (FlueGasStateBlock) New instance

class idaes.unit_models.power_generation.IdealProp_FlueGas.**FlueGasStateBlockData**(*component*)
This is an example of a property package for calculating the thermophysical properties of flue gas using the ideal gas assumption.

build()
Callable method for Block construction

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_enthalpy_flow_terms(*p*)
Method which returns a valid expression for enthalpy flow to use in the energy balances.

get_material_density_terms(*p, j*)
Method which returns a valid expression for material density to use in the material balances .

get_material_flow_basis()
Method which returns an Enum indicating the basis of the material flow term.

get_material_flow_terms(*p, j*)
Method which returns a valid expression for material flow to use in the material balances.

model_check()
Model checks for property block

class idaes.unit_models.power_generation.IdealProp_FlueGas.**PhysicalParameterData**(*component*)
Property Parameter Block Class

Contains parameters and indexing sets associated with properties for flue gas. The ideal gas assumption is applied.

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

idaes.unit_models.power_generation.boiler_heat_exchanger module

Power Plant IDAES heat exchanger model. The boiler heat exchanger model consist of a cross flow shell and tube hx that can be used for any of the boiler components, such as economizer, reheater or superheaters (primary, secondary, etc). The model includes shell and tube rigorous heat transfer calculations and pressure drop calculations for shell side. Note that this model assumes no phase transitions (if user requires phase transitions, they need a general model)

the main config arguments:

- delta T method: counter-current or co-current

- tube_arrangement: in-line or staggered

- **has radiation:**

    - True if model is used as a reheater or superheater unit

    - Gas emissivity calculated (Gas temperature above 700 K)

- SI units (consistent with prop pack)

- heat transfer calc U = f(Nu, Re, Pr)

- Pressure drop tube and shell side (fricction factor calc.)

class idaes.unit_models.power_generation.boiler_heat_exchanger.**BoilerHeatExchanger**(**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 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:** { **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 ethalpy balance for material, **EnergyBalanceType.enthalpyPhase** - ethalpy 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 T calculations (default = 'counter-current')

– 'counter-current' : counter-current flow arrangement

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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (BoilerHeatExchanger) New instance

class idaes.unit_models.power_generation.boiler_heat_exchanger.**BoilerHeatExchangerData**(*comp*
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={'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 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 infomation (tee=True)

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating whcih 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

class idaes.unit_models.power_generation.boiler_heat_exchanger.**DeltaTMethod**
> An enumeration.

class idaes.unit_models.power_generation.boiler_heat_exchanger.**TubeArrangement**
> An enumeration.

idaes.unit_models.power_generation.feedwater_heater_0D module

This file contains 0D feedwater heater models. These models are suitable for steady state calculations. For dynamic modeling 1D models are required. There are two models included here.

1) FWHCondensing0D: this is a regular 0D heat exchanger model with a constraint added to ensure all the steam fed to the feedwater heater is condensed at the outlet. At the shell outlet the molar enthalpy is equal to the the staurated liquid molar enthalpy.

2) FWH0D is a feedwater heater model with three sections and a mixer for combining another feedwater heater's drain outlet with steam extracted from the turbine. The drain mixer, desuperheat, and drain cooling sections are optional. Only the condensing section is required.

class idaes.unit_models.power_generation.feedwater_heater_0D.**FWH0D**(*args,
> ***kwargs*)
> Feedwater Heater Model This is a 0D feedwater heater model. The model may contain three 0D heat exchanger models representing the desuperheat, condensing and drain cooling sections of the feedwater heater. Only the condensing section must be included. A drain mixer can also be optionally included, which mixes the drain outlet of another feedwater heater with the steam fed into the condensing section.

> > **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:** { **True** - construct holdup terms, **False** - do not construct holdup terms}

> > > > **has_drain_mixer** Add a mixer to the inlet of the condensing section to add water from the drain of another feedwaterheater to the steam, if True

has_desuperheat Add a mixer desuperheat section to the heat exchanger

has_drain_cooling Add a section after condensing section to cool condensate.

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.}

condense ProcessBlockData

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}

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 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.}

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 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.}

delta_temperature_callback Callback for for temperature difference calculations

flow_pattern Heat exchanger flow pattern, **default** - HeatExchanger-FlowPattern.countercurrent. **Valid values:** { **HeatExchangerFlowPattern.countercurrent** - countercurrent flow, **HeatExchangerFlowPattern.cocurrent** - cocurrent flow, **HeatExchangerFlowPattern.crossflow** - cross flow, factor times countercurrent temperature difference.}

desuperheat ProcessBlockData

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}

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 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.}

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 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.}

delta_temperature_callback Callback for for temperature difference calculations

flow_pattern Heat exchanger flow pattern, **default** - HeatExchanger-FlowPattern.countercurrent. **Valid values:** { **HeatExchangerFlowPattern.countercurrent** - countercurrent flow, **HeatExchangerFlowPattern.cocurrent** - cocurrent flow, **HeatExchangerFlowPattern.crossflow** - cross flow, factor times countercurrent temperature difference.}

cooling ProcessBlockData

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}

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 flowsheet, **PropertyParameterObject** - a PropertyPa-
rameterBlock 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 con-
structed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **Mate-
rialBalanceType.useDefault - refer to property package for default bal-
ance type **MaterialBalanceType.none** - exclude material balances, **Ma-
terialBalanceType.componentPhase** - use phase component balances, **Ma-
terialBalanceType.componentTotal** - use total component balances, **Mate-
rialBalanceType.elementTotal** - use total element balances, **MaterialBal-
anceType.total** - use total material balance.}

