IDAES Documentation

Release 0.5.0

IDAES team

Sep 06, 2018

CONTENTS

1 Project Goals 1

2 Collaborating institutions 3

3 Contents 5

4 Indices and tables 265

Python Module Index 267

Index 269

CHAPTER

ONE

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

TWO

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)

CHAPTER

THREE

CONTENTS

3.1 Installation Instructions

Contents

- *Installation Instructions*
  - *Dependencies*
    - *CPLEX*
    - *Gurobi*
    - *IPOPT*
  - *Installation on Linux/Unix*
    - *Install Pyomo*
    - *Install IDAES*
    - *IDAES Developer installation*
  - *Installation on Windows*
    - *Python Distribution*
    - *Pyomo*
    - *IDAES*
      - *Option 1: Download zip file*
      - *Option 2: Using Git*

The IDAES toolkit is written in Python. It should run under versions of Python 2.7 and 3.6, and above. The toolkit uses [Pyomo](https://www.pyomo.org), a Python-based optimization language. See the Pyomo website for details.

Note: Although Python can run on most operating systems, *we are currently only supporting installation of the IDAES PSE framework on Linux*. This is due largely to complications of installing third-party solvers, not inherent properties of the PSE framework itself, and we plan to support Windows and Mac OSX installation in the not-too-distant future.

3.1.1 Dependencies

Some of the model code depends on external solvers.

CPLEX

- Getting CPLEX
- Setting up CPLEX Python

Gurobi

- Gurobi license
- Gurobi solver
- Gurobi Python setup

IPOPT

- Installing IPOPT

3.1.2 Installation on Linux/Unix

Install Pyomo

- Install the master branch of PyUtilib from GitHub using pip:

  pip install git+https://github.com/PyUtilib/pyutilib

- Install the IDAES branch of Pyomo from GitHub using pip:

  pip install git+https://github.com/Pyomo/pyomo@IDAES

Install IDAES

- The installation is performed by a script at the top level called *install.sh*. This script will work on UNIX and MacOS systems. There is no Windows script at this time. See below for installation instructions on Windows.
- This script uses an advanced, but common, Python packaging system called [Conda](https://conda.io/docs/). Please first consult the [Conda documentation](https://conda.io/docs/user-guide/) to install this on your system. You can use either Anaconda or Miniconda.
- Conda allows you to to create separate environments containing files, packages and their dependencies that will not interact with other environments. The install script will automatically create a conda environment, but you need to pick a name for it. Pick a name using only letters, numbers, dashes, or underscores, such as "idaes-python3". Run the script with this name as the first argument:

  ```
  ` ./install.sh MY_NAME `
  ```

- **The script will run commands to:**

  1. Create the conda environment
  2. Build and install the IDAES code into the environment
  3. Build the HTML documentation

- **If the install *succeeds*, the console will display:** *** SUCCESS ***
- **If the install *fails*, the console will display:** *!! FAILURE !!*

IDAES Developer installation

The following instructions are for developers and advanced users.

- Install the master branch of IDAES from GitHub:

  git clone https://github.com/IDAES/idaes.git

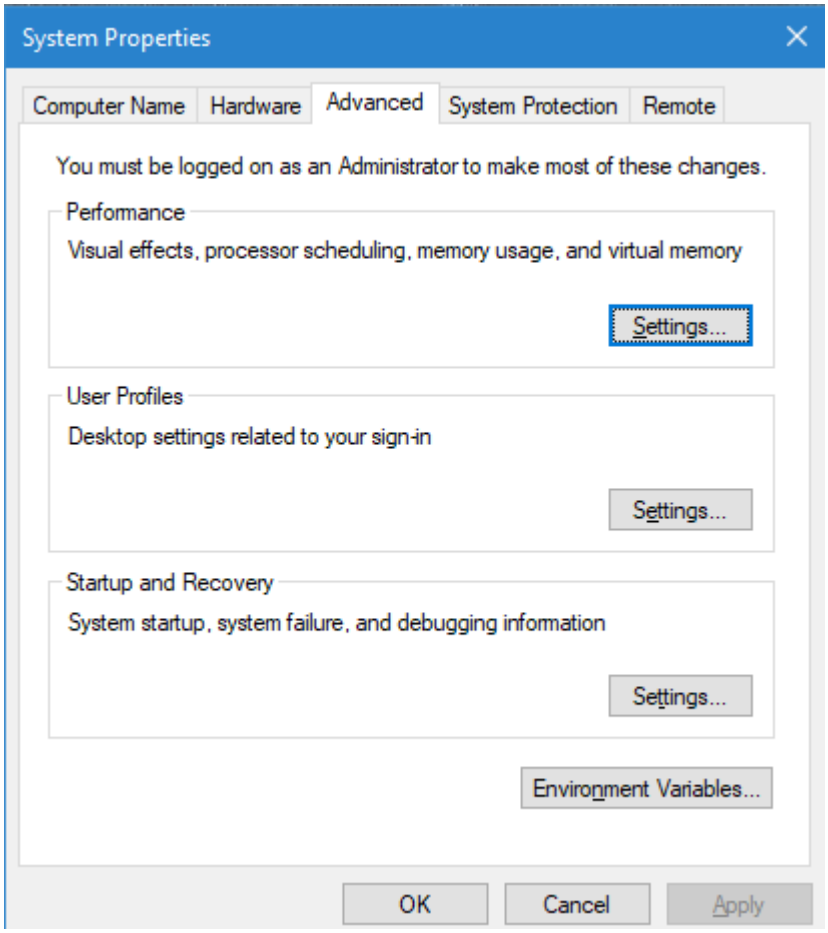
- Create/switch to your preferred Python environment
- Install the requirements with *pip install -r requirement.txt*
- Run *python setup.py develop*.

3.1.3 Installation on Windows

Note: We are NOT supporting Windows at this time. Some developers on the team have had success with the following instructions, but we do not promise that they will work for all users, nor will we prioritize helping debug problems.

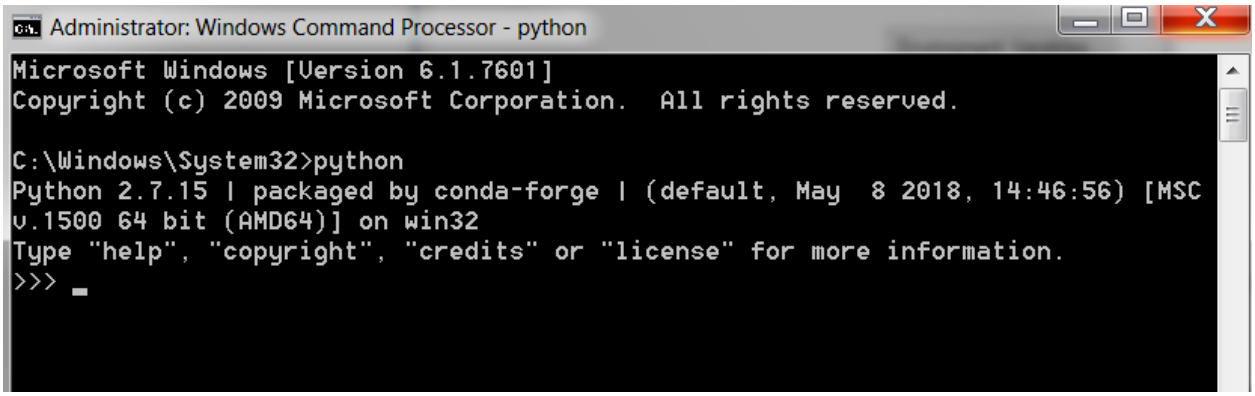
Python Distribution

- Install Anaconda for Windows
- Add Anaconda and Anaconda scripts to the path "c:users<user>Anaconda2" and "c:users<user>Anaconda2Scripts". To do this, search for "Edit system variables" in Windows search. Click on "Edit system environment variables". Click on "Environment Variables". Under "System Variables", search for the variable "Path" and click "Edit"



1. For Windows 10:
   (a) In the new dialog box, click on "New" and add the path where you find the python.exe file. If you installed Anaconda2, this should be in "c:users<user>Anaconda2". Copy the address and paste it here.
   (b) Repeat for "c:users<user>Anaconda2Scripts".
2. For earlier versions:
   (a) Add path to the existing list, use semicolon as separator
   (b) Type "c:users<user>Anaconda2;c:users<user>Anaconda2Scripts"

- Restart the command prompt and type *python*. If the path variable was added correctly, then you should be able to see the python interpreter as shown below.



Pyomo

- See instructions for pyomo installation. As mentioned, you can either use the pip or the conda install methods which come included with the Anaconda distribution but conda may be preferable if you installed Anaconda.
- To install pyomo using python's **pip** package, follow these steps:
  1. Launch the "Anaconda prompt". You can find this in the start menu under Anaconda.
  2. Navigate to the "Scripts" folder in Anaconda. Or simply type, *where pip* in the prompt. This should return 1 paths and this should be in the scripts folder.
  3. Pip install pyomo from trunk (we recommend installing the IDAES branch of pyomo)
     (a) Install the master branch of PyUtilib from GitHub using pip:

     pip.exe install git+https://github.com/PyUtilib/pyutilib

     (b) Install the master branch of Pyomo from GitHub using pip:

     pip.exe install git+https://github.com/Pyomo/pyomo@IDAES
- To install using python's **conda** package, follow the following steps:
  1. Launch the "Anaconda prompt". You can find this in the start menu under Anaconda.
  2. Navigate to the "Scripts" folder in Anaconda. Or simply type, *where conda* in the prompt. This should return 2 paths and one of these should be in the scripts folder.
  3. In the scripts folder run the following commands:

     conda.exe install -c conda-forge pyomo

     conda.exe install -c conda-forge pyomo.extras
- If the installation was successful, you should see the pyomo executable listed in the Scripts folder. You can check this using the *where pyomo* command.

IDAES

Option 1: Download zip file

- From the IDAES repository on GitHub, click on "Clone or download" on the right in green. Click on "Download zip".
- Extract the contents in the desired directory you want IDAES in.

- Open command prompt and navigate to the folder where you extracted the contents of the IDAES repository (*cd <user>/. . . /<desired directory>/IDAES/*).
    1. Run: *python setup.py develop*

Option 2: Using Git

- Install git for Windows.
- If cloning the repository from the command line, move to a directory where you want to install the IDAES repository. Then run the following command:
    1. *git clone https://github.com/IDAES/idaes.git*
- Enter your github user id and password. The git installation in 1 should have added the git executable to your system path and you should be able to execute git commands from the command line.
- Open command prompt and navigate to the folder where you extracted the contents of the IDAES repository (*cd <user>/. . . /<desired directory>/IDAES/*).
    1. Run: *python setup.py develop*

3.2 Core Library

Contents:

3.2.1 Introduction to IDAES Models

The IDAES Toolkit, based on Python and Pyomo, aims to provide multi-scale, simulation-based, open source computational tools and *Models* to support the design, analysis, optimization, scale-up, operation and troubleshooting of innovative, advanced fossil energy systems.

See the main IDAES web site for more information about the project.

3.2.2 IDAES Tutorials

The following tutorials have been prepared to guide new users through learning the IDAES modeling framework and how to use the IDAES model library. The tutorials are written for users with little to no existing knowledge of Python and Pyomo, however it is likely to be helpful to be familiar with these and the concepts of object-oriented programing in general. The IDAES documentation contains some recommended resources for an introduction to Pyomo, Python and object-oriented programming which the user is encouraged to read.

These tutorials all require Python, Pyomo and IDAES to be installed, as well as a non-linear program solver to be installed for use in solving the flowsheets being generated. The tutorial uses IPOPT as the default solver, however users may make use of any NLP solver they have available. If no solver is available, users should read the IDAES documentation for guidance on finding and installing an NLP solver.

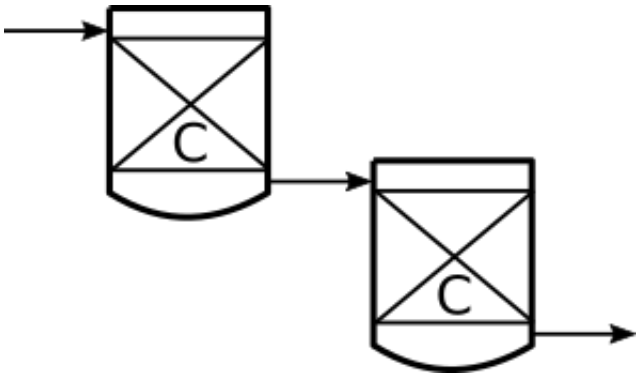
The tutorials are designed to be followed in order, and will gradually introduce users the different parts of the IDAES toolset and how to use them. Completed examples of each tutorial and supporting models are available in the *idaes/examples/core/tutorials* folder.

Contents

Tutorial 1 - Basic Flowsheets

Introduction

For this tutorial on how to construct simple flowsheets within the IDAES framework, we will use a simple steady-state process of two well-mixed reactors (CSTRs) in series, with the following system reactions occurring in each.



$$a + 2b \rightleftharpoons c + d$$

$$a + 2c \rightleftharpoons 2e$$

$$a + b \rightleftharpoons f$$

In this tutorial, you will learn how to:

- import the necessary libraries from Pyomo and IDAES
- create a flowsheet object
- add a property package to a flowsheet
- add unit models to a flowsheet
- connect units together
- set inlet conditions and design variables
- initialize a model
- solve the model and print some results

First Steps

The first thing we need to do is import some components from Pyomo which will be used in our model:

- *ConcreteModel* will be used to form the basis of our model, and
- *SolverFactory* will be used to solve our flowsheet.

All of these can be imported from *pyomo.environ* using the following format:

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

Next, we need to import *FlowsheetBlock* and *Stream* from the *ideas.core*. FlowsheetBlock used as the basis for constructing all flowsheets in IDAES, and provides the necessary infrastructure for constructing a flowsheet, whilst Stream is used to connect different Unit Models together.

```
from idaes.core import FlowsheetBlock, Stream
```

Finally, we need to import models for the reactors and the properties calculations we wish to use. For this tutorial we will be using the CSTR model available in the IDAES model library, and a set of property calculations that has been prepared and is available at *idaes/examples/core/tutorials/tutorial_1_properties.py*. We will also import and use the IDAES Feed and Product Blocks to use in our flowsheet.

These can be imported with the following code:

```
import tutorial_1_properties as properties_rxn
from idaes.models import Feed, CSTR, Product
```

Setting up a model and flowsheet

Now that we have imported the necessary libraries, we can begin constructing the model of our process. The first step in constructing a model within the IDAES framework is to create a *ConcreteModel* with which to contain the flowsheet. This is the same as creating a ConcreteModel within Pyomo.

```
m = ConcreteModel()
```

Next, we add a *FlowsheetBlock* object to our model. *FlowsheetBlock* is an IDAES modelling object which contains all the attributes required of a flowsheet within the IDAES framework (such as the time domain for dynamic models). *FlowsheetBlock* supports a number of different options, which can be set by providing arguments when the block is instantiated. For the current example we want a steady-state flowsheet, which is done by setting the argument *dynamic=False*.

To add a flowsheet named *fs* to our model, we do the following:

```
m.fs = FlowsheetBlock(dynamic=False)
```

This creates a flowsheet block within our model which we can now start to populate with property calculations and unit models. As unit models depend on property packages for part of their construction, we need to add the property package to our flowsheet first. This is done by creating an instance of a *PropertyParameterBlock*, which is part of all IDAES property packages. This block contains all the information required to set up a property package for our flowsheet, and can be passed to a unit model in order to set up the property calculations within that model.

To add a property package with the name *properties_rxn* to our Flowsheet, we do the following:

```
m.fs.properties_rxn = properties_rxn.PropertyParameterBlock()
```

The example property package for this tutorial is fairly simple, and does not need any arguments; however more complex packages may have options that can be set during construction. Users should always refer to the documentation for any property package they are using to understand any available options. It is also possible for a flowsheet to contain multiple property packages, which can be used in different parts of the flowsheet.

The next thing we need for our Flowsheet is a source of material entering the process, for which we will use a Feed Block. For now, we will just create the Feed Block, and we will come back later to specify the conditions of the incoming material. When we create our Feed Block, we need to pass the property package we wish to use for the mateiral stream to the unit model as an argument.

```
m.fs.Feed = Feed(property_package=m.fs.properties_rxn)
```

Next, we can add our two CSTRs to the flowsheet, which we will call *Tank1* and *Tank2*. In both cases, we again need to pass the property package to the unit model as an argument. Unit models can also take a number of other arguments, however these will be covered in later tutorials.

To add a unit model to a flow sheet, create an instance of the model and pass any desired arguments to it. For example:

```
m.fs.Tank1 = CSTR(property_package=m.fs.properties_rxn)
m.fs.Tank2 = CSTR(property_package=m.fs.properties_rxn)
```

Finally, let us add a Product Block to serve as a marker for the final state of the material. Just like for Feed and CSTR, we need to provide the property package argument.

```
m.fs.Product = Product(property_package=m.fs.properties_rxn)
```

Feed and Product Blocks are not necessary in flowsheets, and a fucntional flowsheet can be built without them. However, these serve as convenient markers for sources and destinations of material within the process.

The final stage in setting up the components in a flowsheet is to call *post_transform_build*. Due to some current limitations in Pyomo (hopefully to be fixed in Pyomo 5.5), there are some parts of IDAES models that cannot be constructed until after any DAE transformations have been applied. *post_transform_build* is a method attached to *FlowsheetBlock* which automatically goes through all unit models attached to the flowsheet and performs any tasks that must be performed after DAE transformations (more on this in later tutorials). This is most commonly associated with the time domain in dynamic models, and is not required in the current model. Thus, for steady-state models, *post_transform_build* can be called immediately after the unit models have been declared.

```
m.fs.post_transform_build()
```

Connecting Units

Now that we have added our unit models to the flowsheet and called *post_transform_build* on our flowsheet, we can begin connecting units together. Connections need to be made after *post_transform_build* is called, as these are some of the things that cause problems with Pyomo's DAE transformation. Each unit model will contain a number of inlets and outlets Port objects, which can be connected using IDAES Stream objects. In our flowsheet, each unit has a single inlet and a single outlet, named *inlet* and *outlet* respectively. For our flowsheet, we need to connect the following:

- outlet of Feed to the inlet of Tank1 (Stream 1),
- outlet of Tank1 to the inlet of Tank2 (Stream 2),
- outlet of Tank2 to the inlet of Product (Stream 3).

```
m.fs.stream_1 = Stream(source=m.fs.Feed.outlet, destination=m.fs.Tank1.inlet)
m.fs.stream_2 = Stream(source=m.fs.Tank1.outlet, destination=m.fs.Tank2.inlet)
m.fs.stream_3 = Stream(source=m.fs.Tank2.outlet, destination=m.fs.Product.inlet)
```

At this point, we have finished constructing our flowsheet, and can now move onto specifying our operating conditions and solving the model.

Setting Operating Conditions

In setting the operating conditions, the first thing we need to specify are the inlet conditions to process, which can be done through the Feed Block. For our model, we need to specify flow rates of each component (a through f) as well as the pressure and temperature of the inlet stream.

The conditions we need to fix are:

- flow_mol_comp[“a”] = 1.0 [mol/s]
- flow_mol_comp[“b”] = 2.0 [mol/s]
- flow_mol_comp[“c”] = 0.1 [mol/s]
- flow_mol_comp[“d”] = 0.0 [mol/s]
- flow_mol_comp[“e”] = 0.0 [mol/s]
- flow_mol_comp[“f”] = 0.0 [mol/s]
- temperature = 303.15 [K]
- pressure = 101325.0 [Pa]

```
m.fs.Feed.fix('flow_mol_comp', comp='a', value=1.0)
m.fs.Feed.fix('flow_mol_comp', comp='b', value=2.0)
m.fs.Feed.fix('flow_mol_comp', comp='c', value=0.1)
m.fs.Feed.fix('flow_mol_comp', comp='d', value=0.0)
m.fs.Feed.fix('flow_mol_comp', comp='e', value=0.0)
m.fs.Feed.fix('flow_mol_comp', comp='f', value=0.0)
m.fs.Feed.fix('temperature', value=303.15)
m.fs.Feed.fix('pressure', value=101325)
```

Additionally, we need to specify some design conditions for the system – in this case the volume and heat of both tanks. Let us fix the volume of each tank to be 10 m^3 and the heat duty to be 0 J/s. The variable names are “volume” and “heat”.

```
m.fs.Tank1.volume.fix(10.0)
m.fs.Tank1.heat.fix(0.0)

m.fs.Tank2.volume.fix(10.0)
m.fs.Tank2.heat.fix(0.0)
```

Initializing and Solving the Model

Now that the model has been constructed and the inlet and design conditions have been specified, we can now work on solving the model. However, most process engineering models cannot be solved in a single step, and require some degree of initialization to get to a solvable state. The models within the IDAES model library contain prebuilt initialization routines which can be used to get each model to a solvable state. For this tutorial, we will use a sequential modular type approach to initializing our flowsheet using these prebuilt methods.

We will begin with initializing the Feed Block, as it is the first unit in our flowsheet. The initialization routine for a Feed Block expects the conditions of the inlet stream to be provided as initial guesses along with any design conditions required to have zero degrees of freedom. However, as the conditions for the Feed are specified (fixed), there is no need to provide additional guesses for these and the initialization routine will make use of the specified value automatically.

The initialization routines also require a non-linear solver to be available to solve the model. This tutorial assumes that you have IPOPT installed, however you can substitute this for other NLP solvers you may have available. In order to do this, you can set the solver keyword when calling the initialization routine with the name of your NLP solver (e.g. *solver='ipopt'*).

```
m.fs.Feed.initialize()
```

Now that the Feed has been initialized, we can use the outlet conditions for the initial guesses for the inlet of *Tank1*. These are provided to the initialization routine through the *state_args* argument (which is a Python *dict*). To get the values from the outlet of Feed, we can use the outlet Port object, using the stream keys to access each variable. Note

that we need to specify a time index for the outlet Port, which for a steady-state model is just 0. Additionally, we need to use the Pyomo *value* method to get the actual value of the variable in question.

```
m.fs.Tank1.initialize(state_args={
        "flow_mol_comp": {
            "a": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["a"].value,
            "b": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["b"].value,
            "c": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["c"].value,
            "d": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["d"].value,
            "e": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["e"].value,
            "f": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["f"].value},
        "pressure": m.fs.Feed.outlet[0].vars["pressure"].value,
        "temperature": m.fs.Feed.outlet[0].vars["temperature"].value})
```

We can then repeat this process for *Tank2*.

```
m.fs.Tank2.initialize(state_args={
        "flow_mol_comp": {
            "a": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["a"].value,
            "b": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["b"].value,
            "c": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["c"].value,
            "d": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["d"].value,
            "e": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["e"].value,
            "f": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["f"].value},
        "pressure": m.fs.Tank1.outlet[0].vars["pressure"].value,
        "temperature": m.fs.Tank1.outlet[0].vars["temperature"].value})
```

At this point, our model should now be initialized and ready to solve. In order to do this, we need to create a *solver* using Pyomo's *SolverFactory*, which is done the same way in IDAES as in Pyomo.

```
solver = SolverFactory('ipopt')
```

Just like in Pyomo, solver options can be provided as well by attaching a dictionary of keywords to the solver object, however these are not needed for this tutorial. Once the solver object is created, we can call it to solve the model and return the results object.

```
results = solver.solve(m, tee=True)
```

Let us print the results object to get more details about our solution.

```
print(results)
```

Finally, let's display the Product Block so we can see what the conditions of the product stream are.

```
m.fs.Product.display()
```

Hopefully you get an output reporting "Optimal Solution Found". You should also see that there were 154 variables and constraints in the problem. If the number of constraints and variables are not equal, check that you specifed all the required inputs (there should be 12 variables which were fixed). You might also see some warnings at the begining about equilibrium reaction, but these can be ignored.