energy_balance_type Indicates what type of energy balance should be con-
structed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **Energy-
BalanceType.useDefault - refer to property package for default balance
type **EnergyBalanceType.none** - exclude energy balances, **EnergyBal-
anceType.enthalpyTotal** - single enthalpy balance for material, **EnergyBal-
anceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBal-
anceType.energyTotal** - single energy balance for material, **EnergyBalance-
Type.energyPhase** - energy balances for each phase.}

momentum_balance_type Indicates what type of momentum balance should
be constructed, **default** - MomentumBalanceType.pressureTotal. **Valid val-
ues:** { **MomentumBalanceType.none** - exclude momentum balances, **Mo-
mentumBalanceType.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 equi-
librium 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 calcula-
tions, **default** - useDefault. **Valid values:** { **useDefault** - use default package
from parent model or flowsheet, **PropertyParameterObject** - a PropertyPa-
rameterBlock 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 calcula-
tions

flow_pattern Heat exchanger flow pattern, **default** - HeatExchanger-
FlowPattern.countercurrent. **Valid values:** { **HeatExchangerFlowPat-**

tern.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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns: (FWH0D) New instance

class idaes.unit_models.power_generation.feedwater_heater_0D.**FWH0DData**(*component*)

build()

General build method for UnitModelBlockData. 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(**args*, ***kwargs*)

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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

class idaes.unit_models.power_generation.feedwater_heater_0D.**FWHCondensing0D**(**args*, ***kwargs*)

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:** { **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 con-
> structed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **Materi-
> alBalanceType.useDefault - refer to property package for default balance
> type **MaterialBalanceType.none** - exclude material balances, **MaterialBal-
> anceType.componentPhase** - use phase component balances, **MaterialBal-
> anceType.componentTotal** - use total component balances, **MaterialBalance-
> Type.elementTotal** - use total element balances, **MaterialBalanceType.total** -
> use total material balance.}

> **energy_balance_type** Indicates what type of energy balance should be con-
> structed, **default** - EnergyBalanceType.useDefault. **Valid values:** { **Ener-
> gyBalanceType.useDefault - refer to property package for default bal-
> ance type **EnergyBalanceType.none** - exclude energy balances, **Energy-
> BalanceType.enthalpyTotal** - single enthalpy balance for material, **Energy-
> BalanceType.enthalpyPhase** - enthalpy balances for each phase, **EnergyBal-
> anceType.energyTotal** - single energy balance for material, **EnergyBalance-
> Type.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, **Momentum-
> BalanceType.pressureTotal** - single pressure balance for material, **Momen-
> tumBalanceType.pressurePhase** - pressure balances for each phase, **Momen-
> tumBalanceType.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 calcula-
> tions, **default** - useDefault. **Valid values:** { **useDefault** - use default package
> from parent model or flowsheet, **PropertyParameterObject** - a PropertyParam-
> eterBlock object.}

> **property_package_args** A ConfigBlock with arguments to be passed to a prop-
> erty 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 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.}

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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns:** (FWHCondensing0D) New instance

class idaes.unit_models.power_generation.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.

idaes.unit_models.power_generation.turbine_inlet module

Steam turbine inlet stage model. This 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.

class idaes.unit_models.power_generation.turbine_inlet.**TurbineInletStage**(**args*, ***kwargs*)

> 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 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:** { **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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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 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.}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (TurbineInletStage) New instance

class idaes.unit_models.power_generation.turbine_inlet.**TurbineInletStageData**(*component*)

build()

Parameters **None** –

Returns None

initialize(*state_args={}*, *outlvl=6*, *solver='ipopt'*, *optarg={'max_iter': 30, 'tol': 1e-06}*)
Initialize the inlet 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** (*int*) – Amount of output (0 to 3) 0 is lowest
- **solver** (*str*) – Solver to use for initialization
- **optarg** (*dict*) – Solver arguments dictionary

idaes.unit_models.power_generation.turbine_multistage module

Multistage steam turbine for power generation.

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, November

class idaes.unit_models.power_generation.turbine_multistage.**TurbineMultistage**(**args*,
**kwargs)

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 ProcessBlockData config

Keys

dynamic Indicates whether the model is dynamic.

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}

has_phase_equilibrium Argument indicating whether phase equilibrium should be cal-
culated for the resulting mixed stream, **default** - False. **Valid values:** { **True** - calcu-
late phase equilibrium in mixed stream, **False** - do not calculate equilibrium in mixed
stream.}

material_balance_type Indicates what type of mass balance should be constructed, **de-
fault** - MaterialBalanceType.componentTotal'. **Valid values:** { **MaterialBalance-
Type.none** - exclude material balances, **MaterialBalanceType.componentPhase** - use
phase component balances, **MaterialBalanceType.componentTotal** - use total com-
ponent balances, **MaterialBalanceType.elementTotal** - use total element balances,
MaterialBalanceType.total - use total material balance.}

property_package Property parameter object used to define property calculations, **de-
fault** - 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 prop-
erty package for documentation.}

num_parallel_inlet_stages Number of parallel inlet stages to simulate partial arc admis-
sion. Default=4

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 addtional 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 addtional 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 addtional 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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (TurbineMultistage) New instance

class idaes.unit_models.power_generation.turbine_multistage.**TurbineMultistageData**(*component*)

build()
> General build method for UnitModelBlockData. 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=6*, *solver='ipopt'*, *optarg={'max_iter': 35, 'tol': 1e-06}*)
> Initialize

throttle_cv_fix(*value*)
> Fix the thottle 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 coefficents at

idaes.unit_models.power_generation.turbine_outlet module

Steam turbine outlet stage model. This 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.

class idaes.unit_models.power_generation.turbine_outlet.**TurbineOutletStage**(*args, **kwargs*)

> 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 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:** { **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 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.}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (TurbineOutletStage) New instance

class idaes.unit_models.power_generation.turbine_outlet.**TurbineOutletStageData**(*component*)

build()

> **Parameters None** –
>
> **Returns** None

initialize(*state_args={}*, *outlvl=6*, *solver='ipopt'*, *optarg={'max_iter': 30, 'tol': 1e-06}*)
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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>
> - **solver** (*str*) – Solver to use for initialization
>
> - **optarg** (*dict*) – Solver arguments dictionary

idaes.unit_models.power_generation.turbine_stage module

Steam turbine stage model. This is a standard isentropic turbine. Under off-design conditions the base efficiency and pressure ratio do not change much for the stages between the inlet and outlet. This 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.