If all has gone well, you should see the following conditions in the Product Block.

- flow_mol_comp["a"] = 0.216 [mol/s]
- flow_mol_comp["b"] = 0.978 [mol/s]
- flow_mol_comp["c"] = 0.332 [mol/s]
- flow_mol_comp["d"] = 0.244 [mol/s]

- flow_mol_comp["e"] = 0.011 [mol/s]
- flow_mol_comp["f"] = 0.534 [mol/s]
- pressure = 101325 [Pa]
- temperature = 402.15 [K]

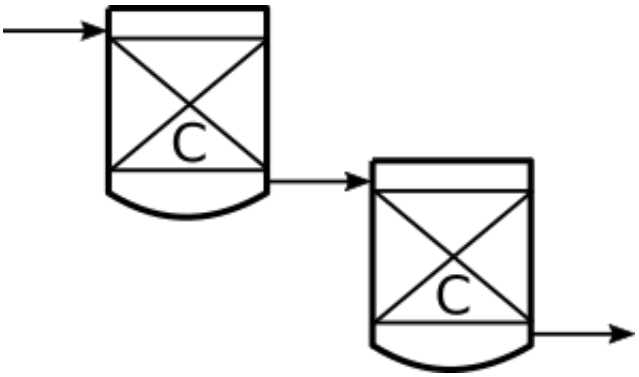
Moving On

The next tutorial will build on this one, so it is recommended that you save this tutorial so you don't have to rewrite the model code.

Tutorial 2 - Basic Flowsheet Optimization

Introduction

In the previous tutorial, we developed a model for a simple flowsheet to simulate the performance of a series of reactions occurring in two CSTRs in series. In this tutorial, we will move from simulating to optimizing the flowsheet by trying to improve the yield of specific components.



$$a + 2b \rightleftharpoons c + d$$

$$a + 2c \rightleftharpoons 2e$$

$$a + b \rightleftharpoons f$$

In this tutorial, you will learn how to:

- add an objective function to a model,
- set bounds on variables,
- add degrees of freedom to the model, and
- solve the resulting optimization problem.

First Steps

The first step in setting up an optimization problem is to create and a flowsheet for the problem and initialize it. Given that most problems of interest to engineers are complex and highly non-linear, it is general best to start with a fully defined problem and solve it for an initial set of conditions (from which we will try to find the optimal solution).

Hopefully you saved the file from the previous tutorial which we will extend to include optimization - otherwise please refer to Tutorial 1 for instructions on how to construct the flowsheet.

Next Steps

Next, we need to import a couple of additional things from Pyomo in order to create and optimization problem:

- *Objective* will be used to define our objective function, and
- *minimize* will be used to tell our solver that it needs to minimize the objective.

These can be added to the existing import from pyomo.enivron at the beginning of our file.

```
from pyomo.environ import ConcreteModel, SolverFactory, Objective, minimize
```

Adding an Objective Function

Now that we have an initialized flowsheet for our problem, we can go about adding an objective function. For this tutorial, let us consider component "f" to be an undesired side product of our reactions, and try to minimize to amount of this component produced.

For this, we add an *Objective* object to our flowsheet (*m.fs* as you may recall) and provide it with an expression for the objective function (in this case the flowrate of component *f* leaving Tank2) and an instruction on whether to minimize and maximize this expression.

```
m.fs.obj = Objective(expr=m.fs.Tank2.outlet[0].vars["flow_mol_comp"]["f"],
                     sense=minimize)
```

Adding Variable Bounds

Next, we need to add some limits on our problem to make sure the results of the optimization are physically reasonable. For this problem, we will add some bounds on the temperatures in both tanks, to make sure that they do not get too hot or too cold. For this tutorial, let us set a lower bound on temperature in both tanks to be 300 [K], and an upper bound of 400 [K] in Tank1 and 450 [K] in Tank2.

As we are using CSTRs for our reactors in this tutorial, the temperatures within the tanks are equal to the temperatures in the outlets, so we can apply our bounds to the outlet temperatures. We can set the upper and lower bounds of a variable by using the Pyomo *setub* and *setlb* methods as shown below.

```
m.fs.Tank1.outlet[0].vars["temperature"].setlb(300)
m.fs.Tank2.outlet[0].vars["temperature"].setlb(300)
m.fs.Tank1.outlet[0].vars["temperature"].setub(400)
m.fs.Tank2.outlet[0].vars["temperature"].setub(450)
```

Adding Degrees of Freedom

Finally, we need to provide our problem with some degrees of freedom that the solver can use to try to find the optimal solution to our problem. The IDAES CSTR model allows for the addition or removal of heat from the reactor, so let's use the reactor heat duties as our degrees of freedom for this problem.

When we were initializing the problem, we specified that the reactor heat duties were fixed variables with a value of 0 [J/s] to make our problem fully defined. We can now unfix these variables to allow the solver to manipulate them, and thus adjust the temperatures in the reactors.

```
m.fs.Tank1.heat.unfix()
m.fs.Tank2.heat.unfix()
```

Solving the Optimization Problem

Now that our optimization problem is fully defined, we can go ahead and try to solve it. We should already have a solver object defined within our flowsheet from Tutorial 1, so we can now apply it to the optimization problem to find our optimal solution.

```
results = solver.solve(m, tee=True)
```

Once again, let us print the results object to get more details about our solution.

```
print(results)
```

Hopefully we will see that our problem has 154 constraints and 156 variables (thus two degrees of freedom), and that an optimal solution was found.

Finally, let's display the Product Block and reactor heat duties so we can see what solution the optimizer found.

```
m.fs.Product.display()
m.fs.Tank1.heat.display()
m.fs.Tank2.heat.display()
```

If all has gone well, you should see the following conditions in the Product Block.

- flow_mol_comp["a"] = 0.497 [mol/s]
- flow_mol_comp["b"] = 1.315 [mol/s]
- flow_mol_comp["c"] = 0.279 [mol/s]
- flow_mol_comp["d"] = 0.184 [mol/s]
- flow_mol_comp["e"] = 0.006 [mol/s]
- flow_mol_comp["f"] = 0.316 [mol/s]
- pressure = 101325 [Pa]
- temperature = 300.0 [K]

You should also see the following heat duties:

- Tank1 = -7364.4 [J/s]
- Tank2 = -4201.7 [J/s]

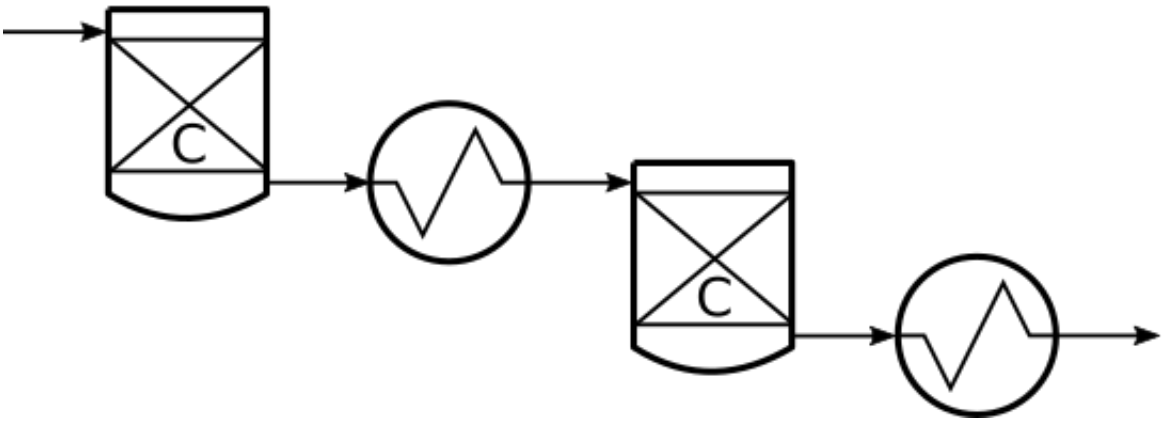
Compared to Tutorial 1, we can see that the amount of component "f" being produced has dropped from 0.534 [mol/s]. This has been achieved by reducing the temperature in both reactors by removing heat, and limiting the temperature in the system to the lower bound (300 [K]). This makes sense, as all the formation of component "f" in our example is exothermic, so reducing temperature will reduce the yield of this component. However, the other reactions are also exothermic, so we also see reduced yields of all other components.

Tutorial 3 – Advanced Flowsheets

Introduction

The previous flowsheets have demonstrated how to construct a flowsheet for a simple process with two units. However, actual processes of interest are generally far more complicated and involve many units and potential many different sets of property calculations.

In this tutorial we will we will add heat exchangers after each of the CSTRs from the previous tutorials to reduce the temperature of the reactant streams. For this, we will need to include a second property package for the cooling water to be used in the heat exchangers.



This tutorial will teach you how to:

- Add multiple property packages to a flowsheet,
- Assign a default property package to use within a flowsheet
- Assign different property packages to different unit models
- Create and utilize sub-flowsheets
- Adjust variable bounds
- Use the model_check utility

Initial Setup

Firstly, we need to import the necessary libraries we will use for our Flowsheet. This is much the same as for the last tutorial, with the addition of importing the IDAES Heat Exchanger, as well as a property package for the cooling water. A set of property calculations has been prepared for this and is available at *idaes/examples/core/tutorials/tutorial_3_H2O_properties.py*.

```
from pyomo.environ import ConcreteModel, SolverFactory

from idaes.core import FlowsheetBlock, Stream

import tutorial_1_properties as rxn_properties
import tutorial_3_H2O_properties as H2O_properties

from idaes.models import Feed, CSTR, HeatExchanger, Product
```

Next, we need to create our ConcreteModel and Flowsheet.

```
m = ConcreteModel()
m.fs = FlowsheetBlock(dynamic=False)
```

Processes with Multiple Property Packages

Most real chemical processes involve multiple groups of streams which have limited interaction with other streams. Some common examples are hot and cold utility streams which never come into direct contact with the other process streams, yet play an important role in the overall process performance.

These different groups of streams likely contain different chemical components and have different thermophysical properties which need to be calculated. Thus, we often want to include multiple sets of property calculations within a single flowsheet to represent these different groups of streams. The IDAES framework fully supports flowsheets with multiple property packages, and facilitates assigning different property packages to different unit models.

Multiple property packages can be assigned to a flowsheet in the same manner as before, as long as each has a unique name. For this tutorial, we will name our property packages *rxn_properties* and *H2O_properties*.

```
m.fs.rxn_properties = rxn_properties.PropertyParameterBlock()
m.fs.H2O_properties = H2O_properties.PropertyParameterBlock()
```

IDAES Flowsheets also support the assignment of a default property package, which is used by unit models if they are not assigned a property package at creation. The default property package for a flowsheet can be set by assigning the value of *flowsheet.config.default_property_package* to the desired property parameter block. For our flowsheet, let us assign *m.fs.rxn_properties* as the default property package.

```
m.fs.config.default_property_package = m.fs.rxn_properties
```

Next, let us add the Feed Block, first reactor and first heat exchanger to our flowsheet, which we will call *Feed*, *Tank1* and *HX1*. Adding the Feed Block and CSTRs has been covered in the previous tutorials, so we will focus on the heat exchanger models here. Like the other models, the heat exchanger model can also be given a number of arguments as instructions on how to construct the model. The most important of these are the property packages for each side of the heat exchanger; *side_1_property_package* and *side_2_property_package*. We will not worry about the other options in this tutorial.

As we have assigned *rxn_properties* as the default property package for our flowsheet, we do not need to assign it whilst adding unit models to our flowsheet (as long as they are to use this property package). Thus, for the CSTR and side 1 of our heat exchanger, we do not need to assign a property package. Thus, all we need to do to add *Feed*, *Tank* and *HX1* is the following:

```
m.fs.Feed = Feed()
m.fs.Tank1 = CSTR()
m.fs.HX1 = HeatExchanger(side_2_property_package=m.fs.H2O_properties)
```

Subflowsheets

Next, we need to add the remaining CSTR and heat exchanger to our model. Rather than attach them directly to our flowsheet, we will instead create a sub-flowsheet to contain them in order to demonstrate how to use these. IDAES flowsheet blocks (both *FlowsheetBlock* and *FlowsheetBlockData*) can be used to as a top level flowsheet or as a sub-flowsheet within a higher level flowsheet. Flowsheets can be nested to any depth desired, however some parts of the core code that need to search the model tree have an arbitrary depth limit of 20 (e.g. unit models will only search up 20 levels of flowsheets looking for a default property package).

When a *FlowsheetBlock* (or a class which inherits from *FlowsheetBlockData*) is created, it checks to see whether it has a parent object which contains a time domain (*parent.time*). If so, it assumes that it is a sub-flowsheet, and makes use of the parents time domain and *dynamic* flag, rather than creating new ones. Thus, a consistent time domain is maintained throughout all the sub-flowsheets and associated models. Each sub-flowsheet is self-contained, and can contain any of the elements a top-level flowsheet can, including property packages, unit models and connections. Sub-flowsheets may also access any component their parent (or higher level flowsheets) contains. Sub-flowsheets may also specify a different default property package from their parent, which will be used by any unit model within the sub-flowsheet in preference to that of their parent.

Creating a sub-flowsheet is done in the same manner as creating a top-level flowsheet, by creating an instance of *FlowsheetBlock* (or a class inheriting from *FlowsheetBlockData*). As a sub-flowsheet inherits its time domain and dynamic status from the parent flowsheet, it is not necessary to specify the *dynamic* argument in this case. For our example, we will also assign a different default property package for our sub-flowsheet, in this case *m.fs.H2O_properties*. This can be done in the same way as for a top-level flowsheet, however if the property package already exists (such as in this case where we have already added it to the top-level flowsheet) the default property package can be passed as an argument when creating the sub-flowsheet.

```
m.fs.block = FlowsheetBlock(default_property_package=m.fs.H2O_properties)
```

The *default_property_package* argument sets the *default_property_package* flag directly during the construction of the flowsheet.

Next, we can add the second CSTR (*Tank2*) and Heat Exchanger (*HX2*) to our new flowsheet in the same way that we would normally. As we have set *H20_properties* as our default properties of the subflowsheet, in this case we need to specify a property package for Tank2 and side 1 of HX2. Remember to assign these to *m.fs.block*.

```
m.fs.block.Tank2 = CSTR(property_package=m.fs.rxn_properties)
m.fs.block.HX2 = HeatExchanger(side_1_property_package=m.fs.rxn_properties)
```

Finally, let us add the Product Block to the Flowsheet. We could add this to either flowsheet, but for this tutorial we will add it to the top-level flowhseet (fs).

```
m.fs.Product = Product()
```

Finally, we can call *post_transform_build* on our main flowsheet to finish the construction of our model. This will automatically call *post_transform_build* on our sub-flowsheet.

```
m.fs.post_transform_build()
```

Connecting Units

Now that we have added all the unit models and sub-flowsheets to our main flowsheet, we can start connecting the units together. We will start with the units in the top-levle flowsheet:

```
m.fs.stream_1 = Stream(source=m.fs.Feed.outlet,
                       destination=m.fs.Tank1.inlet)

m.fs.stream_2 = Stream(source=m.fs.Tank1.outlet,
                       destination=m.fs.HX1.side_1_inlet)
```

We can connect units across different sub-flowsheets in exactly the same way as we connect units within a single flowsheet, just with the inclusion of the extra flowsheet leyer:

```
m.fs.stream_3 = Stream(source=m.fs.HX1.side_1_outlet,
                       destination=m.fs.block.Tank2.inlet)
```

When connecting units within subflowsheets (e.g. *Tank2* and *HX2*), be careful where you place the connecting *Constraint*. The constraint can be placed in the subflowsheet that contains the unit models, *or* in any higher level flowsheet, and the model will be constructed successfully. However, placing the connecting constraints in higher level flowsheets may become confusing, so we recommend keeping the connecting constraints with the sub-flowsheet which contains the units being connected. For our example, we would place the constraint in *m.fs.block* as shown below:

```
m.fs.block.stream_4 = Stream(source=m.fs.block.Tank2.outlet,
                             destination=m.fs.block.HX2.side_1_inlet)

m.fs.block.stream_5 = Stream(source=m.fs.block.HX2.side_1_outlet,
                             destination=m.fs.Product.inlet)
```

Setting Inlet Conditions

From here, everything follows in the same way as in the earlier tutorials. First, we need to set our inlet and design conditions. We will use the same inlet and design conditions for *Tank1* from the previous tutorials (don't forget that Tank2 is now in a sub-flowsheet).

```
m.fs.Feed.fix('flow_mol_comp', comp='a', value=1.0)
m.fs.Feed.fix('flow_mol_comp', comp='b', value=2.0)
m.fs.Feed.fix('flow_mol_comp', comp='c', value=0.1)
m.fs.Feed.fix('flow_mol_comp', comp='d', value=0.0)
m.fs.Feed.fix('flow_mol_comp', comp='e', value=0.0)
m.fs.Feed.fix('flow_mol_comp', comp='f', value=0.0)
m.fs.Feed.fix('temperature', value=303.15)
m.fs.Feed.fix('pressure', value=101325)

m.fs.Tank1.volume.fix(10.0)
m.fs.Tank1.heat.fix(0.0)

m.fs.block.Tank2.volume.fix(10.0)
m.fs.block.Tank2.heat.fix(0.0)
```

For the heat exchangers, we will set the following inlet conditions for the side_2_inlet in both HX1 and HX2. As we do not have a Feed Block for these stream, we will set the value of the varaibles directly through the Port object. The conditions we will use are:

- flow_mol = 5 [mol/s]
- temperature = 303.15 [K]
- pressure = 101325.0 [Pa]

One important thing to note is that IDAES Port objects (and many other things) are always indexed by time (even for steady-state models). Thus, when specifying conditions we need to include the time index(es) at which we wish to fix the value. For our steady-state model, there is only a single time index of *0* which we could use to specify the inlet conditions. However, more generally we use Pyomo slice notation (*:*) to set the value at all time indexes.

To fix an variable in a Connecotr object, we need to use the form:

m.fs.Tank1.inlet[:].vars["variable name"].fix(value)

If the variable we are trying to fix is indexed by something (for example a flow rate which is indexed by component), we also add the index of the variable as follows:

m.fs.Tank1.inlet[:].vars["variable name"][index].fix(value)

Thus, we need to write the follwing for our Heat Exchanger inlets:

```
m.fs.HX1.side_2_inlet[:].vars["flow_mol"].fix(5.0)
m.fs.HX1.side_2_inlet[:].vars["temperature"].fix(303.15)
m.fs.HX1.side_2_inlet[:].vars["pressure"].fix(101325.0)

m.fs.block.HX2.side_2_inlet[:].vars["flow_mol"].fix(5.0)
m.fs.block.HX2.side_2_inlet[:].vars["temperature"].fix(303.15)
m.fs.block.HX2.side_2_inlet[:].vars["pressure"].fix(101325.0)
```

We also need to set the volume for both sides of each heat exchanger, plus the heat_transfer_coefficient and heat_transfer_area variables. Let us use the following conditions (don't forget to set them for both heat exchangers).

- heat transfer coefficient = 100 [W/m^2.K]
- heat transfer area = 2.0 [m^2]

```
m.fs.HX1.heat_transfer_coefficient.fix(100.0)
m.fs.HX1.heat_transfer_area.fix(2.0)

m.fs.block.HX2.heat_transfer_coefficient.fix(100.0)
m.fs.block.HX2.heat_transfer_area.fix(2.0)
```

Using the Model Check Utility and Setting Variable Bounds

Models of chemical processes are generally very complex, and even simple mistakes can make a problem difficult or impossible to solve. In order to assist users with solving models (and debugging them when they fail), the IDAES framework contains tools which allow model developers to write simple tests for common problems that may arise when using their models. Some examples of potential model tests include:

- checking for variables with values set outside of the variable bounds,
- "sanity checks" for model behavior - e.g. a compressor with a negative pressure change.

IDAES flowsheets contain a method called *model_check* which searches through all the unit models attached to the flowsheet and calls the models *model_check* method (if it exists). The developer of each unit model (and each submodel and property package) is responsible for writing any model checks which makes sense for their particular model.

All the core IDAES unit models contain model checks, and the *H2O_properties* package contains a simple *model_check* method which checks that the *temperature* and *pressure* variables fall within the bounds set for those variables. Let's call the *model_check* method on our main flowsheet (*m.fs*) and see the results (you will need to run the flowsheet to see the results).

```
m.fs.model_check()
```

You should see an output saying "INFO - idaes.core - Executing model checks.". This indicates that the *model_check* method has been called, and will be followed by any outputs from the *model_check* methods. In this case, there should be no other outputs, as all the checks for our model should pass.

In order to demonstrate a model check which fails, let's change the lower bound on *temperature* in one of our heat exchangers to 310.0 K (which is higher than our fixed inlet value). AS discussed in the last tutorial, the upper and lower bounds of a variable can be set using the *setub* and *setlb* methods respectively, which all Pyomo variables have. To set the lower bound of the side 2 inlet temperature to *HX1*, we use the following code (this needs to go before the model_check call):

```
m.fs.HX1.side_2_inlet[:].vars["temperature"].setlb(310.0)
```

CAUTION - users should be careful when changing preset variable bounds in any model, especially when trying to widen the bounds. Many models are fitted to a limited range of data, and bounds are set to prevent these from being extrapolated beyond the fitted region where the quality of the fit may not be guaranteed (or in many cases, where the fit is known to be bad). Users should keep this in mind when looking at and adjusting bounds in models they did not create.

Another thing users should be aware of is that bounds are set independently for each instance of a model. In our current example, there are multiple instances of the *H2O_properties* model throughout our flowsheet (2 instances in each unit model to be precise, for a total of 8). When we changed the lower bound above, we changes the bound in only one of these instances, and the remaining 7 instances still have the old bound set.

Now that we have changed the lower bound on temperature, let us run the model checks again. This time we would expect to see a warning that our inlet temperature has a value lower than the lower bound.

We should now see a second line in the output from the *model_check* method, which says "ERROR - idaes.unit_model.properties - fs.HX1.side_2.properties_in[0.0] Temperature set below lower bound.".

The first part of the message tells us the severity of the problem found, in this case "ERROR". IDAES model checks use the following levels for model checks:

- ERROR - indicates an issue that is likely to cause a problem when solving the model (e.g. variables out of bounds),
- WARNING - indicates something that may be wrong, but may still be solvable (e.g. a compressor with negative pressure increase),
- INFO - message for information only.

The second part of the message indicates which part of the IDAES framework raised the issue. This is mostly useful for debugging by advanced users and won't be covered here.

The important part of the output comes from the last two parts of the message. The third part tell us which part of our particular model the issue came from, in this case *fs.HX1.side_2.properties_in[0.0]*. We already know that *fs.HX1* is our first heat exchanger, and *side_2.properties_in[0.0]* indicates that the issue came from the inlet properties for side 2 of this heat exchanger (at time 0.0). More detail on understanding the internal structure of IDAES unit models will be given in the next tutorial.

The final part of the message was written by the developer of the *H2O_properties* package, and tells us what caused the issue - in this case the lower bound of the *temperature* variable. Hopefully this message will make it clear what needs to be fixed, however the quality of these messages depends on the model developer.

Before we continue, let us change the lower bound on temperature back to its original value (298.15 K) so that our model is feasible.

Initializing and Solving the Model

Next we need to initialize our model. We will use the same procedure as in previous tutorials, sequentially initializing units using the outlet of the previous unit (the side_2 inlets of the Heat Exchangers are fixed, so we do not need to provide conditions for these).

```
m.fs.Feed.initialize()
m.fs.Tank1.initialize(state_args={
        "flow_mol_comp": {
            "a": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["a"].value,
            "b": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["b"].value,
            "c": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["c"].value,
            "d": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["d"].value,
            "e": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["e"].value,
            "f": m.fs.Feed.outlet[0].vars["flow_mol_comp"]["f"].value},
```

(continues on next page)

(continued from previous page)

```
        "pressure": m.fs.Feed.outlet[0].vars["pressure"].value,
        "temperature": m.fs.Feed.outlet[0].vars["temperature"].value})
m.fs.HX1.initialize(state_args_1={
        "flow_mol_comp": {
            "a": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["a"].value,
            "b": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["b"].value,
            "c": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["c"].value,
            "d": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["d"].value,
            "e": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["e"].value,
            "f": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["f"].value},
        "pressure": m.fs.Tank1.outlet[0].vars["pressure"].value,
        "temperature": m.fs.Tank1.outlet[0].vars["temperature"].value})

m.fs.block.Tank2.initialize(state_args={
    "flow_mol_comp": {
        "a": m.fs.HX1.side_1_outlet[0].vars["flow_mol_comp"]["a"].value,
        "b": m.fs.HX1.side_1_outlet[0].vars["flow_mol_comp"]["b"].value,
        "c": m.fs.HX1.side_1_outlet[0].vars["flow_mol_comp"]["c"].value,
        "d": m.fs.HX1.side_1_outlet[0].vars["flow_mol_comp"]["d"].value,
        "e": m.fs.HX1.side_1_outlet[0].vars["flow_mol_comp"]["e"].value,
        "f": m.fs.HX1.side_1_outlet[0].vars["flow_mol_comp"]["f"].value},
    "pressure": m.fs.HX1.side_1_outlet[0].vars["pressure"].value,
    "temperature": m.fs.HX1.side_1_outlet[0].vars["temperature"].value})
m.fs.block.HX2.initialize(state_args_1={
    "flow_mol_comp": {
        "a": m.fs.block.Tank2.outlet[0].vars["flow_mol_comp"]["a"].value,
        "b": m.fs.block.Tank2.outlet[0].vars["flow_mol_comp"]["b"].value,
        "c": m.fs.block.Tank2.outlet[0].vars["flow_mol_comp"]["c"].value,
        "d": m.fs.block.Tank2.outlet[0].vars["flow_mol_comp"]["d"].value,
        "e": m.fs.block.Tank2.outlet[0].vars["flow_mol_comp"]["e"].value,
        "f": m.fs.block.Tank2.outlet[0].vars["flow_mol_comp"]["f"].value},
    "pressure": m.fs.block.Tank2.outlet[0].vars["pressure"].value,
    "temperature": m.fs.block.Tank2.outlet[0].vars["temperature"].value})
```

Now that our model is initialized, we can create a solver and use it to solve our flowsheet.

```
solver = SolverFactory('ipopt')
results = solver.solve(m, tee=True)
```

Finally, let's display the both of the outlets from HX2 (side_1_outlet and side_2_outlet) and check our results.

```
print(results)

m.fs.block.HX2.side_1_outlet.display()
m.fs.block.HX2.side_2_outlet.display()
```

The expected conditions are:

Side 1

- flow_mol_comp["a"] = 0.275 [mol/s]
- flow_mol_comp["b"] = 1.047 [mol/s]
- flow_mol_comp["c"] = 0.323 [mol/s]
- flow_mol_comp["d"] = 0.233 [mol/s]
- flow_mol_comp["e"] = 0.010 [mol/s]

- flow_mol_comp["f"] = 0.487 [mol/s]
- pressure – 101325.0 [Pa]
- temperature = 324.82 [K]

Side 2

- flow_mol = 5.0 [mol/s]
- pressure = 101325.0 [Pa]
- temperature = 318.79 [K]

Tutorial 4 – Dynamic Flowsheets

Introduction

Up until now, we have only looked at steady-state flowsheets, so in this tutorial we will look at how to build a dynamic flowsheet. To do this, we will modify the process from Tutorial 1 to simulate dynamic behavior by including:

- accumulation and holdup,
- a Mixer for the different feed components,
- pressure driven flow leaving each Tank, and
- a step-change in feed rates.

This tutorial will teach you how to:

- Set up a dynamic flowsheet,
- Add steady-state models to dynamic flowsheets,
- Add constraints to existing unit models,
- Transform the time domain using Pyomo.dae,
- set steady-state initial conditions, and
- plot some results.

First Steps

Firstly, we need to import the necessary libraries we will use for our Flowsheet. In addition to the imports from Tutorial 1, we need to import *Constraint*, *Var* and *TransformationFactory*.

```
from pyomo.environ import ConcreteModel, Constraint, SolverFactory,␣
↪TransformationFactory, Var

from idaes.core import FlowsheetBlock, Stream

import tutorial_1_properties as properties_rxn
from idaes.models import CSTR, Mixer
```

We also need to import a library to use to plot our results, for which we will use matplotlib:

```
import matplotlib.pyplot as plt
```

Dynamic Flowsheets

In order to create a dynamic model for a flowsheet, we start with a ConcreteModel as before.

```
m = ConcreteModel()
```

When we add our FlowsheetBlock however, we need to indicate that this will be a dynamic flowsheet. We have already seen that FlowsheetBlock, which we have so far set to *False* (this is the default setting by the way, so it is not necessary to specify *dynamic* = *False* for steady-state flowsheets). For a dynamic Flowsheet, we simply need to specify *dynamic* = *True* when creating our FlowsheetBlock.

FlowsheetBlock also accepts another argument named *time_set* that can be used to specify a set of points within the time domain. The time domain needs to be set with a start and end point at a minimum, and the user can specify additional intermediate points if desired. *time_set* expects a list of points, and has the following defaults:

- if dynamic = True, time_set = [0.0, 1.0]
- if dynamic = False, time_set = [0.0]

For our flowsheet, let us add the following points to our time domain:

- Start point = 0.0 s
- End point = 5e5 s
- Internal point at 1.0 s

Thus, we write the following to create our FlowsheetBlock:

```
m.fs = FlowsheetBlock(dynamic=True, time_set=[0, 1, 500000])
```

Adding Property Packages and Models

Dynamic FlowsheetBlocks behave the same way as steady-state FlowsheetBlocks, so we can add our property packages and unit models in the same manner as before. First, we need to create the property package, and we will also set it as the default property package for our flowsheet.

```
m.fs.properties_rxn = properties_rxn.PropertyParameterBlock()
m.fs.config.default_property_package = m.fs.properties_rxn
```

Whilst we have declared our flowsheet to be a dynamic process, there are many circumstances where we would have unit operations which react sufficiently quickly that we would like to consider them to be steady-state operations. The IDAES framework supports this by allowing us to declare Unit Models as steady-state model by specifying *dynamic* = *False* when declaring Unit Models (note that the reverse is not supported, i.e. dynamic models in steady-state flowsheets). If the *dynamic* argument is not provided when a Unit Model is added to a Flowsheet, it takes this argument from its parent object. This can also be applied to sub-flowsheets, allowing for steady-state sub-Flowsheets within dynamic Flowsheets.

For this tutorial, let us add a Mixer unit at the beginning of our process to mix two feed streams prior to being fed to Tank1. We will assume this is an ideal mixer, which means that we can assume it can be represented by a steady-state model.

```
m.fs.Mix = Mixer(dynamic=False)
```

Next, we can add our two Tank models as before. As we have set a default property package and we want our CSTRs to be dynamic models, we do not need to provide any arguments when adding these to our Flowsheet (the dynamic argument will be inherited from the Flowsheet).

```
m.fs.Tank1 = CSTR()
m.fs.Tank2 = CSTR()
```

Adding Variables and Constraints to Existing Models

For our dynamic process, we want to model the flow of material leaving each tank as being pressure driven flow; that is flowrate leaving each tank is proportional to the depth of fluid in each tank. However, the core CSTR model does not include the calculations necessary for modeling this behavior. Thus, we will need to add these Constraints ourselves.

First, we will need to add some additional Variables to our Tank models that we will use in our calculations. We can do this by simply adding Pyomo *Var* objects to the CSTR objects already constructed in our Flowsheet. We need to add the following Variables to Tank1 (Note: *volume_flow* is a state variable and thus technically belongs in a Property Block. As we have not covered these yet, we will place it in the Unit Model for now).

```
m.fs.Tank1.height = Var(m.fs.Tank1.time,
                        initialize=1.0,
                        doc="Depth of fluid in tank [m]")
m.fs.Tank1.area = Var(initialize=1.0,
                      doc="Cross-sectional area of tank [m^2]")
m.fs.Tank1.volume_flow = Var(m.fs.Tank1.time,
                             initialize=4.2e5,
                             doc="Volumetric flow leaving tank")
m.fs.Tank1.flow_coeff = Var(m.fs.Tank1.time,
                            initialize=5e-5,
                            doc="Tank outlet flow coefficient")
```

Next, we can add the extra Constraints we need. This is done using Pyomo *Constraint* objects, which again can be added directly to our CSTR objects. To create a Constraint, we first need to write a rule describing the equation we wish to write, and to then create a Constraint using this rule. Let us start with a Constraint relating the volume of reacting fluid, V_t, to the cross-sectional area of the tank, A, and the depth of the fluid, h_t, at a given point in time t (we will assume A is constant with time):

$$V_t = A \times h_t$$

To do this, we first write the following rule:

```
def geometry(b, t):
    return b.volume[t] == b.area*b.height[t]
```

This is a Python method that returns an expression involving Pyomo objects (volume, area and height). In this rule we have used *b* to represent the object to which we are adding the Constraint (and from which we will gather these variables). When we create the Constraint in the next step, Pyomo automatically passes the object that the Constraint is being added to (in this case Tank1) to the rule, allowing us to write the rule in a general form.

We then use this rule to construct the actual Constraint in our model.

```
m.fs.Tank1.geometry = Constraint(m.fs.Tank1.time, rule=geometry)
```

We also need to write the following Constraints for *volume_flow*:

$$F_{vol,t} = F_{mol,t}/\rho_{mol,t}$$

$$F_{vol,t} = C_t \times h_t$$

where $F_{vol,t}$ is volume_flow, $F_{mol,t}$ is the total molar flowrate of material leaving the tank, $\rho_{mol,t}$ is the total density of the fluid in the tank and C_t is flow_coeff. $F_{mol,t}$ and $\rho_{mol,t}$ both come from the Property Block associated with the

fluid in the tank (and thus the tank outlet), which is called *Tank1.holdup.properties_out*. To write these Constraints, we use the following code:

```
def volume_flow_calculation(b, t):
    return b.volume_flow[t] == (
                b.holdup.properties_out[t].flow_mol /
                b.holdup.properties_out[t].dens_mol_mol['Liq'])
m.fs.Tank1.volume_flow_calculation = Constraint(
        m.fs.Tank1.time,
        rule=volume_flow_calculation)

def outlet_flowrate(b, t):
    return b.volume_flow[t] == b.flow_coeff[t]*b.height[t]
m.fs.Tank1.outlet_flowrate = Constraint(m.fs.Tank1.time,
                                        rule=outlet_flowrate)
```

We then need to repeat this for Tank2.

```
m.fs.Tank2.height = Var(m.fs.Tank2.time,
                        initialize=1.0,
                        doc="Depth of fluid in tank [m]")
m.fs.Tank2.area = Var(initialize=1.0,
                      doc="Cross-sectional area of tank [m^2]")
m.fs.Tank2.volume_flow = Var(m.fs.Tank2.time,
                             initialize=4.2e5,
                             doc="Volumetric flow leaving tank")
m.fs.Tank2.flow_coeff = Var(m.fs.Tank2.time,
                            initialize=5e-5,
                            doc="Tank outlet flow coefficient")
```

When it comes to adding the Constraints to Tank2, we can reuse the rules we wrote for Tank1, as we wrote these in a general form. When we create the Constraints in Tank2 now, the Variables in Tank2 will be used instead.

```
m.fs.Tank2.geometry = Constraint(m.fs.Tank2.time, rule=geometry)
m.fs.Tank2.volume_flow_calculation = Constraint(
        m.fs.Tank2.time,
        rule=volume_flow_calculation)
m.fs.Tank2.outlet_flowrate = Constraint(m.fs.Tank2.time,
                                        rule=outlet_flowrate)
```

Transforming the Time Domain

At this point in previous tutorials, we would now call *post_transform_build* to finish constructing our models. However, in dynamic flowsheets, we first need to transform the differential equations into a form that can be handled by our solver. When the time domain is created for a dynamic flowsheet, it is created as a Pyomo ContinuousSet object and the associated derivatives as DerivativeVars. However, most solvers do not understand these types of objects so they need to be transformed prior to solving. This is done using Pyomo's TransformationFactory; for more details on what happens during DAE transformation, see the Pyomo documentation.

Pyomo provides support for a number of types of DAE transformation:

- finite difference methods - backward, forward and central difference methods, and
- orthogonal collocation methods - Lagrange-Radau and Lagrange-Legendre roots.

For time domains, we generally use a 1st order backward finite difference method.

To use Pyomo's DAE Transformation, we first need to create a Transformation object, and to then apply it to our model object.

```
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(m.fs,
                     nfe=200,
                     wrt=m.fs.time,
                     scheme='BACKWARD')
```

Once the DAE transformation has been applied to the time domain, we can call *post_transform_build* to finish the model construction.

```
m.fs.post_transform_build()
```

Connecting Units

As before, we cannot start connecting units together until *post_transform_build* has been called. Now that that is done, we can add Streams as necessary:

```
m.fs.stream_1 = Stream(source=m.fs.Mix.outlet,
                       destination=m.fs.Tank1.inlet)

m.fs.stream_2 = Stream(source=m.fs.Tank1.outlet,
                       destination=m.fs.Tank2.inlet)
```

Setting Design and Operating Constraints

For this tutorial, let us split our feed into two parts (with the same temperature and pressure):

- a stream with 10.0 mol/s of "a" and 1.0 mol/s of "c"
- a stream with 20.0 mol/s of "b"

As we did not add Feed Blocks to this flowsheet, we will fix these through the inlet Port objects of the Mixer. Note that for a Mixer, there is a single inlet object which is indexed by time and inlet name. Thus, when we fix the feed conditions, we need to do so at all points in time for each inlet. We can do this using slice notation as shown below:

```
m.fs.Mix.inlet[:, "1"].vars["flow_mol_comp"]["a"].fix(10.0)
m.fs.Mix.inlet[:, "1"].vars["flow_mol_comp"]["b"].fix(0.0)
m.fs.Mix.inlet[:, "1"].vars["flow_mol_comp"]["c"].fix(1.0)
m.fs.Mix.inlet[:, "1"].vars["flow_mol_comp"]["d"].fix(0.0)
m.fs.Mix.inlet[:, "1"].vars["flow_mol_comp"]["e"].fix(0.0)
m.fs.Mix.inlet[:, "1"].vars["flow_mol_comp"]["f"].fix(0.0)
m.fs.Mix.inlet[:, "1"].vars["temperature"].fix(303.15)
m.fs.Mix.inlet[:, "1"].vars["pressure"].fix(101325.0)

m.fs.Mix.inlet[:, "2"].vars["flow_mol_comp"]["a"].fix(0.0)
m.fs.Mix.inlet[:, "2"].vars["flow_mol_comp"]["b"].fix(20.0)
m.fs.Mix.inlet[:, "2"].vars["flow_mol_comp"]["c"].fix(0.0)
m.fs.Mix.inlet[:, "2"].vars["flow_mol_comp"]["d"].fix(0.0)
m.fs.Mix.inlet[:, "2"].vars["flow_mol_comp"]["e"].fix(0.0)
m.fs.Mix.inlet[:, "2"].vars["flow_mol_comp"]["f"].fix(0.0)
m.fs.Mix.inlet[:, "2"].vars["temperature"].fix(303.15)
m.fs.Mix.inlet[:, "2"].vars["pressure"].fix(101325.0)
```

We also need to fix values for area, flow_coeff and heat duty in both Tanks.

```
m.fs.Tank1.area.fix(0.5)
m.fs.Tank1.flow_coeff.fix(5e-6)
m.fs.Tank1.heat.fix(0.0)

m.fs.Tank2.area.fix(0.5)
m.fs.Tank2.flow_coeff.fix(5e-6)
m.fs.Tank2.heat.fix(0.0)
```

Setting Initial Conditions

When setting up dynamic models, we also need to provide initial conditions for the system. One approach is to specify that the system is at steady-state initially (i.e. all time derivatives are equal to zero at the starting time). In order to simplify specifying this, IDAES FlowsheetBlocks have the following method:

```
m.fs.fix_initial_conditions('steady-state')
```

Steady-state initial conditions are often only useful for modeling simple systems under ideal circumstances, and users should choose a set of initial conditions that suit their actual circumstances.

Initializing the Flowsheet

The best method for initializing dynamic problems is an open question, and it is often difficult to find a good way to do this. For this tutorial, it is sufficient to initialize the entire model at a single, steady-state operating condition and to then introduce a time disturbance later. Thus, we can just use the same approach that we have used previously to initialize the flowsheet.

```
m.fs.Mix.initialize()
m.fs.Tank1.initialize(state_args={
        "flow_mol_comp": {
            "a": m.fs.Mix.outlet[0].vars["flow_mol_comp"]["a"].value,
            "b": m.fs.Mix.outlet[0].vars["flow_mol_comp"]["b"].value,
            "c": m.fs.Mix.outlet[0].vars["flow_mol_comp"]["c"].value,
            "d": m.fs.Mix.outlet[0].vars["flow_mol_comp"]["d"].value,
            "e": m.fs.Mix.outlet[0].vars["flow_mol_comp"]["e"].value,
            "f": m.fs.Mix.outlet[0].vars["flow_mol_comp"]["f"].value},
        "pressure": m.fs.Tank1.outlet[0].vars["pressure"].value,
        "temperature": m.fs.Tank1.outlet[0].vars["temperature"].value})
m.fs.Tank2.initialize(state_args={
        "flow_mol_comp": {
            "a": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["a"].value,
            "b": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["b"].value,
            "c": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["c"].value,
            "d": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["d"].value,
            "e": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["e"].value,
            "f": m.fs.Tank1.outlet[0].vars["flow_mol_comp"]["f"].value},
        "pressure": m.fs.Tank1.outlet[0].vars["pressure"].value,
        "temperature": m.fs.Tank1.outlet[0].vars["temperature"].value})
```

Before moving on, we should also solve the entire flowsheet at our initial conditions, as the Constraints within the Streams are not yet initialized (even though the values on both sides should be the same).

```
solver = SolverFactory('ipopt')
results = solver.solve(m, tee=True)
```

Creating a Disturbance

Now that our model is initialized, let's create a disturbance in the system so that we can observe the response. For now, let us halve the flowrate of component "b" at time 1, which is done as follows:

```
for t in m.fs.time:
    if t >= 1.0:
        m.fs.Mix.inlet[t, "2"].vars["flow_mol_comp"]["b"].fix(10.0)
```

Then, we can solve the model with the new disturbance and print the results so that we can see if the problem converged.

```
results = solver.solve(m, tee=True)
print(results)
```

Hopefully you will see that the solver found an optimal solution, and that the problem had 40588 variables and constraints.

Plotting the Results

The best way to observe the results of our disturbance is to plot the flowrates of the different components leaving Tank2. To do this, we first need to collect the time and flow information in a form that can be understood by our plotting library.