`class` idaes.unit_models.power_generation.turbine_stage.**TurbineStage**(*args*,
**kwargs)

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 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:** { **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 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.}

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (TurbineStage) New instance

class idaes.unit_models.power_generation.turbine_stage.**TurbineStageData**(*component*)

> **build**()
>
> > **Parameters None** –
> >
> > **Returns** None
>
> **initialize**(*state_args={}, outlvl=6, 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**
> >
> > - **state_args** (*dict*) – Initial state for property initialization
> >
> > - **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
> >
> > - **solver** (*str*) – Solver to use for initialization
> >
> > - **optarg** (*dict*) – Solver arguments dictionary

idaes.unit_models.power_generation.valve_steam module

This provides valve models for steam and liquid water. These are for steam cycle control valves and the turbine throttle valves.

class idaes.unit_models.power_generation.valve_steam.**SteamValve**(**args, **kwargs*)

> Basic steam valve models
>
> > **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 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}

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 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.}

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_rule** This is a rule that returns a time indexed valve function expression. This is required only if valve_function==ValveFunctionType.custom

> > **phase** Expected phase of fluid in valve in {"Liq", "Vap"}

- **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (SteamValve) New instance

class idaes.unit_models.power_generation.valve_steam.**SteamValveData**(*component*)

> **build**()

>> **Parameters None** –

>> **Returns** None

> **initialize**(*state_args={}*, *outlvl=6*, *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**

- **state_args** (*dict*) – Initial state for property initialization

- **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **solver** (*str*) – Solver to use for initialization

- **optarg** (*dict*) – Solver arguments dictionary

class idaes.unit_models.power_generation.valve_steam.**ValveFunctionType**
> An enumeration.

Submodules

idaes.unit_models.cstr module

Standard IDAES CSTR model.

class idaes.unit_models.cstr.**CSTR**(**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 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:** { **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 flowsheet, **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`) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (CSTR) New instance

class idaes.unit_models.cstr.**CSTRData**(*component*)

> Standard CSTR Unit Model Class

> **build**()

> > Begin building model (pre-DAE transformation). :param None:

> > > **Returns** None

idaes.unit_models.equilibrium_reactor module

Standard IDAES Equilibrium Reactor model.

class idaes.unit_models.equilibrium_reactor.**EquilibriumReactor**(**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 ProcessBlockData 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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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, **MomentumBalance-Type.pressureTotal** - single pressure balance for material, **MomentumBalance-Type.pressurePhase** - pressure balances for each phase, **MomentumBalance-Type.momentumTotal** - single momentum balance for material, **MomentumBalance-Type.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 equlibirum 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 flowsheet, **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*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (EquilibriumReactor) New instance

class idaes.unit_models.equilibrium_reactor.**EquilibriumReactorData**(*component*)
> Standard Equilibrium Reactor Unit Model Class

> **build**()
> > Begin building model.

> > > **Parameters None** –

> > > **Returns** None

idaes.unit_models.feed module

Standard IDAES Feed block.

class idaes.unit_models.feed.**Feed**(**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 ProcessBlockData 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 flowsheet, **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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (Feed) New instance

class idaes.unit_models.feed.**FeedData**(*component*)
> Standard Feed Block Class

> **build**()
> > Begin building model.

> > > **Parameters None** –

> **Returns** None

initialize(*state_args={}, outlvl=6, 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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>
> - **optarg** – solver options dictionary object (default={'tol': 1e-6})
>
> - **solver** – str indicating which solver to use during initialization (default = 'ipopt')
>
> **Returns** None

idaes.unit_models.feed_flash module

Standard IDAES Feed block with phase equilibrium.

class idaes.unit_models.feed_flash.**FeedFlash**(**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 ProcessBlockData 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 flowsheet, **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (FeedFlash) New instance

class idaes.unit_models.feed_flash.**FeedFlashData**(*component*)
> Standard Feed block with phase equilibrium

build()
> Begin building model.

> > **Parameters None** –

> > **Returns** None

class idaes.unit_models.feed_flash.**FlashType**
> An enumeration.

idaes.unit_models.flash module

Standard IDAES flash model.

class idaes.unit_models.flash.**Flash**(**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 ProcessBlockData 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 **Energy-**

> > > BalanceType.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** - oulet 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 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.}

> > - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> > **Returns** (Flash) New instance

class idaes.unit_models.flash.**FlashData**(*component*)

> Standard Flash Unit Model Class

> **build**()

> > Begin building model (pre-DAE transformation).

> > > **Parameters** **None** –

> > > **Returns** None

idaes.unit_models.gibbs_reactor module

Standard IDAES Gibbs reactor model.

class idaes.unit_models.gibbs_reactor.**GibbsReactor**(**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 ProcessBlockData 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 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.}
>
> - **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (GibbsReactor) New instance

class idaes.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 repect to product species mole flow rates and the multiples are specified as constraints.

build()

Begin building model (pre-DAE transformation).

> **Parameters None** –
>
> **Returns** None

idaes.unit_models.heat_exchanger module

Heat Exchanger Models.

class idaes.unit_models.heat_exchanger.**HeatExchanger**(**args*, ***kwargs*)

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 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:** { **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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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 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.}

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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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, **MomentumBalance-**

> > **Type.pressureTotal** - single pressure balance for material, **MomentumBalance-Type.pressurePhase** - pressure balances for each phase, **MomentumBalance-Type.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 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.}
> >
> > **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (HeatExchanger) New instance

class idaes.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=1000*)
    Heat exchanger initialization method.

    Parameters

    - **state_args_1** – a dict of arguments to be passed to the property initialization for side_1 (see documentation of the specific property package) (default = {}).

    - **state_args_2** – a dict of arguments to be passed to the property initialization for side_2 (see documentation of the specific property package) (default = {}).

    - **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for

each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **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 transfered (default = 10000)

> **Returns** None

set_scaling_factor_energy(*f*)
> This function sets scaling_factor_energy for both side_1 and side_2. This factor multiplies the energy balance and heat transfer equations in the heat exchnager. The value of this factor should be about 1/(expected heat duty).
>
> **Parameters** **f** – Energy balance scaling factor

class idaes.unit_models.heat_exchanger.**HeatExchangerFlowPattern**
> An enumeration.

idaes.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.unit_models.heat_exchanger.**delta_temperature_lmtd_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.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.

idaes.unit_models.heat_exchanger_1D module

Basic IDAES 1D Heat Exchanger Model.

1D Single pass shell and tube HX model with 0D wall conduction model

class idaes.unit_models.heat_exchanger_1D.**HeatExchanger1D**(**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 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:** { **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:** { **True** - construct holdup
terms, **False** - do not construct holdup terms}

material_balance_type Indicates what type of mass balance should be con-
structed, **default** - MaterialBalanceType.useDefault. **Valid values:** { **Ma-
terialBalanceType.useDefault - refer to property package for default bal-
ance type **MaterialBalanceType.none** - exclude material balances, **Material-
BalanceType.componentPhase** - use phase component balances, **MaterialBal-
anceType.componentTotal** - use total component balances, **MaterialBalance-
Type.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:** { **EnergyBalance-
Type.useDefault - refer to property package for default balance type **Energy-
BalanceType.none** - exclude energy balances, **EnergyBalanceType.enthalpyTotal**
- single enthalpy balance for material, **EnergyBalanceType.enthalpyPhase** - en-
thalpy 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 con-
structed, **default** - MomentumBalanceType.pressureTotal. **Valid values:** { **Mo-
mentumBalanceType.none** - exclude momentum balances, **MomentumBalance-
Type.pressureTotal** - single pressure balance for material, **MomentumBalance-
Type.pressurePhase** - pressure balances for each phase, **MomentumBalance-
Type.momentumTotal** - single momentum balance for material, **MomentumBal-
anceType.momentumPhase** - momentum balances for each phase.}

has_pressure_change Indicates whether terms for pressure change should be con-
structed, **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 (de-
fault = '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 Property-
BlockData and used when constructing these (default = 'use_parent_value') -
'use_parent_value' - get package from parent (default = None) - a dict (see prop-
erty 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 transformating 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:** { **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 transformating 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 lenghth 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (HeatExchanger1D) New instance

class idaes.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=5*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>
> - **optarg** – solver options dictionary object (default={'tol': 1e-6})
>
> - **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')
>
> **Returns** None

class idaes.unit_models.heat_exchanger_1D.**WallConductionType**
An enumeration.

idaes.unit_models.heater module

Basic heater/cooler models

class idaes.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 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:** { **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 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.}

> > > - **initialize** (*dict*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> > **Returns** (Heater) New instance

class idaes.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

idaes.unit_models.mixer module

General purpose mixer block for IDAES models

class idaes.unit_models.mixer.**Mixer**(**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 ProcessBlockData 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 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.}
> > > >
> > > > **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 minimimum 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 StateBock 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (Mixer) New instance

class idaes.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 specified 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 mutiple 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 fuction.

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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={})

- **solver** – str indicating whcih 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=6*)

    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 = mimimum 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 = mimimum 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.

class idaes.unit_models.mixer.**MixingType**

    An enumeration.

class idaes.unit_models.mixer.**MomentumMixingType**

    An enumeration.

idaes.unit_models.plug_flow_reactor module

Standard IDAES PFR model.

class idaes.unit_models.plug_flow_reactor.**PFR**(**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 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:** { **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 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.}

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 transformating 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 to use when transforming length domain, **default** - 3.

- **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

>   **Returns** (PFR) New instance

class idaes.unit_models.plug_flow_reactor.**PFRData**(*component*)

> Standard Plug Flow Reactor Unit Model Class

> **build**()
>
> > Begin building model (pre-DAE transformation).
> >
> > > **Parameters None** –
> > >
> > > **Returns** None

idaes.unit_models.pressure_changer module

Standard IDAES pressure changer model.

class idaes.unit_models.pressure_changer.**PressureChanger**(**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 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:** { **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 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.}

- **initialize** (*[dict](#)*) – ProcessBlockData config for individual elements. Keys are Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (PressureChanger) New instance

class idaes.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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>>>>
>>>> - **optarg** – solver options dictionary object (default={'tol': 1e-6})
>>>>
>>>> - **solver** – str indicating whcih solver to use during initialization (default = 'ipopt')
>>>
>>> **Returns** None

initialize(*state_args=None*, *routine=None*, *outlvl=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>
> - **optarg** – solver options dictionary object (default={'tol': 1e-6})
>
> - **solver** – str indicating whcih 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

class idaes.unit_models.pressure_changer.**ThermodynamicAssumption**
    An enumeration.

idaes.unit_models.product module

Standard IDAES Product block.

class idaes.unit_models.product.**Product**(**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 ProcessBlockData 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 flowsheet, **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (Product) New instance

class idaes.unit_models.product.**ProductData**(*component*)

Standard Product Block Class

build()

Begin building model.