To begin with, we create an empty list to hold the values of each variable we wish to plot (and the time domain).

```
time = []
a = []
b = []
c = []
d = []
e = []
f = []
```

Next, we iterate over the time domain, and append the value of each variable to the appropriate list.

```
for t in m.fs.time:
    time.append(t)
    a.append(m.fs.Tank2.outlet[t].vars["flow_mol_comp"]["a"].value)
    b.append(m.fs.Tank2.outlet[t].vars["flow_mol_comp"]["b"].value)
    c.append(m.fs.Tank2.outlet[t].vars["flow_mol_comp"]["c"].value)
    d.append(m.fs.Tank2.outlet[t].vars["flow_mol_comp"]["d"].value)
    e.append(m.fs.Tank2.outlet[t].vars["flow_mol_comp"]["e"].value)
    f.append(m.fs.Tank2.outlet[t].vars["flow_mol_comp"]["f"].value)
```

Next, we need to create a plot object, and to add each curve that we wish to plot.

```
plt.figure(1)
plt.plot(time, a, label='a')
plt.plot(time, b, label='b')
plt.plot(time, c, label='c')
plt.plot(time, d, label='d')
plt.plot(time, e, label='e')
plt.plot(time, f, label='f')
```

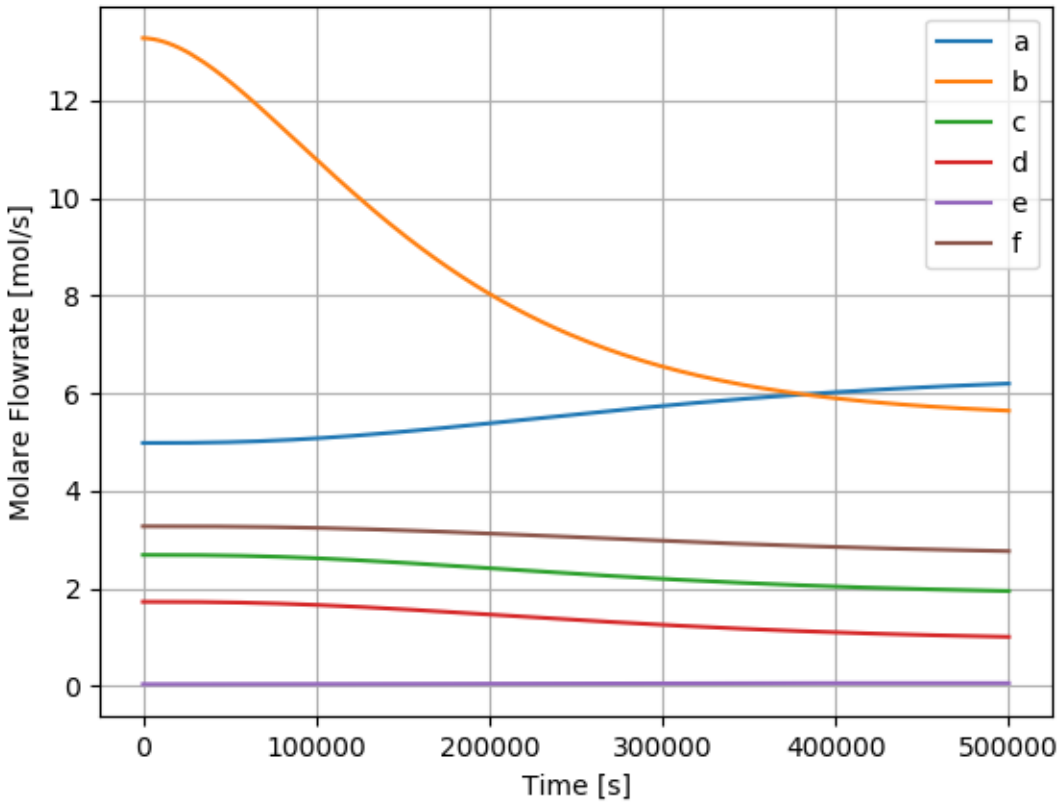
We can also add legends and axis labels as follows:

```
plt.legend()
plt.grid()
plt.xlabel("Time [s]")
plt.ylabel("Molar Flowrate [mol/s]")
```

Finally, we need to display the plot object so that we can see it.

```
plt.show(block=True)
```

If all goes well, you should see something like this:



Tutorial 5 – Modifying Unit Models

Introduction

In the previous tutorial we, we added a set of Variables and Constraints to two CSTRs to simulate pressure driven flow. In cases where we would like to make repeated use of the same modifications to a model, it would be convenient to create a new model with these modification instead of making the changes to each instance of a simpler model. This tutorial will teach you how to take an existing model and add modifications to it such that it can be used repeatedly in a flowsheet.

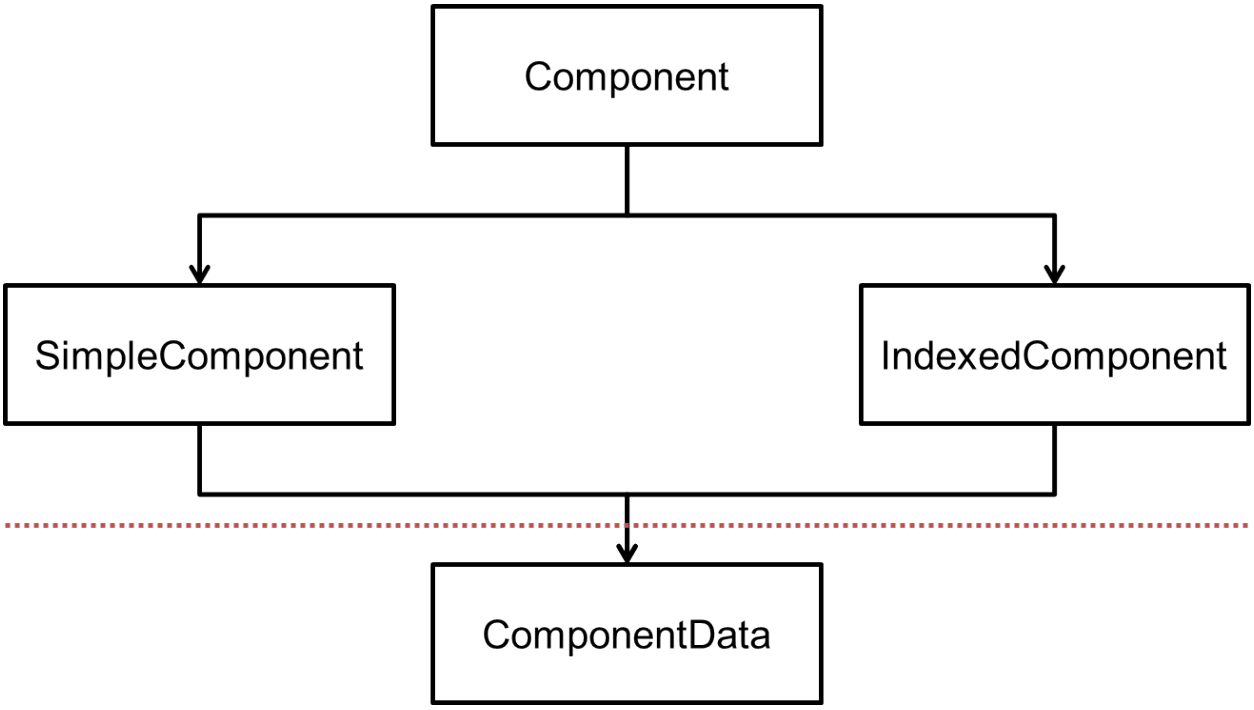
For this tutorial, we will create a new model for a CSTR with pressure driven flow.

This tutorial will teach you about:

- Pyomo component structure,
- building on an existing unit model (class inheritance),
- replacing existing methods (overloading), and
- adding new components to a unit model.

Pyomo Component Structure

The first thing we need to cover before we go further is to briefly introduce Pyomo's underlying component structure, and how IDAES interacts with this. When we add a Pyomo component to a model, we call the Component class along with some arguments (for example, Block). Behind the scenes, the Pyomo Component class then checks whether the component is indexed or not, and calls either SimpleComponent or IndexedComponent as appropriate (i.e. SimpleBlock or IndexedBlock). This class then calls upon a ComponentData class (i.e. BlockData) to populate the component with the necessary information. This is shown in the figure below.



As model developers in IDAES, we are most interested in the BlockData class, as this is where we write the instructions for a given instance of a Block - i.e. the Variables and Constraints that make up our models. We would prefer not the have to worry about the other three classes, as these are instructions on how to construct the underlying objects (and are standard across all our models).

IDAES handles all of this for us using the *declare_process_block_class* decorator, so that we only need to write the BlockData object for our model. The remaining three class are handled through meta-class, and those who wish to know more should read the documentation on process_block. For the rest of us, it is important to keep this behavior in mind, as it affects the underlying class structure we are working with.

First Steps

The first steps as always are to import the necessary components from the Pyomo and IDAES libraries. First, we will need to import Var from Pyomo and declare_process_block_class from IDAES.

```
from pyomo.environ import Var
from idaes.core import declare_process_block_class
```

We also need to import the model we are going to use as the basis for our new model, which in our case is CSTR. However, the class we want to modify is the CSTRData class (the BlockData class associated with the CSTR model) instead of CSTR (the equivalent Block class for a CSTR).

```
from idaes.models.cstr import CSTRData
```

Inheriting from Existing Models

The next step is to create a new set of model classes for our CSTR with pressure driven flow, which we will call CSTR2 for this tutorial. To do this, we need to declare a new class (which will form our BlockData class) which inherits from CSTRData, and to apply the declare_process_block_class decorator to it. This is done as shown below:

```
@declare_process_block_class("CSTR2")
class CSTR2Data(CSTRData):
```

The first line above is the decorator, which creates a new set of meta-classes with the name CSTR2 for us, based on the class declared in the second line. This handles all the work of setting up the CSTR2, SimpleCSTR2 and IndexedCSTR2 classes for us, so that we don't need to worry about them. The second line declares our new CSTR2Data class, which inherits from CSTRData (as indicated by the argument in the parentheses).

The build Method

In order to populate our model with the necessary Variables and Constraints, we need to point Pyomo to a set of instructions for creating these. For this, we create a method named *build*, which is called by default whenever we add a model to a Flowsheet. This build method belongs in the CSTR2Data class.

For our new pressure-driven CSTR model, our build method needs to do two things:

1. construct the base CSTR model, and,
2. add the Constraints for pressure-driven flow.

The base CSTR model which we have inherited from has its own build method, which contains the instructions for constructing a standard CSTR. Rather than rewrite all these instructions, we can instead call the original CSTR model's build method to do this for us. We do this by making use of Python's super() method, which allows us to access the methods of our parent class. We will also call a second method (which we haven't written yet) to add the pressure-driven flow constraints. All of this is shown below:

```
def build(self):
    super(_CSTRData, self).build()

    self.add_pressure_driven_flow()
```

Adding New Methods, Variables and Constraints

Next, we will declare the new add_pressure_driven_flow method, which will contain the instructions for adding the Variables and Constraints we need for pressure-driven flow.

```
def add_pressure_Driven_flow(self):
```

Here, self is a standard placeholder for the object to which the method belongs (in this case our new model). Within this method, we can now write the code necessary to declare the new Variables and Constraints much like we did in Tutorial 3.

We need to add the following:

- height, area, volume_flow and flow_coeff variables,
- $V_t = A \times h_t$ Constraint,
- $F_{vol,t} = F_{mol,t} \times \rho_{mol,t}$ Constraint, and,
- $F_{vol,t} = C_t \times h_t$ Constraint.

All of this is shown below (along with the method declaration - note the indentation).

```
def add_pressure_driven_flow(self):
    self.height = Var(self.time,
                      initialize=1.0,
                      doc="Depth of fluid in tank [m]")
    self.area = Var(initialize=1.0,
                    doc="Cross-sectional area of tank [m^2]")

    self.volume_flow = Var(self.time,
                           initialize=4.2e5,
                           doc="Volumetric flow leaving tank")

    self.flow_coeff = Var(self.time,
                          initialize=5e-5,
                          doc="Tank outlet flow coefficient")

    def geometry(b, t):
        return b.volume[t] == b.area*b.height[t]
    self.geometry = Constraint(self.time, doc="Tank geometry constraint",␣
↪rule=geometry)

    def volume_flow_calculation(b, t):
        return b.volume_flow[t] == (
                    b.holdup.properties_out[t].flow_mol /
                    b.holdup.properties_out[t].dens_mol_phase['Liq'])
    self.volume_flow_calculation = Constraint(self.time, doc="Flow volume constraint",
↪ rule=volume_flow_calculation)

    def outlet_flowrate(b, t):
        return b.volume_flow[t] == b.flow_coeff[t]*b.height[t]
    self.outlet_flowrate = Constraint(self.time, doc="Outlet flow correlation",␣
↪rule=outlet_flowrate)
```

Replacing Other Methods

The above is all we need to change for this tutorial, however with more complex models the are other methods that we might need to replace. The three most important ones are:

- post_transform_build
- model_check
- initialize

In our case, the existing versions of these we inherit from CSTR are sufficient for our this tutorial, however in many cases we will need to make changes to these as well. If there is anything that needs to be done after the DAE transformation (such as adding new inlets or outlets), this needs to go in *post_transform_build*. We can also write model_check and initialize methods customized for our new model if needed.

Using Our New Model

To put our new model to use, all we need to do is import our CSTR2 class (the one created by the decorator), and use it in the same way as we have used CSTR previously. Try to repeat Tutorial 3 using our new class instead - all you should need to do is use CSTR2 in place of CSTR (and skip the part on adding the new Variables and Constraints - our new model does this for us).

Tutorial 6 – Creating New Unit Models

Introduction

The previous tutorial demonstrated how we can modify existing unit models, but we will often run into situations where there is no existing model that is suitable for our needs. In these cases, we will need to build a model for our unit from the ground up.

The IDAES modeling framework has been built to provide a large degree of flexibility to the user when developing new models, whilst providing tools to facilitate many of the common tasks associated with developing new models. This tutorial will guide you through the development of a new unit model using the IDAES CSTR unit model as an example. The actual IDAES CSTR unit model can be found at *idaes/core/models/cstr.py* for those who wish to see the final code.

This tutorial will teach you about:

- IDAES modeling standards,
- IDAES core modeling classes,
- Creating a new unit model,
- Config blocks and how they are used,
- Holdup blocks and how to interact with them,
- the *build_inlets* and *build_outlets* methods, and,
- writing initialization methods.

IDAES Modeling Standards

The IDAES modeling framework relies on Flowsheets, Unit Models and Property Packages being able to communicate to each other with a minimum of effort on the part of the user. In order to do this, IDAES has developed a set of *modeling standards* which all models should conform to. For developers of unit models, the most important of these to be aware of is the IDAES standard naming convention for thermophysical, transport and reaction properties. The standard names for all properties are listed in the *IDAES standards documentation*.

Components of Unit Models

Whilst every unit model is different, there are many common features between them. Every unit model will have some number of inlets and outlets, and at least one set of balance equations. The IDAES modeling framework endeavors to

facilitate the construction of these common features by providing a set of core classes and methods to automate their construction.

All unit models in IDAES start by inheriting from the IDAES *UnitBlockData* class. This class contains a set of methods for automating things like setting up the time domain and creating the inlets and outlets to the unit. Users can find documentation on the UnitBlockData class in the *Unit Model* documentation.

The next class used in developing unit models within IDAES are the *Holdup* block classes. These classes are used to automatically generate the material, energy and momentum balances for a specified volume of material (control volume) within the unit. The user is able to provide a simple set of instructions on what terms they wish to include in a given Holdup, and a unit model may contain as many *Holdup* blocks as required. The IDAES modeling framework currently supports three different types of Holdup blocks, and details of each can be found in the IDAES documentation for *Holdup Blocks*.

- *0D Holdup Block*
- *1D Holdup Block*
- *Static Holdup Block*

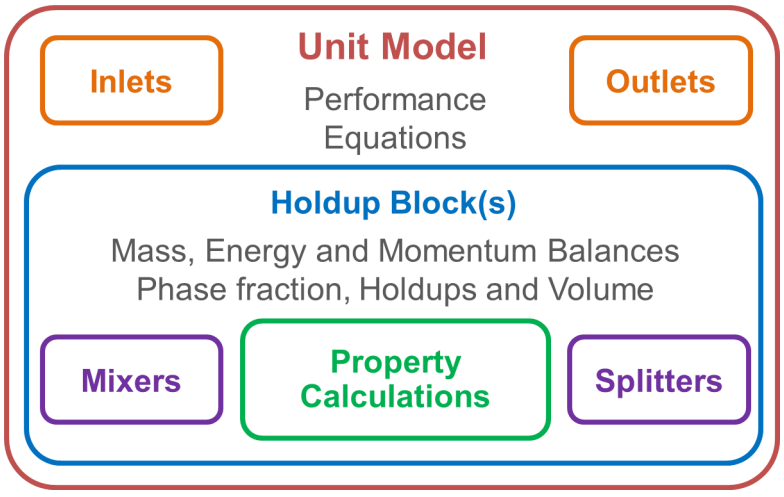
Some additional classes that are indirectly with the development of unit models, which you will not need to use directly but should be aware of are:

- *Property Package classes*
- *Inlet Mixer and Outlet Splitter classes*

As we have seen in earlier tutorials, property packages are used to calculate the various properties required for solving process models. Within the IDAES framework, each Holdup block is associated with a number of Property Blocks, each of which represents a single material state within the unit.

Inlet Mixers and Outlet Splitters are used to support multiple streams connecting to a single inlet or outlet in a unit model. These are used within IDAES for developing general mixer, splitter and separator models, as well as for developing superstructure based process synthesis problems.

The general structure of an IDAES unit model is shown in the figure below.



Creating a New Unit Model

As with any model, the first steps in developing a new unit model are to import the necessary components from the Pyomo and IDAES libraries. For this tutorial, the first thing we need to import is the *Var* object from pyomo.environ.

```
from pyomo.environ import Var
```

The next thing we are going to need is *ConfigValue* from pyomo.common.config. pyomo.common is a library of useful tool provided as part of Pyomo, and the *config* sub-library contains a number of tools that facilitate passing and validation of construction arguments to Pyomo models.

```
from pyomo.common.config import ConfigValue
```

From the IDAES core libraries, we are going to need to import *UnitBlockData*, *declare_process_block_class*, *Holdup0D* and *CONFIG_Base*.

```
from idaes.core import UnitBlockData, declare_process_block_class, Holdup0D, CONFIG_
↪Base
```

Additionally, we will also make use of the following utility functions provided by IDAES.

```
from idaes.core.util.config import is_parameter_block, list_of_strings
from idaes.core.util.misc import add_object_ref
```

Creating the Unit Model Class

The next step is to create the class which will be used to construct our new unit model. Similar to the previous tutorial, we need to declare a new class (which will form our BlockData class) and to apply the declare_process_block_class decorator to it. In this case, our class will inherit from UnitBlockData, which is the IDAES core class which forms the basis of all unit models in the core library.

```
@declare_process_block_class("CSTR")
class CSTRData(UnitBlockData):
```

Config Blocks

As we have seen in the previous tutorials, many models within IDAES (in fact all models) have optional construction arguments that the user can provide to control what type of model is constructed. This is automatically handled within IDAES by using *config blocks*, which are a Pyomo feature designed specifically for handling and organizing construction arguments for models (**Note**: despite the name, *config blocks* are not *Blocks* in the sense of the Pyomo component). For those with understanding of Python *dict* objects, *config blocks* can be thought of as specialized *dicts* with added the added features of validation and documentation of each key-value pair.

Each model class must define its own *config block*, along with the different construction arguments that it contains and their default values, domains (valid values) and documentation. To save model developers from having to write all of these themselves, IDAES provided a default *config block* which is suitable for many common unit models. In order to make use of this *config block*, all we need to do is make a local instance of the *config block* at the class level of our new class (i.e. immediately after the class declaration and not inside any method definition) as shown below:

```
CONFIG = CONFIG_Base()
```

The default IDAES *config block* defines a number of construction arguments which are common to most unit models, and further documentation on these can be found in the documentation for *Holdup blocks*. The arguments in the default *config block* are used by any associated Holdup blocks to determine which terms should be included in the material, energy and momentum balances when they are constructed.

Each of the construction arguments present in the default *config block* indicates whether a specific phenomenon is expected to occur within a unit, or what form of balance equation needs to be written for the unit in question. The

default *config block* assigns a default value to each of these, however each unit model is different, and will have different combinations of phenomena that are expected to occur. Being that we are developing a model for a CSTR, we would expect there to be some form of chemical reaction occurring within our unit, as well as the possibility of a heating or cooling jacket. Among the construction arguments in the default *config block* are the following:

- has_rate_reactions,
- has_equilibrium reactions, and,
- has_heat_transfer.

Each of these construction arguments are used to tell the unit model (and any associated Holdup blocks) whether or not to include these terms in the material and energy balances. Seeing as we expect these phenomena to be present in our unit, let's set the default value for these arguments to True.

```
CONFIG.get("has_rate_reactions")._default = True
CONFIG.get("has_equilibrium_reactions")._default = True
CONFIG.get("has_heat_transfer")._default = True
```

Property Package Arguments

The next thing we need to do is add some extra arguments to the *config block* for our new class. One important thing that is not included in the default *config block* are arguments for providing information on property blocks to the unit model. The reason for this is that some unit models may involve more than one material (e.g. heat exchangers), and thus require multiple property packages. Unit models require two arguments for each property package which they will use:

- a reference to a Property Parameter Block, and,
- a set of construction arguments to pass onto the Property Block when it is constructed (generally optional).

Our CSTR only requires one Property Package, so let's create two new entries in the *config block*. Each new entry needs to be given a unique name, and can also be given a default value, a domain and a set of documentation strings (long and/or short). The domain is a callable object of some type that can be used to validate the value provided by the user, thus ensuring a meaningful value is provided. For our example, let us add the following two arguments:

```
CONFIG.declare("property_package", ConfigValue(
    default=None,
    domain=is_parameter_block,
    description="Property package to use for holdup")
CONFIG.declare("property_package_args", ConfigValue(
    default={},
    description="Arguments to use for constructing property packages")
```

In the above code, we first declare a new entry in our *config block* and provide a name for the argument (e.g. "property_package"). We then need to specify the type of entry, which in this case is a "ConfigValue" - for more information on the different types available see the Pyomo documentation for *config blocks*. We can then provide a default value for the argument, and set the domain and documentation strings. In our example, we set the domain for "property_package" to *is_parameter_block*, which is an IDAES method that checks that the value provided by the user points to a valid Property Parameter Block.

Accessing Construction Arguments

Now that we have set up our *config block* the next question is how do we make use of it? When we create an instance of a class, the IDAES framework automatically passes any arguments provided in the object declaration to the new

objects *config block* so they are available for use in the object. So, in order to make use of the construction arguments, all we need to do is look to *object.config.argument_name* to get the current value of the construction argument.

The build Method

After setting up our *config block*, the next step is to write the *build* method for our new class, and to then call the UnitModelData *build* method (see the documentation on *Unit Model classes* for information on what is performed by this method).

```
def build(self):
    super(CSTRData, self).build()
```

Holdup Blocks

Once we have the basic framework of our unit model set up, the next thing we should do is add some Holdup blocks to represent any control volumes we have in our model. The Holdup Blocks take a set of construction arguments similar to those in *CONFIG_Base*, and write a set of material, energy and momentum balances based on these arguments (see documentation on *Holdup Blocks* for more details). For a CSTR, we have a single well-mixed control volume representing the material in the tank, so we can use a single Holdup0D to represent our system.

```
self.holdup = Holdup0D()
```

In this case, we do not need to provide any arguments to the Holdup block when we create it - if we do not provide any arguments the Holdup block will automatically look to the unit model's *config block* for its construction arguments.

Adding Performance Equations

Adding the Holdup block to our model automatically generates the material, energy and momentum balances we need, so all that is left is to write some Constraints describing how the unit performs. For a CSTR, the key performance equation is the relationship between the extent of reaction, rate of reaction and the volume of the tank:

$$X_{t,i} = V_t \times r_{t,i}$$

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.