> **Parameters None** –

> **Returns** None

initialize(*state_args={}, outlvl=6, 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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

> **Returns** None

idaes.unit_models.separator module

General purpose separator block for IDAES models

class idaes.unit_models.separator.**EnergySplittingType**

An enumeration.

class idaes.unit_models.separator.**Separator**(**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 ProcessBlockData 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 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.}

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 **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** - oulet 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 spearation 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 StateBock 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*) – 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

> **Returns** (Separator) New instance

class idaes.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 mutiple 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*)
Creates outlet Port objects and tries to partiton 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 SeparatorData. 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=6*, *optarg={}*, *solver='ipopt'*, *hold_state=False*)
Initialization routine for separator (default solver ipopt)

Keyword Arguments

- **outlvl** – sets output level of initialization routine * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default=None)

- **solver** – str indicating whcih 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

partition_outlet_flows(*mb*, *outlet_list*)

Creates outlet Port objects and tries to partiton 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=6*)

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

class idaes.unit_models.separator.**SplittingType**

An enumeration.

idaes.unit_models.statejunction module

Standard IDAES StateJunction model.

class idaes.unit_models.statejunction.**StateJunction**(**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 ProcessBlockData 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 flowsheet, **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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

Returns (StateJunction) New instance

class idaes.unit_models.statejunction.**StateJunctionData**(*component*)

> Standard StateJunction Unit Model Class

build()

> Begin building model. :param None:
>
> > **Returns** None

initialize(*state_args={}*, *outlvl=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
> >
> > - **optarg** – solver options dictionary object (default={'tol': 1e-6})
> >
> > - **solver** – str indicating which solver to use during initialization (default = 'ipopt')
> >
> > **Returns** None

idaes.unit_models.stoichiometric_reactor module

Standard IDAES STOICHIOMETRIC reactor model

class idaes.unit_models.stoichiometric_reactor.**StoichiometricReactor**(**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 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:** { **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:** { **EnergyBalance-Type.useDefault - refer to property package for default balance type **Energy-BalanceType.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 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 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.}

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 Block-Data 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.

    Returns (StoichiometricReactor) New instance

class idaes.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

---

idaes.unit_models.translator module

Generic template for a translator block.

class idaes.unit_models.translator.**Translator**(**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 ProcessBlockData 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 sumation 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 notcalculate 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 overide the default behavior of matching the BlockData index exactly to the index in initialize.
>
> **Returns** (Translator) New instance

class idaes.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=6*, *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 * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output

- **optarg** – solver options dictionary object (default={'tol': 1e-6})

- **solver** – str indicating which solver to use during initialization (default = 'ipopt')

Returns None

idaes.util package

Submodules

idaes.util.sphinxdoctest_plugin module

This module implements pytest plugin for Sphinx doc tests.

In a nutshell, it uses the pytest *pytest_collect_file()* plugin hook to recognize the Sphinx Makefile. Then it does a quick and dirty parse of that Makefile to extract the command Sphinx is using to run the doctests, which it recognizes by being the first command in the Makefile target named by *SPHINX_DOCTEST_TARGET*. The parser is able to handle simple Makefile variable expansion, though not currently nested variables so don't do that.

The mechanics of the pytest plugin mechanism are such that the Makefile is wrapped with a subclass of `pytest.File`, SphinxMakefile, which implements the *collect* method to yield a subclass of `pytest.Item` called SphinxItem, that in turn implements a few methods to run the test and report the result. The bulk of the code in running the test is parsing the output to look for errors, and thus decide whether all the doctests passed, or not.

The drawback of this whole setup is of course some extra complexity. The advantage is that (a) whatever the Makefile does is what this plugin should do, for running the command, as long as the command is the first (and only significant) thing that occurs in the target, and (b) if there ends up being more than one Makefile, it should all continue to work.

exception `idaes.util.sphinxdoctest_plugin.`**SphinxCommandFailed**

class `idaes.util.sphinxdoctest_plugin.`**SphinxDoctestFailure** (*name*, *parent*, *details*)

class `idaes.util.sphinxdoctest_plugin.`**SphinxDoctestItem** (*name*, *parent*, *wd*, *cmd*)

repr_failure (*excinfo*)
This is called when self.runtest() raises an exception.

runtest()
>   Run the Sphinx doctest.

class `idaes.util.sphinxdoctest_plugin.`**SphinxDoctestSuccess**(*name*, *parent=None*, *config=None*, *session=None*, *nodeid=None*)

exception `idaes.util.sphinxdoctest_plugin.`**SphinxHadErrors**

class `idaes.util.sphinxdoctest_plugin.`**SphinxMakefile**(*fspath: py._path.local.LocalPath*, *parent=None*, *config=None*, *session=None*, *nodeid=None*)

>   **collect**()
>   >   returns a list of children (items and collectors) for this collection node.

>   **warnings_file**
>   >   Get warnings and errors output file, if any, from the Sphinx Makefile.

class `idaes.util.sphinxdoctest_plugin.`**SphinxWarnings**(*fspath: py._path.local.LocalPath*, *parent=None*, *config=None*, *session=None*, *nodeid=None*)

>   **collect**()
>   >   returns a list of children (items and collectors) for this collection node.

class `idaes.util.sphinxdoctest_plugin.`**SphinxWarningsItem**(*name*, *parent*, *path: pathlib.Path*)

>   **repr_failure**(*excinfo*)
>   >   This is called when self.runtest() raises an exception.

idaes.util.testutil module

Utility code for testing IDAES code.

`idaes.util.testutil.`**run_notebook**(*path: str*, *name: str*)
>   Run a specific jupyter notebook 'name' located at *path*.

idaes.util.update_workshop_materials module

This script downloads a python file from pyomo.org that will allow us to update the workshop material easily during a workshop.