In order to implement this Constraint, the first thing we are going to need is the volume material in the tank. One of the options for the Holdup block is *include_holdup*, which tells the Holdup block to calculate the holdup of material and energy within the control volume. This of requires the Holdup block to have a volume, so we should make use of this if it is present (i.e. if *include_holdup* is True). Otherwise, we need to create a new variable for volume. We do this as shown below:

```
if self.config.include_holdup:
    add_object_ref(self, "volume", self.holdup.volume)
else:
    # Add a volume variable, as CSTRs always need a volume
    self.volume = Var(self.time,
                      initialize=1.0,
                      doc="Reactor volume")
```

Next, we need the extent of reaction and rate of reaction terms to write our Constraint. Rate of reaction is a property of the state of the material and thus can be found in the outlet Property Block, whilst the Holdup Block automatically

generates an extent of reaction term for each reaction when the *has_rate_reaction* construction argument is True (which we set earlier). Let us create a reference to the extent of reaction in our unit model for convenience:

```
add_object_ref(self, "rate_reaction_idx", self.holdup.rate_reaction_idx)
```

Now that we have the terms we need, let's write our performance Constraint (indexed across all points in time so it can handle dynamic problems).

```
def rate_reaction_extents(b, t, k):
    return b.holdup.rate_reaction_extent[t, k] == (
                b.volume[t] *
                b.holdup.properties_out[t].reaction_rate[k])
self.rate_reaction_extents = Constraint(self.time,
                                        self.rate_reaction_idx,
                                        doc="Extents of reaction")
```

Whilst it might not seem like much, that is all we need to write to create the Constraints necessary to describe a CSTR. A lot of the work is being done automatically for us by the Holdup block (such as setting up property calculations, writing material, energy and momentum balances), so we as model developers only need to provide a set of instruction on what to build and a few important Constraints to describe the performance of our unit.

The post_transform_build Method

As has been discussed in previous tutorials, there are some parts of model construction that cannot be completed until after all model transformations have been applied. The most significant of these is the construction of the inlet and outlet *Port* objects which are used to connect unit models together. In order to facilitate this, any steps that need to be performed after transformations are applied are placed in a separate method in the model class named *post_transform_build*, which can be called by the flowsheet model.

build_inlets and build_outlets Methods

In order to simplify the construction of the inlet and outlet Port objects, the IDAES UnitBlockData class contains two methods which automate this for the user. These methods have three arguments which describe the Port object to be constructed.

- holdup - indicates which Holdup block the Port should be associated with. If not specified, the methods assume there is a Holdup block with the name *holdup*.
- an optional list of names to use if multiple Streams are to be connected to a single Port (with inherent mixing/splitting of the flows).
- an optional number of Streams that will be connected to the Port (with inherent mixing/splitting).

Only one of the list of names or number of Streams needs to be specified (and an Exception will be raised if both are provided and are not consistent). More documentation on the *build_inlets* and *build_outlets* methods can be found in the *Unit Models* documentation.

Writing Initialization Routines

To be added once the new initialization framework is completed.

Writing Model Checks

To be added once the new initialization framework is completed (as we might overhaul this at the same time).

3.2.3 IDAES Modeling Standards

Contents

- *IDAES Modeling Standards*
  - *Model Formatting and General Standards*
    - *Headers and Meta-data*
    - *Version Numbering*
    - *Licensing Information and Disclaimers*
    - *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*

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.

Version Numbering

TBD

Licensing Information and Disclaimers

TBD

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 overloading 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.

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.

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 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 Temperature	t_bub
Compressibility Factor	compress_fact
Concentration	conc
Density	dens
Dew Temperature	temperature_dew
Diffusivity	diffus
Diffusion Coefficient (binary)	diffus_binary
Enthalpy	enth
Entropy	entr
Fugacity	fug
Fugacity Coefficient	fug_coeff
Gibbs Energy	energy_gibbs
Heat Capacity (const. P)	cp
Heat Capacity (const. V)	cv
Heat Capacity Ratio	heat_capacity_ratio
Helmholtz Energy	energy_helmholtz
Henry's Constant	henry
Mass Fraction	mass_frac
Material Flow	flow
Molecular Weight	mw
Mole Fraction	mole_frac
pH	pH
Pressure	pressure

Continued on next page

Table 1 – continued from previous page

Variable	Standard Name
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

3.2.4 IDAES Modeling Framework

The IDAES Modeling Framework is divided into a number of layers in order to facilitate the modular construction of complex flowsheets. Each part of the framework has an associated base class, which serves as a foundation for constructing different types of models and to automate the tasks common to all models of that type. Understanding the different layers within the IDAES Framework, and what Variables and Constraints belong in which layer, is important for anybody wishing to use the full capabilities available.

The different parts of the IDAES modeling framework are described in the following section, along with a general discussion of the concepts behind the framework.

Modeling Classes

IDAES Modeling Concepts

Contents

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

Introduction

The purpose of this section of the documentation 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).

When other models are added to the Flowsheet, or to a descendant of the Flowsheet, that model makes a reference to the original time domain. This is handled automatically by the framework so that the user does not need to worry about it. This ensures that all models within the Flowsheet have a consistent time domain.

Different models may handle the time domain differently, but in general all IDAES models contain a component named time, which is a reference to the original time domain. 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 index 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.

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 component:

1. Unit models, representing unit operations,
2. Streams, 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 component:

1. Holdup Blocks, which represent volume of material over which we wish to perform material, energy and/or momentum balances, and,
2. PropertyBlocks, which represent the thermophysical and transport 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.

Holdup Blocks

A key feature of the IDAES modeling framework is the use of Holdup Blocks. As mentioned above, Holdup Blocks represent a volume of material over which material, energy and/or momentum balances can be performed. Holdup Blocks automate the task of writing these balance equations, writing a set of generic balance equations containing specific terms based on instructions from the modeler (see Holdup documentation for more details). Holdup Blocks

also automate the creation of Property Blocks associated with the flow of material in and out of the holdup volume so that the user does not need to do this.

Holdup Blocks maybe associated with inlets and/or outlets from their parent unit model, in which case the material, energy and momentum flows that make up these inlets and outlets connect directly to the Holdup block. The IDAES framework support having multiple inlets or outlets to a single Holdup block, in which case the flows are mixed or split within the Holdup block. In these cases, separate Mixer and Splitter sub-models are created by the Holdup block to perform the mixing and/or splitting.

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, thermophsyical, transport and reaction properties of a material (which are functions solely of the local state of the material). Additionally, Property blocks 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 Property Block to know the extensive flow information.

The calculations involved in Property Blocks 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. 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 contain in the Parameter block.

Mixers and Splitters

Mixers and Splitters are created automatically inside of Holdup blocks when a user declares multiple inlets and/or outlets to a Unit model. Mixer and Splitter blocks contain a set of material, energy and momentum balances for mixing or splitting the material flow as necessary.

Streams

Streams are used to connect different Unit models together within a Flowsheet. The contents of Streams are generated automatically when a user declares a Stream, and include:

- constraints linking the source outlet and destination inlet states,
- values for the state variables being transferred.

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. Holdup 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).

IDAES Modeling Classes

As a Python based modeling environment, Pyomo and IDAES make use of Classes to construct the various components that make up a model for a system. This section will discuss the different Classes available within the IDAES Modeling Framework and how they are intended to be used. Users need to be aware of object-oriented programming and Classes, and those who are not are encouraged to read **{add some reference}**.

Flowsheet Models

Contents

- *Flowsheet Models*
  - *Introduction*
  - *Default Property Packages*
  - *Flowsheet Configuration Arguments*
  - *Flowsheet Classes*

Introduction

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 inherits this time domain from its parent. 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_set - use 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.**FlowsheetBlock**(**args*, ***kwargs*)

FlowsheetBlock is a specialized Pyomo block for IDAES flowsheet models, and contains instances of FlowsheetBlockData.

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model}
- **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**.}

Returns New FlowsheetBlock 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

model_check()

This method runs model checks on all unit models in a flowsheet.

This method searches for objects which inherit from UnitBlockData and executes the model_check method if it exists.

Parameters **None** –

Returns None

post_transform_build()
Due to current limitations with pyomo.dae, certain tasks must be performed after the DAE transformation is applied. This method is used to automate these tasks in all components of a flowsheet.

This method searches all child objects for a post_transform_build method, and runs it if present.

Parameters **None** –

Returns None

print_active_units()
Print a list of the active units in the flowsheet.

print_all_units()
Print a list of all flowsheet units with a binary indicator.

0/1 indicator of whether it is active

Unit Models

Contents

- *Unit Models*
  - *Introduction*
  - *UnitBlock Construction Arguments*
  - *Collecting Time Domain*
  - *Modeling Support Methods*
  - *UnitBlock Classes*

Introduction

The UnitBlock is class is designed to form the basis of all IDAES UnitModels, and contains a number of methods which are common to all Unit Models.

UnitBlock Construction Arguments

The UnitBlock class by default has only one construction argument, which is listed below. However, most models inheriting from UnitBlock 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 'use_parent_value' 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 UnitBlock 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 'use_parent_value' then the Unit model looks to its parent model for the dynamic argument, otherwise the value provided at construction is used. Next, the UnitBlock collects the time domain from its parent model and sets this as the time domain for the Unit model.

Finally, if the Unit model has the 'include_holdup' construction argument, then this is checked to ensure that if dynamic = True then include_holdup is also True. If this check fails then a warning is raised and the include_holdup argument automatically set to be True.

Modeling Support Methods

The UnitBlock 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 Mmodels 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, UnitBlock contains a method named build_inlets which can automatically create an inlet (or set of inlets) to a specified Holdup block. The build_inlets method supports connecting multiple inlets to a single Holdup block, in which case an inlet Mixer Block will be created to mix the inlets prior to entering the Holdup (see documentation for Ports). The build_inlets method is described in more detail in the documentation below.

Build Outlets Method

Similar to build_inlets, UnitBlock also has a method named build_outlets for constructing outlets from Unit models. build_outlets also supports multiple outlets from a single holdup, in which case an outlet Splitter Block is created to split the outlet flow (see documentation for Ports). 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, UnitBlock 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 UnitModels 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, UnitBlock 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.

UnitBlock Classes

class idaes.core.unit_model.**UnitBlock**(**args*, ***kwargs*)

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic or not (default = 'use_parent_value'). 'use_parent_value' - get flag from parent (default = False) True - set as a dynamic model False - set as a steady-state model

Returns New UnitBlock instance

class idaes.core.unit_model.**UnitBlockData**(*component*)

This is the class for process unit operations models. These are models that would generally appear in a process flowsheet or superstructure.

build()

General build method for UnitBlockData. 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

build_inlets(*holdup=None*, *inlets=None*, *num_inlets=None*)

This is a method to build inlet Port objects in a unit model and connect these to holdup blocks as needed. This method supports an arbitary number of inlets and holdup blocks, and works for both simple (0D) and 1D IDAES holdup blocks.

Keyword Arguments

- **= holdup block to which inlets are associated. If left None,** (holdup) – assumes a default holdup (default = None).
- **= argument defining inlet names (default** (*inlets*) – None). inlets may be None or list. - None - assumes a single inlet. - list - use names provided in list for inlets (can be

  other iterables, but not a string or dict)
- **= argument indication number** (*num_inlets*) – construct (default = None). Not used if inlets arg is provided. - None - use inlets arg instead - int - Inlets will be named with sequential numbers from 1

  to num_inlets.

Returns A Pyomo Port object and assoicated components.

build_outlets(*holdup=None*, *outlets=None*, *num_outlets=None*, *split_type='flow'*)

This is a method to build outlet Port objects in a unit model and connect these to holdup blocks as needed. This method supports an arbitary number of outlets and holdup blocks, and works for both simple (0D) and 1D IDAES holdup blocks.

Keyword Arguments

- **= holdup block to which inlets are associated. If left None,** (holdup) – assumes a default holdup (default = None).
- **= argument defining outlet names (default** (*outlets*) – None). outlets may be None or list. - None - assumes a single outlet. - list - use names provided in list for outlets (can be

other iterables, but not a string or dict)

- **= argument indication number** (*num_outlets*) – construct (default = None). Not used if outlets arg is provided. - None - use outlets arg instead - int - Outlets will be named with sequential numbers from

  1 to num_outlets.

- **= argument defining method to use to split outlet flow** (*split_type*) – in case of multiple outlets (default = 'flow'). - 'flow' - outlets are split by total flow - 'phase' - outlets are split by phase - 'component' - outlets are split by component - 'total' - outlets are split by both phase and

  component

  – **'duplicate' - all outlets are duplicates of the** total outlet stream.

Returns A Pyomo Port object and assoicated components.

display_P()
Display pressure variables associated with the UnitBlockData.

display_T()
Display temperature variables associated with the UnitBlockData.

display_flows()
Display component flow variables associated with the UnitBlockData.

display_total_flows()
Display total flow variables associated with the UnitBlockData.

display_variables(*simple=True*, *descend_into=True*)
Display all variables associated with the UnitBlockData.

Parameters **simple** (*bool, optional*) – Print a simplified version showing only variable values.

initialize(*state_args=None*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)
This is a general purpose initialization routine for simple unit models. This method assumes a single Holdup block called holdup, 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 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

is_process_unit()
Tag to indicate that this object is a process unit.

model_check()
This is a general purpose initialization routine for simple unit models. This method assumes a single Holdup block called holdup and tries to call the model_check method of the holdup 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 Holdup blocks present.

Parameters **None** –

Returns None

Streams

Contents

- *Streams*
  - *Introduction*
  - *Construction Arguments*
  - *Port Validation*
  - *Stream Constraints*
  - *Stream Variables*
  - *StreamData Class*
  - *VarDict Class*

Introduction

Stream Blocks are used within the IDAES framework for connecting Unit Models at the Flowsheet level. Each Stream connects an inlet and an outlet from a Unit model.

Construction Arguments

Stream Blocks have four construction arguments, which describe the source and destination of the Stream. The construction arguments are:

- source - the outlet Port object to use as the source of the Stream.
- source_idx - (optional) index to use when source Port represents multiple outlets (i.e. comes from an OutletSplitter Block).
- destination - the inlet Port object to use as the destination of the Stream.
- destination_idx - (optional) index to use when destination Port represents multiple inlets (i.e. comes from an InletMixer Block).

Port Validation

The first step in constructing a Stream is to check that both the source and destination Ports are compatible. This is done with the following checks:

1. Ports must have the same length.
2. Ports must have the same set of named variables (keys).
3. Variables within Ports must have the same indexing sets.

If any of these tests fail, an Exception is raised stating that the Ports are not compatible and why.

Stream Constraints

The main purpose of a Stream is to connect the outlet of one unit to the inlet of another, which is done using equality Constraints. Streams automatically write a set of equality Constraints which relate each member of the source Port to the equivalent member of the destination Port.

Stream Variables

Streams also automatically generate local representations of the variables associated with the Stream, so that users can evaluate the quantities present in the Stream. These values are stored in a VarDict, and support a number of methods as shown below.

StreamData Class

Streams support a number of methods for solving models and evaluating results which are documented below.

class idaes.core.stream.**StreamData**(*component*)

This is the class for process streams. These are blocks that connect two unit models together.

activate(*var=None*)

Method for activating Constraints in Stream. If not provided with any arguments, this activates the entire Stream block. Alternatively, it may be provided with the name of a variable in the Stream, in which case only the Constraint associated with that variable will be activated.

Parameters **var** – name of a variable in the Stream for which the corresponding Constraint should be activated (default = None).

Returns None

build()

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

Parameters **None** –

Returns None

converged(*tolerance=1e-06*)

Check if the values on both sides of a Stream are converged.

Parameters **tolerance** – tolerance to use when checking if Stream is converged. (default = 1e-6).

Returns A Bool indicating whether the Stream is converged

deactivate(*var=None*)
Method for deactivating Constraints in Stream. If not provided with any arguments, this deactivates the entire Stream block. Alternatively, it may be provided with the name of a variable in the Stream, in which case only the Constraint associated with that variable will be deactivated.

Parameters **var** – name of a variable in the Stream for which the corresponding Constraint should be deactivated (default = None).

Returns None

display(*side='source'*, *display_constraints=False*, *tolerance=1e-06*, *ostream=None*, *prefix=''*)
Display the contents of Stream Block.

Parameters

- **side** – side of Stream to display values from (default = 'soruce'). Valid values are 'source' and 'destination'.
- **display_constraints** – indicates whether to display Constraint information (default = False).
- **tolerance** – tolerance to use when checking if Stream is converged. (default = 1e-6).
- **ostream** – output stream (default = None)
- **prefix** – str to append to each line of output (default = '')

VarDict Class

class idaes.core.stream.**VarDict**
This class creates an object which behaves like a Pyomo IndexedVar. It is used to contain the separate Vars contained within IndexedPorts, and make them look like a single IndexedVar. This class supports the fix, unfix and display attributes.

display(*side='source'*, *ostream=None*, *prefix=''*)
Print component information

Parameters

- **side** – which side of port to display (default = 'source'). Valid values are 'source' or 'destination'.
- **ostream** – output stream (default = None)
- **prefix** – str to append to each line of output (default = '')

Returns None

fix(*value=None*, *side='destination'*)
Method to fix Vars.

Parameters

- **value** – value to use when fixing Var (default = None).
- **side** – side of port to fix (default = 'destination'). Valid values are 'source', 'destination' or 'all'.

Returns None

unfix(*side='destination'*)
Method to unfix Vars.

Parameters

- **value** – value to use when fixing Var (default = None)
- **side** – side of port to fix (default = 'destination'). Valid values are 'source', 'destination' or 'all'.

Returns None

Holdup Classes

The IDAES process modeling framework is built around the concept of Holdup Blocks. A Holdup Block represents a single volume of material over which a set of material, energy and pressure balances can be applied.

Whilst each unit model is different, the material, energy and momentum balances all have a similar form with the main difference being the terms which appear in the balances. This allows the Holdup blocks to automatically generate the balance equations based on a set of instructions provided by the user, saving the user from the need to write the equations themselves. This section of the IDAES documentation goes through the different types of Holdup blocks available in the IDAES core framework the equations they create.

The IDAES framework contains three types of Holdup blocks for different applications; Holdup0D, Holdup1D and HoldupStatic.

- Holdup0D is the most common type of Holdup block, and represents a single well mixed volume of fluid with a single inlet flow and a single outlet flow. This type of Holdup block is uesfull for representing most simple unit operations.
- Holdup1D is useful for holdup volumes with variations in one spatial domain, such as plug flow reactors or flow in pipes.
- HoldupStatic is useful for certain special cases where the material volume has either no through flow (dead zones) or units with negligible volume (e.g. ideal mixers and splitters). More details on potential uses of HoldupStatic are given below.

Holdup blocks are not intended to be used outside of an IDAES UnitModel. Most significantly, the Holdup block expects certain components to be present in the parent model, and will raise Exceptions if these are not present.

Common Holdup Tasks

Contents

- *Common Holdup Tasks*
  - *Introduction*
  - *Construction Arguments*
  - *Setting up the time domain*
  - *Getting Property Package Information*
  - *Collecting Indexing Sets for Property Package*
  - *HoldupData Class*

Introduction

All of the IDAES Holdup block classes are built on a common core Class which automates the tasks required for all Holdup blocks. The common tasks performed by the base class are:

- Collecting construction arguments from the UnitModel,
- Determining if the Holdup should be steady-state or dynamic and getting the time domain,
- Collecting the information necessary for creating Property Blocks, and
- Collecting the component and phase lists from the Property package.

Construction Arguments

All Holdup blocks make use of a common set of construction arguments, which determine amongst other things which terms to include in the balance equations. The list of common construction argument is provided below:

Construction Argument	Default Value
property_package	None
property_package_args	{}
include_holdup	True
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

The property_package and property_package_args arguments are used to inform the Holdup block of which property package to use when constructing the associated Property Blocks, and any instructions to be provided when constructing the blocks. The balance type arguments are used to control what type of balance equation is written (more details are given in the documentation for the relevant Holdup class). The remaining arguments govern whether or not a specific term should be included in the balances equations. If a term is not included, it is replaced with a 0 in the relevant expression, and the associated variable is not constructed.

When an instance of a Holdup block is created, values for these may be provided as arguments otherwise they will be assigned a value of 'use_parent_value'. In these cases, the Holdup block looks to the UnitModel and checks to see if the UnitModel has assigned a default value for these arguments and uses this if available. If the UnitModel does not have a default value for an argument, then the Holdup reverts to the default value provided in the table above.

CONFIG_Base

To assist Unit model developers with setting up the configuration blocks for their models, the IDAES framework contains a prebuilt configuration block that can be inherited by UnitModels which contains most of the required construction arguments already, along with preset defaults and methods to validate user inputs. This prebuilt configuration block is named CONFIG_Base, and is available in holdup.py in the IDAES core.

CONFIG_Base contains the following arguments prebuilt for the developer:

- include_holdup
- material_balance_type
- energy_balance_type
- momentum_balance_type
- has_rate_reactions
- has_equilibrium_reactions
- has_phase_equilibrium
- has_mass_transfer
- has_heat_transfer
- has_work_transfer
- has_pressure_change

Information on property packages is not included in the default CONFIG_Base, as UnitModels may have multiple property packages.

Setting up the time domain

The next common task the Holdup block performs is to determine if it should be dynamic or steady-state and to collect the time domain from the UnitModel. Holdup blocks have an argument *dynamic* which can be provided during construction which specifies if the Holdup should be dynamic (dynamic = True) or steady-state (dynamic = False). If the argument is not provided, the Holdup block will inherit this argument from the UnitModel.

After setting the dynamic argument, the Holdup block then gets a reference to the time domain from the UnitModel. If the block containing the Holdup block does not have an attribute named time, and error will be returned.

Getting Property Package Information

If a reference to a property package was not provided by the UnitModel as an argument, the Holdup block first checks to see if the UnitModel has a property_package argument set, and uses this if present. Otherwise, the Holdup 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 not default_property_package is found, an Exception is returned.

Collecting Indexing Sets for Property Package

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

The indexing sets the Holdup block 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
- rate_reaction_idx - a list of rate controlled reactions present in the system. Used if has_rate_reactions = True to determine how many generation terms are required. If rate_reaction_idx is found, the Holdup also looks for a list of stoichiometric coefficients (rate_reaction_stoichiometry). If either of these is not found, a warning is raised and has_rate_reaction is set to False.

- equilibrium_reaction_idx - a list of equilibrium reactions present in the system. Used if has_equilibrium_reactions = True to determine how many generation terms are required. If equilibrium_reaction_idx is found, the Holdup also looks for a list of stoichiometric coefficients (equilibrium_reaction_stoichiometry). If either of these is not found, a warning is raised and has_equilibrium_reaction is set to False.
- phase_equilibrium_idx - a list of phase equilibrium reactions present in the system. Used if has_phase_equilibrium = True to determine how many generation terms are required. If phase_equilibrium_idx is found, the Holdup also looks for a list of species which are in phase equilibrium (phase_equilibrium_list). If either of these is not found, a warning is raised and has_phase_equilibrium is set to False.
- If material_balance_type is set to use element balances, the Holdup block tries to find a component named element_list (which contains a list of elements present in the species of the system). IF this is not found, an Exception is returned.

HoldupData Class

class idaes.core.holdup.**HoldupData**(*component*)

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

The most signfiicant role of the Holdup class is to set up the build arguments for the holdup block, automatically link to the time domain of the parent block, and to get the information about the property package.

build()

General build method for Holdup blocks. This method calls a number of sub-methods which automate the construction of expected attributes of all Holdup blocks.

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

Parameters **None** –

Returns None

get_property_package()

This method gathers the necessary information about the property package to be used in the holdup block.

If a property package has not been provided by the user, the method searches up the model tree until it finds an object with the 'default_property_package' attribute and uses this package for the holdup block.

The method also gathers any default construction arguments specified for the property package and combines these with any arguments specified by the user for the holdup block (user specified arguments take priority over defaults).

Parameters **None** –

Returns None

Holdup0D

Contents

- *Holdup0D*
  - *Introduction*

- *Holdup0D Equations*
- *Holdup0D Variables*
- *Initialization*
- *Holdup0dData Class*

Introduction

The Holdup0D block is the most commonly used of all the Holdup block classes, 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. Holdup0D blocks contain two Property blocks - one for the incoming material and one for the material within and leaving the volume - and write a set of material, energy and momentum balance equations with distinct inlet and outlet flows.

Holdup0D Equations

Holdup0D contains support for a number of different forms of the material, energy and momentum balances, as well as options for controlling which terms will appear in these equations. The different options available are outlined below along with the equations written with each choice. In all cases, the extensive flow terms are provided by the associated Property Blocks.

Material Balance Types

Holdup0D provides support for three different types of material balance.

- component_phase - material balances are written for each component in each phase (e.g. separate balances for liquid water and steam). 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 include_holdup is True holdup terms will still appear for these species, however these will be set to 0). The equations written by the Holdup block for phase-component balances have the form:

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

where $M_{t,p,j}$ is the holdup of component j in phase p within the control volume and time t, $F_{in,t,p,j}$ and $F_{out,t,p,j}$ are the flow of material into and out of the control volume respectively, $N_{kinetic,t,p,j}$, $N_{equilibrium,t,p,j}$ and $N_{pe,t,p,j}$ are the generation of species j in phase p by kinetic, chemical equilibrium and phase equilibrium controlled reactions respectively, and $N_{transfer,t,p,j}$ is a term to allow for other forms of mass transfer within or across the system boundary.

- component_total - material balances will be written for each component across all phases (e.g. one balance for both liquid water and steam). Phase equilibrium terms are not included in this form of the material balance. This form can be useful for steady-state systems with phase-equilibrium. However users should be careful using this form of the material balance, especially for dynamic systems, as there are often additional degrees of freedom that need to be specified. The equations written by the Holdup block for total component balances have the form:

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

- element_total - material balances are written for each element in the system (e.g. one material balance for hydrogen and one for oxygen). Only flow and mass transfer terms are included in this form of the material balance. This form of the material balance can be useful for certain reactive systems, and is necessary for

performing Gibbs energy minimization. The equations written by the Holdup block for total element balances have the form:

$$\sum_p \frac{\partial M_{t,p,e}}{\partial t} = \sum_p F_{in,t,p,e} - \sum_p F_{out,t,p,e} + \sum_p N_{transfer,t,p,e}$$

where $M_{t,p,e}$ is the holdup of element e in phase p within the control volume at time t, $F_{in,t,p,e}$ and $F_{out,t,p,e}$ are the flow of element e in phase p into and out of the control volume respectively, and $N_{transfer,t,p,e}$ is a term to allow for other forms of mass transfer within or across the system boundary.

- none - no material balances are written.

Energy Balance Types

Holdup0D currently supports only one form of energy balance.

- total - one energy balance is written for the entire holdup, summing contributions from all phases. The form of the total energy balance written by the holdup block is:

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

where E_p is the holdup of energy in phase p at time t, $H_{in,t,p}$ and $H_{out,t,p}$ are the flow of energy into and out of the control volume, Q_t is the heat transferred into the system, W_t is the work transfer into the system and s is a scaling factor.

- none - no energy balances are written.

Momentum Balance Types

Holdup0D currently supports only one form of momentum balance.

- total - one momentum balance is written for the entire holdup. Currently this is a simple pressure balance across the system with a potential pressure drop term. The form of the total momentum balance written by the holdup block is:

$$0 = s \times P_{in,t} - s \times P_{out,t} + s \times \Delta P_t$$

where $P_{in,t}$ and $P_{out,t}$ are the pressure into and out of the control volume at time t, ΔP_t is the pressure drop between the inlet and outlet, and s is a scaling factor.

- none - no momentum balances are written.

Supporting Equations

Holdup0D also creates a number of supporting Variables and Constraints as required by the balance equations, which are summarized below.

Phase Fraction

For systems with more than one phase present (determined automatically from the phase list provided by the property package), Holdup0D creates a phase fraction variable for each phase, and enforces the following constraint at all points in time t:

$$\sum_p \phi_{t,p} = 1$$

When only one phase is present, ϕ is automatically substituted with 1 in all equations.

Holdup Calculations

Holdup0D also calculates the holdup terms for the material and energy balances (unless include_holdup is False), and automatically writes Constraints for these based on information provided by the property package. The form of the holdup constraint for component balances is:

$$M_{t,p,j} = V_t \times \phi_{t,p} \times \rho_{t,p,j}$$

where $M_{t,p,j}$ is the holdup of species j in phase p at time t, V_t is the volume of the control volume at time t (this supports control volumes of varying volume), $\phi_{t,p}$ is the phase fraction of phase p at time t and $\rho_{t,p,j}$ is the material density of component j in phase p at time t (provided by the outlet Property Block). For phase-component pairs which do not exist (as indicated by the Property Block), the following Constraint is written instead (this is required to close the degrees of freedom):

$$M_{t,p,j} = 0$$

For holdup blocks using element balances, an elemental holdup is required instead, which is calculated as follows:

$$M_{t,p,e} = V_t \times \phi_{t,p} \times \rho_{t,p,j} \times e_{j,e}$$

where $e_{j,e}$ is the number of moles of element e per mole of component j.

For the holdup term in the energy balances, the following Constraints are written:

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

where $E_{t,p}$ is the energy holdup in phase p at time t and $\rho e_{t,p}$ is the volumetric energy density in phase p (provided by the Property Block).

Stoichiometric Constraints

Holdup0D also automatically generates stoichiometric Constraints to ensure conservation of mass (and elements) between products and reactants in all types of reactions (including phase equilibrium). For kinetic and chemical equilibrium reactions, an extent of reaction terms is generated for each reaction, and Constraint is written to relate these to the generation terms in the balance equations. The form of these equations is:

$$N_{t,p,j} = \sum_r \nu_{r,p,j} \times X_{t,r}$$

where $N_{t,p,j}$ is the generation of species j in phase p at time t, $\nu_{r,p,j}$ is the stoichiometric coefficient for the generation of species j in phase p by reaction r and $X_{t,r}$ is the extent of reaction r at time t.

For phase equilibrium reactions, the stoichiometric constraints can be simplified as it is known that the equilibrium is 1:1 between the same species in two different phases. In this case, rather than write separate generation terms for each phase-component pair, a generation term is written for each phase equilibrium reaction that occurs, and the term substituted directly into the material balance (with a ± 1 factor to determine which phase is 'product' phase and which the 'reactant' phase).

Element Flow Terms

Element balances require an additional set of equations for relating component based flows to their element equivalents. When element balances are chosen for a holdup block, the following additional Constraints are written:

$$F_{t,p,e} = F_{t,p,j} \times e_{j,e}$$

where $e_{j,e}$ is the number of moles of element e per mole of component j.

Construction Options

Options available in Holdup0D for specifying which terms should appear within the balance equations. Most terms in the balance equations have an associated construction argument, which are described below:

- dynamic - controls whether accumulation terms will be included in the balance equations, and the necessary constraints for relating these to the holdup terms. If set to True, accumulation terms will be constructed and included in the model. The number of accumulation terms created depends on the number of time points, phases and components present in the model.
- include_holdup - controls whether terms and constraints for the holdup of material and energy should be constructed (momentum holdup is not yet supported). If set to True, material and energy holdup terms are created, along with constraints linking these to the volume of the holdup and the material and energy densities. If dynamic = True, include_holdup must also be True (this is automatically checked by the framework, and a warning raised if set incorrectly). If dynamic = False, then include_holdup may be set to True or False as desired by the user, depending on whether the holdup terms are required in the flowsheet.
- has_rate_reactions - controls whether generation terms should be constructed for rate controlled reactions. If this is True, generation terms are created for each phase and component pair and included in the material balance equations. Additionally, an extent of reaction term is created for each reaction identified in the associated property package, and linked to the generation term via stoichiometric constraints. The Unit model or user is then expected to provide a set of Constraints relating the extent of reaction terms to the performance of the given unit operation.
- has_equilibrium_reactions - controls whether generation terms should be constructed for equilibrium controlled reactions (excluding phase equilibrium). If this is True, generation terms are created for each phase-component pair, and included in the material balance equations. Additionally, stoichiometric constraints are written relating the different generation term. The property package is expected to provide a set of equilibrium constraints which enforce the equilibrium conditions.
- has_phase_equilibrium - controls whether generation terms should be constructed for phase equilibrium reactions. If this is True, generation terms are created for each phase-component pair, and included in the material balance equations. The property package is expected to provide a set of equilibrium constraints which enforce the equilibrium conditions.
- has_mass_transfer - controls whether mass transfer terms should be constructed in the material balance equations. If True, the generic mass transfer terms will be included, and the UnitModel or user will need to provide constraints for these terms.
- has_heat_transfer - controls whether the heat transfer term should be included in the energy balance equation. If True, the Q term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_work_transfer - controls whether the work transfer term should be included in the energy balance equation. If True, the W term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_pressure_change - controls whether the pressure change term should be included in the momentum balance equation. If True, the ΔP term is constructed and the UnitModel or user will need to provide a constraint for this term.

Holdup0D Variables

The following is a table of all variables that may be constructed by a Holdup0D block (depending on options chosen), along with the names used to identify these quantities within actual code (indices are shown in the same order they appear in the code).

Variable	Name
V_t	volume
$M_{t,p,j}$	material_holdup
$\frac{\partial M_{t,p,j}}{\partial t}$	material_accumulation
$M_{t,p,e}$	element_holdup
$\frac{\partial M_{t,p,e}}{\partial t}$	element_accumulation
$E_{t,p}$	energy_holdup
$\frac{\partial E_{t,p}}{\partial t}$	energy_accumulation
$\phi_{t,p}$	phase_fraction
$N_{kinetic,t,p,j}$	rate_reaction_generation
$N_{equilibrium,t,p,j}$	equilibrium_reaction_generation
$N_{pe,t,r}$	phase_equilibrium_generation
$N_{transfer,t,p,j}$	mass_transfer_term
$N_{transfer,t,p,e}$	elemental_mass_transfer
Q_t	heat
W_t	work
ΔP_t	deltaP
$X_{kinetic,t,r}$	rate_reaction_extent
$X_{equilibrium,t,r}$	equilibrium_reaction_extent

Additionally, two scaling factors are generated for the energy and momentum balances, named scaling_factor_energy and scaling_factor_momentum respectively.

Initialization

Holdup0D has an initialization method which can be called as part of initializing an associated UnitModel. The initialization method takes a set of state arguments which is passed on to the associated property package, mixers and splitters. As initialization of UnitModels often require the inlet state of the unit to be held at a fixed state whilst initializing the unit, the holdup initialization routine can be instructed to hold the inlet state at a fixed state until instructed to unfix it (an associated release_state method exists for this purpose). The procedure followed by the initialization routine is as follows:

1. If an inlet mixer is present, the initialization routine of the mixer is called.
2. The inlet Property Block is initialized (properties_in).
3. The outlet Property Block is initialized (properties_out).
4. If an outlet splitter is present, the initialization routine of the splitter is called.
5. If not instructed to hold the inlet state fixed, the release_state method is called, otherwise a dictionary of information on what state variables were fixed is returned to be used when calling release_state.

The associated release_state method takes the dictionary of flags returned above, and uses this to unfix any variables fixed during initialization.

Holdup0dData Class

class idaes.core.holdup.**Holdup0dData**(*component*)
0-Dimensional (Non-Discretised) Holdup Class

This class forms the core of all non-discretized IDAES models. It builds property blocks and adds mass, energy and momentum balances. The form of the terms used in these constraints is specified in the chosen property package.

build()
Build method for Holdup0D blocks. This method calls submethods to setup the necessary property blocks, distributed variables, material, energy and momentum balances based on the arguments provided by the user.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)
Initialisation routine for holdup (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 initialisation routine. **Valid values: 0** - no output (default), **1** - return solver state for each step in routine, **2** - include solver output infomation (tee=True)
- **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 varaibles 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 relase_state method.

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()
This method exectues the model_check methods on the associated property blocks (if they exist). This method is generally called by a unit model as part of the unit's model_check method.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging

Returns None

Holdup1D

Contents

- *Holdup1D*
  - *Introduction*

- *Spatial Domain*
- *Holdup1D Equations*
- *Holdup1D Variables*
- *Initialization*
- *Holdup1dData Class*

Introduction

The Holdup1D block is used to model systems with variations in one spatial dimension, such as plug flow reactors and idea pipes. Holdup1D blocks contain a set of Indexed Property Blocks (with an instance of a Property Block at each node in the spatial domain), and write a set of material, energy and momentum balance equations with differential flow terms.

Spatial Domain

Holdup1D blocks have a normalized domain, name ldomain, which represents the spatial dimension of the unit. The developer has the option of specifying a domain to inherit from a parent model, or to construct a new domain for each holdup. This domain is normally defined as a Pyomo ContinuousSet with bounds of 0 and 1, and must be transformed using Pyomo.dae before the model can be solved. This is done automatically during construction of Holdup1D, based on the following arguments provided by the developer.

- inherited_length_domain - allows the developer to provide an existing length domain to use within the holdup. This will be used wherever ldomain is called for, and no new domain will be created. Note: the balance equations in Holdup1D are written in normalized form, and thus expect a normalized domain. Unexpected behavior may occur if a non-normalized domain is provided.
- length_domain_set - allows the user to specify a set of points to use as part of the spatial domain (default = [0.0, 1.0]). These points will be used to populate the initial domain, with additional points added later during transformation as required. This allows the user to specify custom grid spacings as required. Note: the balance equations in Holdup1D are written in normalized form, and thus expect a normalized domain. Unexpected behavior may occur if a non-normalized domain is provided.
- flow_direction - indicates the direction of flow within the length domain. Options are:
  - 'forward' - flow from 0 to 1 (default)
  - 'backward' - flow from 1 to 0
- discretization_method - specifies the method to use when discretizing the spatial domain. Options are:
  - BFD - backwards finite difference method (default)
  - FFD - forwards finite difference method
  - OCLR - orthogonal collocation on finite elements (Lagrange-Radau roots)
  - OCLL - orthogonal collocation on finite elements (Lagrange-Legendre roots)
- finite_elements - number of finite elements to use when discretizing domain (default = 20)
- collocation_ponts - number of collocation points to use per finite element (default = 3, collocation methods only)

Holdup1D Equations

Holdup1D contains support for a number of different forms of the material, energy and momentum balances, as well as options for controlling which terms will appear in these equations. The different options available are outlined below along with the equations written with each choice. In all cases, the extensive flow terms are provided by the associated Property Blocks, and the inlet boundary condition is provided by the inlet stream (inlet may be at $x = 0$ or $x = 1$ depending on flow direction).

Material Balance Types

Holdup1D provides support for three different types of material balance.

- component_phase - material balances are written for each component in each phase (e.g. separate balances for liquid water and steam). 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 include_holdup is True holdup terms will still appear for these species, however these will be set to 0). The equations written by the Holdup block for phase-component balances have the form:

$$@x \neq inlet, L \times \frac{\partial M_{t,x,p,j}}{\partial t} = fd \times \frac{\partial F_{t,x,p,j}}{\partial x} + L \times N_{kinetic,t,x,p,j} + L \times N_{equilibrium,t,x,p,j}$$
$$+L \times N_{pe,t,x,p,j} + L \times N_{transfer,t,x,p,j} + \frac{A}{L} \times J_{diffusion,t,x,p,j}$$

where $M_{t,x,p,j}$ is the holdup of component j in phase p at point :math:'x' and time t, L and A are the (total) length and area of the control volume respectively, $F_{t,x,p,j}$ is the flow of species j in phase p at point x and time t, $N_{kinetic,t,x,p,j}$, $N_{equilibrium,t,x,p,j}$ and $N_{pe,t,x,p,j}$ are the generation of species j in phase p by kinetic, chemical equilibrium and phase equilibrium controlled reactions respectively, $N_{transfer,t,x,p,j}$ is a term to allow for other forms of mass transfer within or across the system boundary and $J_{diffusion,t,x,p,j}$ is the diffusive material flux of component j in phase p at point x and time t.

- component_total - material balances will be written for each component across all phases (e.g. one balance for both liquid water and steam). Phase equilibrium terms are not included in this form of the material balance. This form can be useful for steady-state systems with phase-equilibrium. However users should be careful using this form of the material balance, especially for dynamic systems, as there are often additional degrees of freedom that need to be specified. The equations written by the Holdup block for total component balances have the form:

$$@x \neq inlet, L \times \sum_p \frac{\partial M_{t,x,p,j}}{\partial t} = fd \times \sum_p \frac{\partial F_{t,x,p,j}}{\partial x} + L \times \sum_p N_{kinetic,t,p,j}$$
$$+L \times \sum_p N_{equilibrium,t,p,j} + L \times \sum_p N_{transfer,t,p,j} + \frac{A}{L} \times J_{diffusion,t,x,p,j}$$

- element_total - material balances are written for each element in the system (e.g. one material balance for hydrogen and one for oxygen). Only flow and mass transfer terms are included in this form of the material balance. This form of the material balance can be useful for certain reactive systems, and is necessary for performing Gibbs energy minimization. The equations written by the Holdup block for total element balances have the form:

$$@x \neq inlet, L \times \sum_p \frac{\partial M_{t,x,p,e}}{\partial t} = fd \times \sum_p \frac{\partial F_{t,x,p,e}}{\partial x} + L \times \sum_p N_{transfer,t,x,p,e} + \frac{A}{L} \times J_{diffusion,t,x,p,e}$$

where $M_{t,x,p,e}$ is the holdup of element e in phase p at point x and time t, $F_{t,x,p,e}$ is the flow of element e in phase p at point x and time t, $N_{transfer,t,x,p,e}$ is a term to allow for other forms of mass transfer, and $J_{diffusion,t,x,p,e}$ is the diffusive material flux of element e in phase p at point x and time t.

- none - no material balances are written.

Energy Balance Types

Holdup1D currently supports only one form of energy balance.

- total - one energy balance is written for the entire holdup, summing contributions from all phases. The form of the total energy balance written by the holdup block is:

$$@x \neq inlet, s \times L \times \sum_p \frac{\partial E_{t,x,p}}{\partial t} = s \times fd \times \sum_p \frac{\partial H_{t,x,p}}{\partial x} + s \times L \times Q_{t,x} + s \times L \times W_{t,x} + s \times \frac{A}{L} \times J_{conduction,t,x,p}$$

where E_p is the holdup of energy in phase p at point x and time t, $H_{t,x,p}$ is the flow of energy in phase p at point x and time t, $Q_{t,x}$ is the heat transferred into the system, $W_{t,x}$ is the work transfer into the system, $J_{conduction,t,x,p}$ is the conductive heat transfer term in phase p at point x and time t, and s is a scaling factor.

- none - no energy balances are written.

Momentum Balance Types

Holdup1D currently supports only one form of momentum balance.

- total - one momentum balance is written for the entire holdup. Currently this is a simple pressure balance across the system with a potential pressure drop term. The form of the total momentum balance written by the holdup block is:

$$@x \neq inlet, 0 = s \times fd \times \frac{\partial P_{t,x}}{\partial x} + s \times L \times \Delta P_{t,x}$$

where $P_{t,x}$ is the pressure at point x and time t, $\Delta P_{t,x}$ is the pressure drop at point x and time t, and s is a scaling factor.

- none - no momentum balances are written.

Supporting Equations

Holdup1D also creates a number of supporting Variables and Constraints as required by the balance equations, which are summarized below.

Geometry Constraints

Holdup1D writes one Constraint relating the volume and length of the control volume.

$$V = L \times A$$

where V is the volume of the control volume, L is the (actual) length of the spatial domain and A is the cross-sectional area of the control volume. Holdup1D does not currently support control volumes of changing volume.

Phase Fraction

For systems with more than one phase present (determined automatically from the phase list provided by the property package), Holdup1D creates a phase fraction variable for each phase, and enforces the following constraint at all points in space x and time t:

$$\sum_p \phi_{t,x,p} = 1$$

When only one phase is present, ϕ is automatically substituted with 1 in all equations.

Holdup Calculations

Holdup1D also calculates the holdup terms for the material and energy balances (unless include_holdup is False), and automatically writes Constraints for these based on information provided by the property package. The form of the holdup constraint for component balances is:

$$M_{t,x,p,j} = A \times \phi_{t,x,p} \times \rho_{t,x,p,j}$$

where $M_{t,x,p,j}$ is the holdup of species j in phase p at point x and time t, a is the cross-sectional area of the control volume, $\phi_{t,x,p}$ is the phase fraction of phase p at point x and time t and $\rho_{t,x,p,j}$ is the material density of component j in phase p at point x and time t (provided by the Property Block). For phase-component pairs which do not exist (as indicated by the Property Block), the following Constraint is written instead (this is required to close the degrees of freedom):

$$M_{t,x,p,j} = 0$$

For holdup blocks using element balances, an elemental holdup is required instead, which is calculated as follows:

$$M_{t,x,p,e} = a \times \phi_{t,x,p} \times \rho_{t,x,p,j} \times e_{j,e}$$

where $e_{j,e}$ is the number of moles of element e per mole of component j.

For the holdup term in the energy balances, the following Constraints are written:

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

where $E_{t,x,p}$ is the energy holdup in phase p at point x and time t and $\rho e_{t,x,p}$ is the volumetric energy density in phase p (provided by the Property Block).

Extensive Flow Terms

Due to the spatial domain, Holdup1D writes the extensive flow terms in the balances equations as partial derivatives with respect to the spatial domain. Due to the way Pyomo.dae works, this also requires the related flow terms to be indexed variables (indexed by the spatial domain). To handle this, Holdup1D creates equivalent variables for these, and equates these to equivalent terms in the Property Blocks.

Stoichiometric Constraints

Holdup1D also automatically generates stoichiometric Constraints to ensure conservation of mass (and elements) between products and reactants in all types of reactions (including phase equilibrium). For kinetic and chemical equilibrium reactions, an extent of reaction terms is generated for each reaction, and Constraint is written to relate these to the generation terms in the balance equations. The form of these equations is:

$$N_{t,x,p,j} = \sum_{r} \nu_{r,p,j} \times X_{t,x,r}$$

where $N_{t,x,p,j}$ is the generation of species j in phase p at point x and time t, $\nu_{r,p,j}$ is the stoichiometric coefficient for the generation of species j in phase p by reaction r and $X_{t,x,r}$ is the extent of reaction r at point x and time t.