The file install_idaes_workshop_materials.py is downloaded from pyomo.org, imported, and the method execute() is then called to do whatever actions are necessary.

Note: - update_workshop_materials.py is a module in the idaes/util folder that does the work - update_workshop_materials.ipynb is a Jupyter notebook in the examples/workshops folder

>   that calls this module

- install_idaes_workshop_materials.py is posted on a site (for now pyomo.org, but later could be a repository on github).

1) JupyterHub: User executes update_workshop_materials.ipynb which calls to update_workshop_materials.py in idaes/util

2) update_workshop_materials.py downloads another python file (install_idaes_workshop_materials.py) from pyomo.org and calls "execute()" from that module.

3) install_idaes_workshop_materials.py does whatever is necessary to get the workshop materials into the user folder (and perform any updates necessary).

TODO: This should probably be changed to get a zip file from an IDAES repository rather than install_idaes_workshop_materials.py from pyomo.org

idaes.util.update_workshop_materials.**download_install_module**()
> Downloads install_idaes_workshop_materials.py from pyomo.org

idaes.util.update_workshop_materials.**import_install_module**(*download_dest*)
> Imports the path in download_dest (install_idaes_workshop_materials.py module)

idaes.vis package

The *idaes.vis* subpackage contains the framework and implementation of plots that are expected to be of general utility within the IDAES framework.

For users, an entry point is provided for IDAES classes to produce plots with the *idaes.vis.plotbase. PlotRegistry* singleton.

Plots will inherit from the interface in *idaes.vis.plotbase.PlotBase*, which provides some basic methods.

The current implementations all use the Python "bokeh" package, and can be found in *idaes.vis.bokeh_plots*.

For more details, please refer to the visualization section of the main IDAES documentation.

Submodules

idaes.vis.bokeh_plots module

Bokeh plots.

class idaes.vis.bokeh_plots.**BokehPlot**(*current_plot=None*)

> **annotate**(*x*, *y*, *label*)
> > Annotate a plot with a given point and a label.
> >
> > **Parameters**
> > - **x** – Value of independent variable.
> > - **y** – Value of dependent variable.
> > - **label** – Text label.
> >
> > **Returns** None
> >
> > **Raises** None
>
> **resize**(*height=-1*, *width=-1*)
> > Resize a plot's height and width.
> >
> > **Parameters**
> > - **height** – Height in screen units.

> - **width** – Width in screen units.

> > **Returns** None

> > **Raises** None

> **save**(*destination*)
> > Save the current plot object to HTML in filepath provided by destination.

> > **Parameters destination** – Valid file path to save HTML to.

> > **Returns** filename where HTML is saved.

> > **Raises** None

> **show**(*in_notebook=True*)
> > Display plot in a Jupyter notebook.

> > **Parameters**

> > - **self** – Plot object.

> > - **in_notebook** – Display in Jupyter notebook or generate HTML file.

> > **Returns** None

> > **Raises** None

class idaes.vis.bokeh_plots.**HeatExchangerNetwork**(*exchangers*, *stream_list*, *mark_temperatures_with_tooltips=False*, *mark_modules_with_tooltips=False*, *stage_width=2*, *y_stream_step=1*)

class idaes.vis.bokeh_plots.**ProfilePlot**(*data_frame*, *x=''*, *y=None*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=None*)

idaes.vis.plotbase module

Base classes for visualization and plotting in IDAES.

Create new plots by inheriting from *PlotBase*. See the *idaes.vis.bokeh_plots* module for examples.

class idaes.vis.plotbase.**PlotBase**(*current_plot*)
> Abstract base class for a plot.

> **annotate**(*x*, *y*, *label: str*)
> > Annotate a plot with a given point and a label.

> > **Parameters**

> > - **x** – Value of independent variable.

> > - **y** – Value of dependent variable.

> > - **label** – Text label.

> **resize**(*height: int = -1*, *width: int = -1*)
> > Resize a plot's height and width.

> > **Parameters**

> > - **height** – Height in screen units.

> > - **width** – Width in screen units.

> > **Returns** None

> **Raises** `None`

save (*destination: str*)

> Save the current plot object to HTML in filepath provided by destination.
>
> > **Parameters** `destination` – Valid file path to save HTML to.
> >
> > **Returns** filename where HTML is saved.
> >
> > **Raises** `None`

show (*in_notebook=True*)

> Display plot in a Jupyter notebook.
>
> > **Parameters** `in_notebook` – Display in Jupyter notebook or generate HTML file.
> >
> > **Returns** None
> >
> > **Raises** `None`

classmethod validate (*data_frame: pandas.core.frame.DataFrame, x: str, y: List[T], legend=None*)

> Validate that the plot parameters are valid.
>
> > **Parameters**
> >
> > - **data_frame** – a pandas data frame of any type.
> > - **x** – Key in data-frame to use as x-axis.
> > - **y** – Keys in data-frame to use as y-axis.
> > - **legend** – List of labels to use as legend for a plot.
> >
> > **Returns** True, '' on valid data frames (if x and y are in the data frame keys) False, "message" on invalid data

class idaes.vis.plotbase.**PlotRegistry**

> Set of associations between objects/classes + a plot name, and the parameters and values needed to perform the plot.
>
> The basic idea is to create a set of named plots associated with a given IDAES class, and then allow the user or other APIs to invoke that plot once the data is populated in an instance of the class. This keeps the details of how to create plots of a given type in the classes that will create them.
>
> For example:

```python
class MyIdaesClass(ProcessBase):
  # .. code for the class
  def plot_setup(self, plot_class):
      # .. details of creating plot_instance from object contents ..
      return plot_instance
PlotRegistry().register(MyIdaesClass, 'basic', MyIdaesClass.plot_setup)

# .. and, later ..
obj = MyIdaesClass(...)
# .. do things that fill "obj" with data ..
# now create the plot
plot = PlotRegistry().get(obj, 'basic')
plot.show()
```

> XXX: This class is not actually used (yet) by any of the IDAES models.

get (*obj, name: str*)

> Get a plot object for the given object + name.

Parameters

- **obj** – Object for which to get the plot

- **name** – Registered name of plot to get

Returns Return value of setup function given to `register()`, or, if that is empty, the retrieved plot object.

register(*obj, name: str, plot: Type[CT_co], setup_fn=None, overwrite: bool = False*)
Register an object/plot combination.

Parameters

- **obj** – Class or instance

- **name** – Name for this association

- **plot** – Plot class

- **setup_fn** – Optional setup function to call. Function should take two arguments: *plot* class instance, *obj* assoc. with plot.

- **overwrite** – If true, allow overwrite of existing entry in the registry

remove_all()
Remove all entries from the registry.

Since the registry is a singleton, this removes all entries from ALL instances. Use with care.

idaes.vis.plotutils module

class idaes.vis.plotutils.**HENStreamType**
Enum type defining hot and cold streams

idaes.vis.plotutils.**add_exchanger_labels**(*plot, x, y_start, y_end, label_font_size, exchanger, module_marker_line_color, module_marker_fill_color, mark_modules_with_tooltips*)
Plot exchanger labels for an exchanger (for Q and A) on a heat exchanger network diagram and add module markers (if needed).

Parameters

- **plot** – bokeh.plotting.plotting.figure instance.

- **label_font_size** – font-size for labels.

- **x** – x-axis coordinate of exchanger (exchangers are vertical lines so we just need 1 x-value)

- **y_start** – y-axis coordinate of exchanger start.

- **y_end** – y-axis coordinate of exchanger end.

- **exchanger** – exchanger dictionary of the form:

```
{'hot': 'H2', 'cold': 'C1', 'Q': 1400, 'A': 159, 'annual_cost':␣
→28358,
 'stg': 2}
```

- **module_marker_line_color** – color of border of the module marker.

- **module_marker_fill_color** – color inside the module marker.

- **`mark_modules_with_tooltips`** – whether to add tooltips to plot or not (currently not utilized).

> **Returns** modified bokeh.plotting.plotting.figure instance with labels added.

> **Raises** `None`

`idaes.vis.plotutils.`**`add_module_markers_to_heat_exchanger_plot`**(*plot*, *x*, *y*, *modules*, *line_color*, *fill_color*, *mark_modules_with_tooltips*)

> Plot module markers as tooltips to a heat exchanger network diagram.

> **Parameters**

- **`plot`** – bokeh.plotting.plotting.figure instance.

- **`x`** – x-axis coordinate of module marker tooltip.

- **`y`** – y-axis coordinate of module marker tooltip.

- **`modules`** – dict containing modules.

- **`line_color`** – color of border of the module marker.

- **`fill_color`** – color inside the module marker.

- **`mark_modules_with_tooltips`** – whether to add tooltips to plot or not (currently not utilized).

> **Returns** bokeh.plotting.plotting.figure instance with module markers added.

> **Raises** `None`

`idaes.vis.plotutils.`**`get_color_dictionary`**(*set_to_color*)

> Given a set, return a dictionary of the form:

```
{'set_member': valid_bokeh_color}
```

> **Args:** set_to_color: set of unique elements, e.g: [1,2,3] or ["1", "2", "3"]

> **Returns:** Dictionary of the form:

```
{'set_member': valid_bokeh_color}
```

> **Raises:** None

`idaes.vis.plotutils.`**`get_stream_y_values`**(*exchangers*, *hot_streams*, *cold_streams*, *y_stream_step=1*)

> Return a dict containing the layout of the heat exchanger diagram including any stage splits.

> **Parameters**

- **`exchangers`** – List of exchangers where each exchanger is a dict of the form:

```
{'hot': 'H2', 'cold': 'C1', 'Q': 1400, 'A': 159, 'annual_cost':␣
↪28358,
'stg': 2}
```

where hot is the hot stream name, cold is the cold stream name, A is the area (in m^2), annual_cost is the annual cost in $, Q is the amount of heat transferred from one stream

to another in a given exchanger and stg is the stage the exchanger belongs to. Additionally a 'utility_type' can specify if we draw the cold stream as water (`idaes.vis.plot_utils.HENStreamType.cold_utility`) or the hot stream as steam (`idaes.vis.plot_utils.HENStreamType.hot_utility`).

Additionally, the exchanger could have the key 'modules', like this:

```
{'hot': 'H1', 'cold': 'C1', 'Q': 667, 'A': 50, 'annual_cost':
↪10979, 'stg': 3,
'modules': {10: 1, 20: 2}}
```

- **hot_streams** – List of dicts representing hot streams where each item is a dict of the form:

```
{'name':'H1', 'temps': [443, 435, 355, 333], 'type': HENStreamType.
↪hot}
```

- **cold_streams** – List of dicts representing cold streams where each item is a dict of the form:

```
{'name':'H1', 'temps': [443, 435, 355, 333], 'type': HENStreamType.
↪hot}
```

- **y_stream_step** – how many units on the HEN diagram to leave between each stream (or sub-stream) and the one above it. Defaults to 1.

Returns

* stream_y_values_dict : a dict of each stream name as key and value being a dict of the form

```
{'default_y_value': 2, 'split_y_values': [1,3]}.
```

This indicates what the default y value of this stream will be on the diagram and what values we'll use when it splits.

* hot_split_streams : list of tuples of the form (a,b) where a is a hot stream name and b is the max. times it will split over all the stages.