For phase equilibrium reactions, the stoichiometric constraints can be simplified as it is known that the equilibrium is 1:1 between the same species in two different phases. In this case, rather than write separate generation terms for each phase-component pair, a generation term is written for each phase equilibrium reaction that occurs, and the term substituted directly into the material balance (with a ± 1 factor to determine which phase is 'product' phase and which the 'reactant' phase).

Diffusion and Conduction Constraints

For systems with diffusive mass transfer or conductive heat transfer, Holdup1D automatically writes constraints relating the relevant flux terms to the associated properties from the Property Block. The diffusive mass transfer term is:

$$x \neq inlet, J_{diffusion,t,x,p,j} = -D_{t,x,p,j} \frac{\partial^2 C_{t,x,p,j}}{\partial x^2}$$

where $D_{t,x,p,j}$ is the diffusion coefficient and $C_{t,x,p,j}$ is the concentration of species j in phase p at point x and time t (both provided by the Property Block).

The conductive heat transfer term is:

$$x \neq inlet, J_{conduction,t,x,p} = -k_{t,x,p} \frac{\partial^2 T_{t,x}}{\partial x^2}$$

where $k_{t,x,p}$ is the thermal conductivity of phase p at point x and time t and $T_{t,x}$ is the temperature of the material at point x and time t (the IDAES framework currently does not support different temperatures in different phases of the same Property Block).

Element Flow Terms

Element balances require an additional set of equations for relating component based flows to their element equivalents. When element balances are chosen for a holdup block, the following additional Constraints are written:

$$F_{t,x,p,e} = F_{t,x,p,j} \times e_{j,e}$$

where $e_{j,e}$ is the number of moles of element e per mole of component j. If mass diffusion is also included, the following Constraint is also written to calculate the elemental diffusive flux term:

$$J_[elemental, t, x, e] == \sum_j (J_{diffusion,t,x,p,j} \times e_{j,e}$$

Construction Options

Options available in Holdup1D for specifying which terms should appear within the balance equations. Most terms in the balance equations have an associated construction argument, which are described below:

- dynamic - controls whether accumulation terms will be included in the balance equations, and the necessary constraints for relating these to the holdup terms. If set to True, accumulation terms will be constructed and included in the model. The number of accumulation terms created depends on the number of time points, phases and components present in the model.
- include_holdup - controls whether terms and constraints for the holdup of material and energy should be constructed (momentum holdup is not yet supported). If set to True, material and energy holdup terms are created, along with constraints linking these to the volume of the holdup and the material and energy densities. If dynamic = True, include_holdup must also be True (this is automatically checked by the framework, and a warning raised if set incorrectly). If dynamic = False, then include_holdup may be set to True or False as desired by the user, depending on whether the holdup terms are required in the flowsheet.
- has_rate_reactions - controls whether generation terms should be constructed for rate controlled reactions. If this is True, generation terms are created for each phase and component pair and included in the material balance equations. Additionally, an extent of reaction term is created for each reaction identified in the associated property package, and linked to the generation term via stoichiometric constraints. The Unit model or user is then expected to provide a set of Constraints relating the extent of reaction terms to the performance of the given unit operation.

- has_equilibrium_reactions - controls whether generation terms should be constructed for equilibrium controlled reactions (excluding phase equilibrium). If this is True, generation terms are created for each phase-component pair, and included in the material balance equations. Additionally, stoichiometric constraints are written relating the different generation term. The property package is expected to provide a set of equilibrium constraints which enforce the equilibrium conditions.
- has_phase_equilibrium - controls whether generation terms should be constructed for phase equilibrium reactions. If this is True, generation terms are created for each phase-component pair, and included in the material balance equations. The property package is expected to provide a set of equilibrium constraints which enforce the equilibrium conditions.
- has_mass_transfer - controls whether mass transfer terms should be constructed in the material balance equations. If True, the generic mass transfer terms will be included, and the UnitModel or user will need to provide constraints for these terms.
- has_heat_transfer - controls whether the heat transfer term should be included in the energy balance equation. If True, the Q term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_work_transfer - controls whether the work transfer term should be included in the energy balance equation. If True, the W term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_pressure_change - controls whether the pressure change term should be included in the momentum balance equation. If True, the ΔP term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_mass_diffusion - controls whether the mass diffusion terms should be included in the material balance equations. If True, the diffusion terms and associated Constraints are constructed.
- has_energy_diffusion - controls whether the energy conduction terms should be included in the energy balance equations. If True, the conduction terms and associated Constraints are constructed.

Holdup1D Variables

The following is a table of all variables that may be constructed by a Holdup1D block (depending on options chosen), along with the names used to identify these quantities within actual code (indices are shown in the same order they appear in the code).

Variable	Name
V	volume
L	length
A	area
$F_{t,x,p,j}$	material_flow
$\frac{\partial F_{t,x,p,j}}{\partial x}$	material_flow_dx
$F_{t,x,p,e}$	element_flow
$\frac{\partial F_{t,x,p,e}}{\partial x}$	element_flow_dx
$H_{t,x,p}$	energy_flow
$\frac{\partial h_{t,x,p}}{\partial x}$	energy_flow_dx
$P_{t,x}$	pressure
$\frac{\partial P_{t,x}}{\partial x}$	pressure_dx
$M_{t,x,p,j}$	material_holdup
$\frac{\partial M_{t,x,p,j}}{\partial t}$	material_accumulation
$M_{t,x,p,e}$	element_holdup
$\frac{\partial M_{t,x,p,e}}{\partial t}$	element_accumulation
$E_{t,x,p}$	energy_holdup

Continued on next page

Table 2 – continued from previous page

Variable	Name
$\frac{\partial E_{t,x,p}}{\partial t}$	energy_accumulation
$\phi_{t,x,p}$	phase_fraction
$N_{kinetic,t,x,p,j}$	rate_reaction_generation
$N_{equilibrium,t,x,p,j}$	equilibrium_reaction_generation
$N_{pe,t,x,r}$	phase_equilibrium_generation
$N_{transfer,t,x,p,j}$	mass_transfer_term
$N_{transfer,t,x,p,e}$	elemental_mass_transfer
$Q_{t,x}$	heat
$W_{t,x}$	work
$\Delta P_{t,x}$	deltaP
$X_{kinetic,t,x,r}$	rate_reaction_extent
$X_{equilibrium,t,x,r}$	equilibrium_reaction_extent
$J_{diffusion,t,x,p,j}$	material_diffusive_flux
$J_[t,x,e]$	elemental_diffusive_flux
$J_{conduction,t,x,p}$	energy_conduction_term
$\frac{\partial^2 C_{t,x,p,j}}{\partial x^2}$	material_concentration_dx2
$\frac{\partial^2 T_{t,x}}{\partial x^2}$	temperature_dx2

Additionally, to scaling factors are generated for the energy and momentum balances, named scaling_factor_energy and scaling_factor_momentum respectively.

Initialization

Holdup1D has an initialization method which can be called as part of initializing an associated UnitModel. The initialization method takes a set of state arguments which is passed on to the associated property package, mixers and splitters. As initialization of UnitModels often require the inlet state of the unit to be held at a fixed state whilst initializing the unit, the holdup initialization routine can be instructed to hold the inlet state at a fixed state until instructed to unfix it (an associated release_state method exists for this purpose). The procedure followed by the initialization routine is as follows:

1. If an inlet mixer is present, the initialization routine of the mixer is called.
2. The Property Block is initialized (properties).
3. If an outlet splitter is present, the initialization routine of the splitter is called.
4. If not instructed to hold the inlet state fixed, the release_state method is called, otherwise a dictionary of information on what state variables were fixed is returned to be used when calling release_state.

The assoiciated release_state method takes the dictionary of flags returned above, and uses this to unfix any variables fixed during initialization.

Holdup1dData Class

class idaes.core.holdup.**Holdup1dData**(*component*)

1-Dimensional Holdup Class

This class is designed to be the core of all 1D discretized IDAES models. It builds property blocks, inlet/outlet ports and adds mass, energy and momentum balances. The form of the terms used in these constraints is specified in the chosen property package.

Assumes constant reactor dimensions

build()
Build method for Holdup1D blocks. This method calls submethods to setup the necessary property blocks, distributed variables, material, energy and momentum balances based on the arguments provided by the user.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *hold_state=True*, *solver='ipopt'*, *optarg=None*)
Initialisation routine for holdup (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 initialisation routine
  - 0 = no output
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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). - True - states varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()
This method exectues the model_check methods on the associated property blocks (if they exist). This method is generally called by a unit model as part of the unit's model_check method.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging

Returns None

HoldupStatic

Contents

- *HoldupStatic*
  - *Introduction*
  - *HoldupStatic Equations*
  - *HoldupStatic Variables*
  - *Initialization*
  - *HoldupStaticData Class*

Introduction

The HoldupStatic block is used for specific cases of holdups where only a single Property Block is required for the control volume. This is primarily intended for dead zones with no through-flow of material or energy (only heat and mass transfer terms). HoldupStatic blocks are distinguished by only having a single Property Block associated with them (Holdup0D on the other hand has separate inlet and outlet property blocks), and the balance equations written by HoldupStatic do not include flow terms.

Using HoldupStatic for Ideal Mixers and Splitters

HoldupStatic can be used in special cases for units with through-flow where there are multiple inlets and/or outlets - such as in the core IDAES Mixer and Splitter models. In these cases, an mixer and/or splitter unit is also part of the Holdup Block (due to the multiple inlets or outlets) which already contain Property Blocks for the separate inlet/outlet streams as well as material, energy and momentum balances for mixing/splitting the material. Rather than create unnecessary duplicates of these, HoldupStatic is used with no balance equations to create a container for the mixer/splitter and a Property Block for the mixed material which is sufficient for modeling an ideal mixer or splitter unit.

HoldupStatic Equations

HoldupStatic contains support for a number of different forms of the material, energy and momentum balances, as well as options for controlling which terms will appear in these equations. The different options available are outlined below along with the equations written with each choice. In all cases, the extensive flow terms are provided by the associated Property Block.

Material Balance Types

HoldupStatic provides support for three different types of material balance.

- component_phase - material balances are written for each component in each phase (e.g. separate balances for liquid water and steam). 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 include_holdup is True holdup terms will still appear for these species, however these will be set to 0). The equations written by the Holdup block for phase-component balances have the form:

$$\frac{\partial M_{t,p,j}}{\partial t} = N_{kinetic,t,p,j} + N_{equilibrium,t,p,j} + N_{pe,t,p,j} + N_{transfer,t,p,j}$$

where $M_{t,p,j}$ is the holdup of component j in phase p within the control volume and time t, $N_{kinetic,t,p,j}$, $N_{equilibrium,t,p,j}$ and $N_{pe,t,p,j}$ are the generation of species j in phase p by kinetic, chemical equilibrium and phase equilibrium controlled reactions respectively, and $N_{transfer,t,p,j}$ is a term to allow for other forms of mass transfer within or across the system boundary.

- component_total - material balances will be written for each component across all phases (e.g. one balance for both liquid water and steam). Phase equilibrium terms are not included in this form of the material balance. This form can be useful for steady-state systems with phase-equilibrium. However users should be careful using this form of the material balance, especially for dynamic systems, as there are often additional degrees of freedom that need to be specified. The equations written by the Holdup block for total component balances have the form:

$$\sum_p \frac{\partial M_{t,p,j}}{\partial t} = \sum_p N_{kinetic,t,p,j} + \sum_p N_{equilibrium,t,p,j} + \sum_p N_{transfer,t,p,j}$$

- element_total - material balances are written for each element in the system (e.g. one material balance for hydrogen and one for oxygen). Only flow and mass transfer terms are included in this form of the material balance. This form of the material balance can be useful for certain reactive systems, and is necessary for performing Gibbs energy minimization. The equations written by the Holdup block for total element balances have the form:

$$\sum_p \frac{\partial M_{t,p,e}}{\partial t} = \sum_p N_{transfer,t,p,e}$$

where $M_{t,p,e}$ is the holdup of element e in phase p within the control volume at time t and $N_{transfer,t,p,e}$ is a term to allow for other forms of mass transfer within or across the system boundary.

- none - no material balances are written.

Energy Balance Types

HoldupStatic currently supports only one form of energy balance.

- total - one energy balance is written for the entire holdup, summing contributions from all phases. The form of the total energy balance written by the holdup block is:

$$s \times \sum_p \frac{\partial E_{t,p}}{\partial t} = s \times Q_t + s \times W_t$$

where E_p is the holdup of energy in phase p at time t, Q_t is the heat transferred into the system, W_t is the work transfer into the system and s is a scaling factor.

- none - no energy balances are written.

Momentum Balance Types

As HoldupStatic consists of a single isolated, well-mixed control volume there is no momentum transfer into or out of the volume. As such, there is no need to write a momentum balance for a HoldupStatic block. However, HoldupStatic still maintains the momentum_balance_type construction argument, which can be passed on to associated mixer and splitter blocks.

Supporting Equations

HoldupStatic also creates a number of supporting Variables and Constraints as required by the balance equations, which are summarized below.

Phase Fraction

For systems with more than one phase present (determined automatically from the phase list provided by the property package), HoldupStatic creates a phase fraction variable for each phase, and enforces the following constraint at all points in time t:

$$\sum_{p} \phi_{t,p} = 1$$

When only one phase is present, ϕ is automatically substituted with 1 in all equations.

Holdup Calculations

HoldupStatic also calculates the holdup terms for the material and energy balances (unless include_holdup is False), and automatically writes Constraints for these based on information provided by the property package. The form of the holdup constraint for component balances is:

$$M_{t,p,j} = V_t \times \phi_{t,p} \times \rho_{t,p,j}$$

where $M_{t,p,j}$ is the holdup of species j in phase p at time t, V_t is the volume of the control volume at time t (this supports control volumes of varying volume), $\phi_{t,p}$ is the phase fraction of phase p at time t and $\rho_{t,p,j}$ is the material density of component j in phase p at time t (provided by the Property Block). For phase-component pairs which do not exist (as indicated by the Property Block), the following Constraint is written instead (this is required to close the degrees of freedom):

$$M_{t,p,j} = 0$$

For holdup blocks using element balances, an elemental holdup is required instead, which is calculated as follows:

$$M_{t,p,e} = V_t \times \phi_{t,p} \times \rho_{t,p,j} \times e_{j,e}$$

where $e_{j,e}$ is the number of moles of element e per mole of component j.

For the holdup term in the energy balances, the following Constraints are written:

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

where $E_{t,p}$ is the energy holdup in phase p at time t and $\rho e_{t,p}$ is the volumetric energy density in phase p (provided by the Property Block).

Stoichiometric Constraints

HoldupStatic also automatically generates stoichiometric Constraints to ensure conservation of mass (and elements) between products and reactants in all types of reactions (including phase equilibrium). For kinetic and chemical equilibrium reactions, an extent of reaction terms is generated for each reaction, and Constraint is written to relate these to the generation terms in the balance equations. The form of these equations is:

$$N_{t,p,j} = \sum_{r} \nu_{r,p,j} \times X_{t,r}$$

where $N_{t,p,j}$ is the generation of species j in phase p at time t, $\nu_{r,p,j}$ is the stoichiometric coefficient for the generation of species j in phase p by reaction r and $X_{t,r}$ is the extent of reaction r at time t.

For phase equilibrium reactions, the stoichiometric constraints can be simplified as it is known that the equilibrium is 1:1 between the same species in two different phases. In this case, rather than write separate generation terms for each phase-component pair, a generation term is written for each phase equilibrium reaction that occurs, and the term substituted directly into the material balance (with a ± 1 factor to determine which phase is 'product' phase and which the 'reactant' phase).

Element Flow Terms

Element balances require an additional set of equations for relating component based flows to their element equivalents. When element balances are chosen for a holdup block, the following additional Constraints are written:

$$F_{t,p,e} = F_{t,p,j} \times e_{j,e}$$

where $e_{j,e}$ is the number of moles of element e per mole of component j.

Construction Options

Options available in HoldupStatic for specifying which terms should appear within the balance equations. Most terms in the balance equations have an associated construction argument, which are described below:

- dynamic - controls whether accumulation terms will be included in the balance equations, and the necessary constraints for relating these to the holdup terms. If set to True, accumulation terms will be constructed and included in the model. The number of accumulation terms created depends on the number of time points, phases and components present in the model.
- include_holdup - controls whether terms and constraints for the holdup of material and energy should be constructed (momentum holdup is not yet supported). If set to True, material and energy holdup terms are created, along with constraints linking these to the volume of the holdup and the material and energy densities. If dynamic = True, include_holdup must also be True (this is automatically checked by the framework, and a warning raised if set incorrectly). If dynamic = False, then include_holdup may be set to True or False as desired by the user, depending on whether the holdup terms are required in the flowsheet.
- has_rate_reactions - controls whether generation terms should be constructed for rate controlled reactions. If this is True, generation terms are created for each phase and component pair and included in the material balance equations. Additionally, an extent of reaction term is created for each reaction identified in the associated property package, and linked to the generation term via stoichiometric constraints. The Unit model or user is then expected to provide a set of Constraints relating the extent of reaction terms to the performance of the given unit operation.
- has_equilibrium_reactions - controls whether generation terms should be constructed for equilibrium controlled reactions (excluding phase equilibrium). If this is True, generation terms are created for each phase-component pair, and included in the material balance equations. Additionally, stoichiometric constraints are written relating the different generation term. The property package is expected to provide a set of equilibrium constraints which enforce the equilibrium conditions.
- has_phase_equilibrium - controls whether generation terms should be constructed for phase equilibrium reactions. If this is True, generation terms are created for each phase-component pair, and included in the material balance equations. The property package is expected to provide a set of equilibrium constraints which enforce the equilibrium conditions.
- has_mass_transfer - controls whether mass transfer terms should be constructed in the material balance equations. If True, the generic mass transfer terms will be included, and the UnitModel or user will need to provide constraints for these terms.
- has_heat_transfer - controls whether the heat transfer term should be included in the energy balance equation. If True, the Q term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_work_transfer - controls whether the work transfer term should be included in the energy balance equation. If True, the W term is constructed and the UnitModel or user will need to provide a constraint for this term.
- has_pressure_change - unused.

HoldupStatic Variables

The following is a table of all variables that may be constructed by a HoldupStatic block (depending on options chosen), along with the names used to identify these quantities within actual code (indices are shown in the same order they appear in the code).

Variable	Name
V_t	volume
$M_{t,p,j}$	material_holdup
$\frac{\partial M_{t,p,j}}{\partial t}$	material_accumulation
$M_{t,p,e}$	element_holdup
$\frac{\partial M_{t,p,e}}{\partial t}$	element_accumulation
$E_{t,p}$	energy_holdup
$\frac{\partial E_{t,p}}{\partial t}$	energy_accumulation
$\phi_{t,p}$	phase_fraction
$N_{kinetic,t,p,j}$	rate_reaction_generation
$N_{equilibrium,t,p,j}$	equilibrium_reaction_generation
$N_{pe,t,r}$	phase_equilibrium_generation
$N_{transfer,t,p,j}$	mass_transfer_term
$N_{transfer,t,p,e}$	elemental_mass_transfer
Q_t	heat
W_t	work
$X_{kinetic,t,r}$	rate_reaction_extent
$X_{equilibrium,t,r}$	equilibrium_reaction_extent

Additionally, one scaling factor is generated for the energy balances, named scaling_factor_energy.

Initialization

HoldupStatic has an initialization method which can be called as part of initializing an associated UnitModel. The initialization method takes a set of state arguments which is passed on to the associated property package, mixers and splitters. As initialization of UnitModels often require the inlet state of the unit to be held at a fixed state whilst initializing the unit, the holdup initialization routine can be instructed to hold the inlet state at a fixed state until instructed to unfix it (an associated release_state method exists for this purpose). The procedure followed by the initialization routine is as follows:

1. If an inlet mixer is present, the initialization routine of the mixer is called.

2. The Property Block is initialized (properties). 4. If an outlet splitter is present, the initialization routine of the splitter is called. 5. If not instructed to hold the inlet state fixed, the release_state method is called, otherwise a dictionary of information on what state variables were fixed is returned to be used when calling release_state.

The associated release_state method takes the dictionary of flags returned above, and uses this to unfix any variables fixed during initialization.

HoldupStaticData Class

class idaes.core.holdup.**HoldupStaticData**(*component*)

Static Holdup Class

This class is designed to be used for unit operations zero volume or holdups with no through flow (such as dead zones). This type of holdup has only a single PropertyBlock index by time (Holdup0D has two).

build()
Build method for HoldupStatic blocks. This method calls submethods to setup the necessary property blocks, distributed variables, material, energy and momentum balances based on the arguments provided by the user.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)
Initialisation routine for holdup (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 initialisation routine
  - 0 = no output (default)
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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). - True - states varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()
This method exectues the model_check methods on the associated property blocks (if they exist). This method is generally called by a unit model as part of the unit's model_check method.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging

Returns None

Ports

Contents

- *Ports*
  - *Introduction*
  - *Port Class*
  - *Inlet Mixer Class*
  - *Outlet Splitter Class*

Introduction

Port Blocks are used by the IDAES framework whenever multiple inlets or outlets are required by a single Holdup Block, and are generally constructed automatically by the *build_inlets* and *build_outlets* methods at the Unit model level. Port Blocks contain a set of Property Blocks for each of the multiple inlet or outlet streams along with the necessary mixing or splitting constraints to relate these to the flow into or out of the Holdup Block.

Port Class

The Port class serves as a base class for all Port Blocks, and automates the task common to all of these. The main tasks of the Port class are:

1. Make references to the time, component_list and phase_list indexing Sets to use when constructing Constraints.
2. Determine which mixing or splitting equations are required by checking the config block of the parent Holdup Block.

Port Construction Arguments

The construction arguments for Port Blocks are determined automatically based on the construction arguments of the assoicated Holdup Block. The available arguments and their values are:

- has_material_balance - indicates whether material mixing/splitting constraints should be constructed. False if parent.config.material_balance_type is 'none', otherwise True.
- has_energy_balance - indicates whether energy mixing/splitting constraints should be constructed. False if parent.config.energy_balance_type is 'none', otherwise True.
- has_momentum_balance - indicates whether momentum mixing/splitting constraints should be constructed. False if parent.config.momentum_balance_type is 'none', otherwise True.

Port Class

class idaes.core.ports.**Port**(*component*)

Base Port Class

This class contains methods common to all Port classes.

build()
General build method for Ports. This method calls a number of methods common to all Port blocks.

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

Parameters **None** –

Returns None

Inlet Mixer Class

InletMixer Blocks are created as a sub-model of any Holdup Block which is declared to have multiple inlets. Based on the configuration arguments of the Holdup Block, the following mixing Constraints are written as required.