* cold_split_streams : list of tuples of the form (a,b) where a is a cold stream name and b is the max. times it will split over all the stages.

Return type Tuple containing 3 dictionaries to be used when plotting the HEN

Raises None

idaes.vis.plotutils.**is_hot_or_cold_utility**(*exchanger*)

Return if an exchanger is a hot or a cold utility by checking if it has the key *utility_type*.

Parameters **exchanger** – dict representing the exchanger.

Returns True if *utility_type* in the *exchanger* dict passed.

Raises None

idaes.vis.plotutils.**plot_line_segment**(*plot*, *x_start*, *x_end*, *y_start*, *y_end*, *color='white'*, *legend=None*)

Plot a line segment on a bokeh figure.

Parameters

- **plot** – bokeh.plotting.plotting.figure instance.

- **x_start** – x-axis coordinate of 1st point in line.

- **x_end** – x-axis coordinate of 2nd point in line.

- **y_start** – y-axis coordinate of 1st point in line.

- **y_end** – y-axis coordinate of 2nd point in line.

- **color** – color of line (defaults to white).

- **legend** – what legend to associate with (defaults to None).

Returns modified bokeh.plotting.plotting.figure instance with line added.

Raises None

idaes.vis.plotutils.**plot_stream_arrow**(*plot*, *line_color*, *stream_arrow_temp*, *temp_label_font_size*, *x_start*, *x_end*, *y_start*, *y_end*, *stream_name=None*)
Plot a stream arrow for the heat exchanger network diagram.

Parameters

- **plot** – bokeh.plotting.plotting.figure instance.

- **line_color** – color of arrow (defaults to white).

- **stream_arrow_temp** – Tempreature of the stream to be plotted.

- **temp_label_font_size** – font-size of the temperature label to be added.

- **x_start** – x-axis coordinate of arrow base.

- **x_end** – x-axis coordinate of arrow head.

- **y_start** – y-axis coordinate of arrow base.

- **y_end** – y-axis coordinate of arrow head.

- **stream_name** – Name of the stream to add as a label to arrow (defaults to None).

Returns modified bokeh.plotting.plotting.figure instance with stream arrow added.

Raises None

idaes.vis.plotutils.**turn_off_grid_and_axes_ticks**(*plot*)
Turn off axis ticks and grid lines on a bokeh figure object.

Parameters **plot** – bokeh.plotting.plotting.figure instance.

Returns modified bokeh.plotting.plotting.figure instance.

Raises None

Submodules

idaes.config module

idaes.config.**create_dir**(*d*)
Create a directory if it doesn't exist.

Parameters **d** (*str*) – directory path to create

Retruns: None

`idaes.config.`**`import_packages`**(*packages*, *optional=True*)

Import plugin package, condensed from pyomo.environ.__init__.py :param packages: list of packages in which to look for plugins :param optional: true, log ImportError but continue; false, raise if ImportError

> **Returns** None

`idaes.config.`**`read_config`**(*read_config*, *write_config*)

Read either a TOML formatted config file or a configuration dictionary. :param config: A config file path or dict

> **Returns** None

idaes.logger module

`idaes.logger.`**`condition`**(*res*)

Get the solver termination condition. Since it seems to be common to have an if block to check for None if the solver call raised a handled exception

`idaes.logger.`**`getIdaesLogger`**(*name*, *level=None*)

Return an idaes logger.

> **Parameters**
>
> - **name** – usually __name__
> - **level** – standard IDAES logging level (default use IDAES config)
>
> **Returns** logger

`idaes.logger.`**`getInitLogger`**(*name*, *level=None*)

Get a model initialization logger

> **Parameters**
>
> - **name** – Object name (usually Pyomo Component name)
> - **level** – Logging detail level (for initialization routines 1 to 6) * 0 = Use default idaes.init logger setting * 1 = Maximum output * 2 = Include solver output * 3 = Return solver state for each step in subroutines * 4 = Return solver state for each step in routine * 5 = Final initialization status and exceptions * 6 = No output
>
> **Returns** logger

`idaes.logger.`**`getModelLogger`**(*name*, *level=None*)

Get a logger for an IDAES model. This function helps users keep their loggers in a standard location and using 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)
>
> **Returns** logger

`idaes.logger.`**`init_tee`**(*logger*, *tee_level=2*)

Function to use in initialization to determine at a given output level whether to use the sovler tee option to print solver output. This function just helps standardize the level for solver output to appear and make the initialization routine code a bit cleaner.

> **Parameters**
>
> - **logger** – logger to get output level from

> • **tee_level** – Level at which to show solver output, usually use default

>> **Returns** (bool)

idaes.logger.**solver_tee**(*logger*, *tee_level=10*)
> Function to produce solver output based on the logging level of a specific logger. This function just helps standardize the level for solver output to appear and make code a bit cleaner.

>> **Parameters**

>>> • **logger** – logger to get output level from

>>> • **tee_level** – Level at which to show solver output, usually use default

>> **Returns** (bool)

idaes.solvers module

idaes.solvers.**download_binaries**(*url=None*, *verbose=False*)
> Download IDAES solvers and libraries and put them in the right location. Need to supply either local or url argument.

>> **Parameters** **url** (*str*) – a url to download binary files to install files

>> **Returns** None

idaes.ver module

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')
```

(continues on next page)

```
...
>>> obj = MyClass()
>>> print(obj.version)
1.2.3.a
```

class idaes.ver.**HasVersion**(**args*)

> Interface for a versioned class.

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.

idaes.ver.**git_hash**()

> Get current git hash, with no dependencies on external packages.

idaes.ver.**package_version = <idaes.ver.Version object>**

> Package's version as an object

4.15 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.

4.16 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.

4.17 Copyright

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

W