InletMixer Construction Arguments

InletMixer have one additional construction argument beyond those inherited from Port.

- inlets - a list of names to use to index the inlets. Used as indices for the inlet Property Blocks constructed by the InletMixer.

Material Mixing Constraints

$$\sum_i F_{t,i,p,j} = F_{t,p,j}$$

where $F_{t,i,p,j}$ is the flow of component j in phase p in inlet i at time t and $F_{t,p,j}$ is the combined flow of component j in phase p at time t across all inlets.

Energy Mixing Constraint

$$s_e \cdot \sum_i H_{t,i,p} = s_e \cdot H_{t,p}$$

where $H_{t,i,p}$ is the flow of energy in phase p in inlet i at time t and $H_{t,p,j}$ is the combined flow of energy in phase p at time t across all inlets. The equation scaling factor s_e is a mutable parameter named *scaling_factor_energy* in the inlet mixer with a default value of 1×10^{-6}.

Pressure Mixing Constraint

For determining the pressure of the mixed stream, the minimum pressure of all the inlets is used. This is calculated as follows:

for i in inlets:

if i = 1: $s_p \cdot P_{min,t,i} = s_p \cdot P_{i,t}$

else: $s_p \cdot P_{min,t,i} = s_p \cdot smin(P_i, P_{min,i-1})$

Here, $P_{t,i}$ is the pressure in inlet i at time t, $P_{min,t,i}$ is an intermediate variable used to calculate the minimum pressure in all inlets tested so far and $smin$ is a smooth minimum function. The smoothing parameter for $smin$ is named *eps_pressure* and has a default value of 1×10^{-3}. The equation scaling factor s_p is a mutable parameter named *scaling_factor_pressure* in the inlet mixer with a default value of 1×10^{-5}.

InletMixerData Class

class idaes.core.ports.**InletMixerData**(*component*)

Inlet Mixer Class

This class builds a mixer to allow for multiple inlets to a single holdup block. The class constructs property blocks for each inlet and creates mixing rules to connect them to the property block within the associated holdup block.

build()

Build method for Mixer blocks. This method calls a number of methods to construct the necessary balance equations for the Mixer.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)

Initialisation routine for InletMixer (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 initialisation routine
  - 0 = no output (default)
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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).

  - **True = state varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()

Calls model checks on all associated Property Blocks.

Parameters **None** –

Returns None

release_state (*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging (default=0)

Returns None

Outlet Splitter Class

OutletSplitter Blocks are created as a sub-model of any Holdup Block which is declared to have multiple outlets. Based on the configuration arguments of the Holdup Block, the following splitting Constraints are written as required.

OutletSplitter Construction Arguments

OutletSplitters have two additional construction argument beyond those inherited from Port.

- outlets - a list of names to use to index the outlets. Used as indices for the outlet Property Blocks constructed by the OutletSplitter.
- split_type - determines the method to be used when splitting the outlet stream. Options are:
  - 'flow' - outlet streams are split based on total flow. All resulting streams have the same intensive state.
  - 'phase' - outlet streams are split by phase fractions. A specified portion of each phase is sent to each outlet.
  - 'component' - outlet streams are split by component. A specified fraction of each component is sent to each outlet.
  - 'total' - outlet streams are split based on phase and component. A specified fraction of each phase-component pair is sent to each outlet.

Split Fraction Constraints

OutletSplitter creates a split_fraction variable which is indexed as determined by the type of split being performed:

- flow - indexed by time and outlet only
- phase - indexed by time, outlet and phase
- component - indexed by time, outlet and component
- total - indexed by time, outlet, phase and component

In all cases, a Set of Constraints are written such the the sum of split fractions for a given outlet must be 1.

$$1 = \sum_{o} sf_{t,o,...}$$

Material Splitting Constraints

$$F_{t,o,p,j} = sf_{t,o,...} \times F_{t,p,j}$$

where $F_{t,o,p,j}$ is the flow of component j in phase p in outlet o at time t after splitting and $F_{t,p,j}$ is the total flow of component j in phase p at time t leaving the holdup.

Energy Splitting Constraints

Energy splitting is handled differently depending on the split type chosen. If split_type is 'phase' then the following Constraint is written:

$$\sum_p H_{t,o,p} = \sum_p sf_{t,o,p} \times H_{t,p}$$

where $H_{t,o,p}$ is the energy flow in phase p in outlet o at time t and $H_{t,p}$ is the total energy flow in phase p at time t leaving the Holdup.

If split_type is not equal to 'phase', then the outlet temperature of all streams is set to be equal.

$$T_{t,o} = T_t$$

where $T_{t,o}$ is the temperature of outlet o at time t and T_t is the temperature of the total flow leaving the Holdup at time t.

Momentum Splitting Constraints

The momentum splitting constraint equates the pressure in each outlet to the pressure of the material leaving the Holdup.

$$P_{t,o} = P_t$$

where $P_{t,o}$ is the pressure of outlet o at time t and P_t is the pressure of the total flow leaving the Holdup at time t.

OutletSplitterData Class

class idaes.core.ports.**OutletSplitterData**(*component*)

Outlet Mixer Class

This class builds a splitter to allow for multiple outlets to a single holdup block. The class constructs property blocks for each outlet and creates splitting rules to connect them to the property block within the associated holdup block.

build()

Build method for Splitter blocks. This method calls a number of methods to construct the necessary balance equations for the Splitter.

Parameters **None** –

Returns None

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

Initialisation routine for OutletSplitter (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 initialisation routine
  - 0 = no output (default)
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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).

  - **True = state varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()

Calls model checks on all associated Property Blocks.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)

Method to relase state variables fixed during initialisation.

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 of logging (default=0)

Property Package Classes

Contents

- *Property Package Classes*
  - *Introduction*
  - *Property Parameter Blocks*
  - *Property Blocks*

Introduction

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.

Property Parameter Blocks

Property 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 proeprty package.

PropertyParameterBase Class

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

class idaes.core.property_base.**PropertyParameterBase**(*component*)

This is the base class for property parameter blocks. These are blocks that contain a set of parameters associated with a specific 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

get_package_units()

Method to return a dictionary of default units of measurement used in the property package. This is used to populate doc strings for variables which derive from the property package (such as flows and volumes). This method should return a dict with keys for the quantities used in the property package (as strs) and values of their default units as strs.

The quantities used by the framewokr are (all optional):

- 'time'
- 'length'
- 'mass'
- 'amount'
- 'temperature'
- 'energy'
- 'current'
- 'luminous intensity'

This default method is a placeholder and should be overloaded by the package developer. This method will return an Exception if not overloaded.

Parameters **None** –

Returns A dict with supported properties as keys and tuples of (method, units) as values.

get_supported_properties()
Method to return a dictionary of properties supported by this package and their assoicated construction methods and units of measurement. This method should return a dict with keys for each supported property.

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.

This default method is a placeholder and should be overloaded by the package developer. This method will return an Exception if not overloaded.

Parameters **None** –

Returns A dict with supported properties as keys.

Property Blocks

Property Blocks are used within all IDAES Unit models (generally within Holdup Blocks) in order to calculate properties given the state of the material. Property Blocks are notably differnt to other types of Blocks within IDAES as they are always indexed by time (and possibly space as well). There are two bases Classes associated with Property Blocks:

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

Construction Arguments

Property Blocks have the following construction arguments:

- parameters - a reference to the associated Property Parameter block which will be used to make references to all necessary parameters.
- has_sum_fractions - this argumnet indicates whether the Property Block should construct constraints which eforce the sum of material fractions (e.g. mass or mole fractions) within the block. This is primarily set to False for unit inlets where it is assumed that the material state is fully specified. This argument can also be used in some conjunction with the following argumnets to indicate whether equilibrium should be enforced.
- calculate_equilibrium_reactions - indicates whether the assoicated Holdup Block or Unit model expects chemical equilibrium to be enforced (if applicable).
- calculate_phase_equilibrium - indicates whether the assoicated Holdup Block or Unit model expects phase equilibrium to be enforced (if applicable).

PropertyBlockDataBase Class

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

class idaes.core.property_base.**PropertyBlockDataBase**(*component*)
This is the base class for property block data objects. These are blocks that contain the Pyomo components associated with calculating a set of thermophysical, transport and reacion properties for a given material.

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

Parameters **None** –

Returns None

PropertyBlockBase Class

class idaes.core.property_base.**PropertyBlockBase**(**args*, ***kwargs*)
This is the base class for property 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 PropertyBlockData objects simultaneously.

initialize(**args*)
This is a default initialization routine for PropertyBlocks to ensure that a routine is present. All PropertyBlockData classes should overload this method with one suited to the particular property package

This method prints a warning and then ends.

Parameters **None** –

Returns None

IDAES Base Classes

Contents

- *IDAES Base Classes*
  - *Introduction*
  - *ProcessBlockData Class*
  - *ProcessBlock Class*
  - *declare_process_block_class Decorator*

Introduction

All of the modeling classes described above build of a set of base classes within the IDAES core. These base classes handle the interaction between the IDAES framework and the underlying Pyomo framework.

ProcessBlockData Class

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()
Default build method for all Classes inheriting from ProcessBlockData. Currently empty, but left in place to allow calls to super().build and for future compatability.

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

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

Parameters **None** –

Returns : None

ProcessBlock Class

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

Process block behaves like a Pyomo Block. The important differences are listed below.

- There is a default rule that calls the build() method for _BlockData subclass ojects, so subclass of _BlockData used in a ProcessBlock should have a build() method. A different rule or no rule (None) can be set with the usual rule argument, if additional steps are required to build an element of a block. A example of such a case is where different elements of an indexed block require addtional information to construct.
- Some of the arguments to __init__, which are not expected arguments of Block, are split off and stored in self._block_data_config. If the _BlockData subclass inherits ProcessBlockData, self._block_data_config is sent to the self.config ConfigBlock.

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.

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 _BlockData. It creates a ProcessBlock subclass to contain it. For example (where ProcessBlockData is a subclass of _BlockData):

@declare_process_block_class(name=MyUnitBlock) class MyUnitBlockData(ProcessBlockData):

This class is a _BlockData subclass contained in a Block subclass # MyUnitBlock ....

The only requirment is that the subclass of _BlockData contain a build() method.

Parameters

- **name** – class name for the model.
- **block_class** – ProcessBlock or a subclass of ProcessBlock, this allows you to use a subclass of ProcessBlock if needed.
- **doc** – Documentation for the class. This should play nice with sphinx.

IDAES Utility Functions

The IDAES library also includes a number of modules which contain utility functions of potential use to modellers.

Serialize Pyomo Model States

This describes the IDAES Pyomo model state JSON serializer module (idaes_models.core.util.model_serializer). All objects inheriting from idaes_models.core.process_base.ProcessBlock (which inherits Pyomo Block) have a from_json and to_json method that uses the functions in this module, passing self for o, and the rest of the arguments remaining the same.

This module can load/save the model state from/to a JSON file or an in memory as a Python dictionary. The dictionary is in a form that can be dumped to JSON.

The following describes the important functions and classes in this module.

Contents

- *IDAES Utility Functions*
  - *util.config*
  - *util.misc*

util.config

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_parameter_block**(*val*)

Domain validator for property package attributes

Parameters **val** – value to be checked

Returns TypeError if val is not an instance of PropertyParameterBase, 'use_parameter_block' or None

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.**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

util.misc

This module contains miscellaneous utility functions that of general use in IDAES models.

idaes.core.util.misc.**add_object_ref**(*local_block*, *local_name*, *external_component*)
Add a reference in a model to non-local Pyomo component. This is used when one Block needs to make use of a component in another Block as if it were part of the local block.

Parameters

- **local_block** – Block in which to add reference
- **local_name** – str name for referenced object to use in local_block
- **external_component** – external component being referenced

Returns None

idaes.core.util.misc.**category**(**args*)
Decorate tests to enable tiered testing.

Suggested categories:

1. frequent
2. nightly
3. expensive
4. research

Parameters ***args** (*tuple of strings*) – categories to which the test belongs

Returns Either the original test function or skip

Return type function

idaes.core.util.misc.**dict_set**(*v*, *d*, *pre_idx=None*, *post_idx=None*, *fix=False*)
Set the values of array variables based on the values stored in a dictionary. There may already be a better way to do this. Should look into it.

The value of Pyomo variable element with index key is set to d[key]

Arguments: v: Indexed Pyomo variable d: dictonary to set the variable values from, keys should match a subset

of Pyomo variable indexes.

pre_idx: fixed indexes before elements to be set or None post_idx: fixed indexes after elements to be set or None fix: bool, fix the variables (otional)

idaes.core.util.misc.**doNothing**(**args*, ***kwargs*)
Do nothing.

This function is useful for instances when you want to call a function, if it exists. For example: getattr(unit, 'possibly_defined_function', getNothing)()

Parameters

- ***args** (*anything*) – accepts any argument
- ****kwargs** (*anything*) – accepts any keyword arguments

Returns None

idaes.core.util.misc.**fix_port**(*port*, *var*, *comp=None*, *value=None*, *port_idx=None*)
Method for fixing Vars in Ports.

Parameters

- **port** – Port object in which to fix Vars
- **var** – variable name to be fixed (as str)
- **comp** – index of var to be fixed (if applicable, default = None)
- **value** – value to use when fixing var (default = None)
- **port_idx** – list of Port elements at which to fix var. Must be list of valid indices,

Returns None

idaes.core.util.misc.**get_pyomo_tmp_files**()
Make Pyomo write it's temporary files to the current working directory, useful for checking nl, sol, and log files for ASL solvers without needing to track down the temporary file location.

idaes.core.util.misc.**get_time**(*results*)
Retrieve the solver-reported elapsed time, if available.

idaes.core.util.misc.**hhmmss**(*sec_in*)
Convert elapsed time in seconds to "d days hh:mm:ss.ss" format. This is nice for things that take a long time.

idaes.core.util.misc.**requires_solver**(*solver*)
Decorate test to skip if a solver isn't available.

idaes.core.util.misc.**round_**(*n*, **args*, ***kwargs*)
Round the number.

This function duplicates the functionality of round, but when passed positive or negative infinity, simply returns the argument.

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

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.misc.**smooth_max**(*a*, *b*, *eps=0.0001*)
Smooth maximum operator.

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.misc.**smooth_min**(*a*, *b*, *eps=0.0001*)
Smooth minimum operator.

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.misc.**smooth_minmax**(*a*, *b*, *eps=0.0001*, *sense='max'*)
General function for creating an expression for a smooth minimum or maximum.

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.**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

- **solve** – 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.misc.**unfix_port**(*port*, *var*, *comp=None*, *port_idx=None*)
Method for unfixing Vars in Ports.

Parameters

- **port** – Port object in which to unfix Vars
- **var** – variable name to be unfixed (as str)

- **comp** – index of var to be unfixed (if applicable, default = None)
- **port_idx** – list of Port elements at which to unfix var. Must be list of valid indices,

Returns None

3.2.5 Models

This section documents the models available in the IDAES Unit Model Library which are available for use. All models in the library are built using the IDAES modeling framework, and are built of the UnitModelData class make use of Holdup Blocks to construct the material, energy and momentum balances. For documentation of the equations written by these, refer to the relevant documentation for Unit Models and Holdup Blocks.

Contents

Feed and Product Blocks

The IDAES model library contains special "units" for representing feeds (sources) and products (sinks) in Flowsheets. These can be used to define feed conditions, especially when the known conditons do not match with the state variables defined in the Property Package (e.g. property package used enthalpy as a state varaible, but temeprature is the known quantity).

Feed Block

Feed Blocks are used to represent sources of material in Flowsheets. These can be used to determine the full state of a material (including equilibrium) based on a sufficient set of state variables prior to being passed to the first unit operation.

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 HoldupStatic Block (named holdup), each with one Outlet Port (named outlet).

Construction Arguments

The Feed model has the following construction arguments:

- property_package - property package to use when constructing Property Block for the Feed Block (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 Block when they are created.
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Feed Blocks have the following construction arguments which are passed to the Holdup Block for determining which terms to construct in the balance equations. Feed Blocks do not support dynamic = True.

Argument	Default Value
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False (cannot be True)
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

Additional Methods

Feed Blocks define three additional methods useful for defining and interpreting the feed conditions. These methods are documented below in the FeedData Class section;

1. display()
2. fix()
3. unfix()

Additional Constraints

Feed Blocks write no additional constraints to the model.

Variables

Feed blocks add no additional Variables.

FeedData Class

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

Standard Feed Block Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

display(*display_block=False*, *ostream=None*, *prefix=''*)

Display the contents of Feed unit.

Parameters

- **display_block** – indicates whether to display the entire Block

- **object** (*default = False*) –
- **ostream** – output stream (default = None)
- **prefix** – str to append to each line of output (default = '')

Returns None

fix(*var*, *comp=None*, *value=None*, *time=None*)
Method for fixing Vars in Feed Block.

Parameters

- **var** – variable name to be fixed (as str)
- **comp** – index of var to be fixed (if applicable, default = None)
- **value** – value to use when fixing var (default = None)
- **time** – list of time points at which to fix var (can be float, int
- **list)** (*or*) –

Returns None

post_transform_build()
Continue model construction after DAE transformation.

Parameters **None** –

Returns None

unfix(*var*, *comp=None*, *time=None*)
Method for unfixing Vars in Feed Block.

Parameters

- **var** – variable name to be unfixed (as str)
- **comp** – index of var to be unfixed (if applicable, default = None)
- **time** – list of time points at which to unfix var (can be float, int
- **list)** (*or*) –

Returns None

Product Block

Product Blocks are used to represent sinks of material in Flowsheets. These can be used as a conventient way to mark the final destination of a matieral 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 HoldupStatic Block (named holdup), each with one Inlet Port (named inlet).

Construction Arguments

The Product model has the following construction arguments:

- property_package - property package to use when constructing Property Block for the Product Block (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 Block when they are created.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).

Additionally, Product Blocks have the following construction arguments which are passed to the Holdup Block for determining which terms to construct in the balance equations. Product Blocks do not support dynamic = True.

Argument	Default Value
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False (cannot be True)
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

Additional Methods

Product Blocks define an additional display() method to output the state of the product material. This method is documented below in the ProductData Class section.

Additional Constraints

Product Blocks write no additional constraints to the model.

Variables

Product blocks add no additional Variables.

ProductData Class

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

Standard Product Block Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

display(*display_block=False*, *ostream=None*, *prefix=''*)
Display the contents of Product unit.

Parameters

- **display_block** – indicates whether to display the entire Block
- **object** (*default = False*) –
- **ostream** – output stream (default = None)
- **prefix** – str to append to each line of output (default = '')

Returns None

post_transform_build()
Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Mixers, Splitters and Separators

Mixer Unit

The IDAES Mixer unit model can be used to represent different types of equipment for mixing streams of material. Mixer Blocks can be used to represent a number of different types of behavior by choosing the appropriate construction arguments.

Degrees of Freedom

Mixers generally have zero degrees of freedom.

Model Structure

The structure of a Mixer unit depends on the construction options chosen. A Mixer unit contains a single Holdup Block (named holdup), the type of which depends on the options as follows;

1. If include_holdup, has_equilibrium_reactions or has_mass_transfer is True, a Holdup0D Block is used,
2. Otherwise, a HoldupStatic Block is used.

Additionally, a Mixer has one Inlet Port object (named inlet and indexed by a list of names) and one Outlet Port object (named outlet).

Construction Arguments

The Mixer model has 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = 2).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Mixer Blocks 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

Additional Constraints

Mixer Blocks write no additional constraints to the model.

Variables

Mixer Blocks add no additional Variables.

MixerData Class

class idaes.models.mixer.**MixerData**(*component*)

Standard Mixer Unit Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Splitter Unit

The IDAES Splitter unit model can be used to represent different types of equipment for splitting streams of material based on total flow. For other types of separation behavior, see the Separator unit model.

Degrees of Freedom

Splitter units generally have degrees of freedom equal to the number of outlets - 1.

Typical fixed variables are:

- split fractions for outlets-1 streams.

Model Structure

The structure of a Splitter unit depends on the construction options chosen. A Splitter unit contains a single Holdup Block (named holdup), the type of which depends on the options as follows;

1. If include_holdup, has_equilibrium_reactions or has_mass_transfer is True, a Holdup0D Block is used,
2. Otherwise, a HoldupStatic Block is used.

Additionally, a Splitter has one Inlet Port object (named inlet) and one Outlet Port object (named outlet and indexed by a list of names).

Construction Arguments

The Splitter model has 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = 2).

Additionally, Splitter Blocks 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

Additional Constraints

Splitter Blocks write no additional constraints to the model.

Variables

Splitter Blocks add one additional Variable beyond those created by the Holdup Block.

Name	Notes
split_fraction	Reference to holdup.outlet_splitter.split_fraction

SplitterData Class

class idaes.models.splitter.**SplitterData**(*component*)

Standard Splitter Unit Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Separator Unit

The IDAES Separator unit model is a general purpose model for different types of equipment for separating or splitting material flows. The Separator unit model supports different methods for separating the material flow to represent different types of equipment. For separations based on total flow, see Splitter.

Degrees of Freedom

Separator units generally have degrees of freedom related to the number of outlet streams and the split type chosen.

- If split_type = 'phase', degrees of freedom are generally $(no.outlets - 1) \times no.phases$
- If split_type = 'component', degrees of freedom are generally $(no.outlets - 1) \times no.components$
- If split_type = 'total', degrees of freedom are generally $(no.outlets - 1) \times no.phases \times no.components$

Typical fixed variables are:

- split fractions.

Model Structure

The structure of a Separator unit depends on the construction options chosen. A Separator unit contains a single Holdup Block (named holdup), the type of which depends on the options as follows;

1. If include_holdup, has_equilibrium_reactions or has_mass_transfer is True, a Holdup0D Block is used,
2. Otherwise, a HoldupStatic Block is used.

Additionally, a Separator has one Inlet Port object (named inlet) and one Outlet Port object (named outlet and indexed by a list of names).

Construction Arguments

The Splitter model has the following construction arguments:

- separation_type - indicates which method to use when separating the outlet material flow. Options are:
  - 'phase' - outlet streams are split by phase fractions. A specified portion of each phase is sent to each outlet.
  - 'component' - outlet streams are split by component. A specified fraction of each component is sent to each outlet.
  - 'total' - outlet streams are split based on phase and component. A specified fraction of each phase-component pair is sent to each outlet.
- 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = 2).

Additionally, Separator Blocks 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

Additional Constraints

Separator Blocks write no additional constraints to the model.

Variables

Separator Blocks add one additional Variable beyond those created by the Holdup Block.

Name	Notes
split_fraction	Reference to holdup.outlet_splitter.split_fraction

SeparatorData Class

class idaes.models.separator.**SeparatorData**(*component*)

Standard Splitter Unit Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Pressure and Temperature Changers

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 Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

Pressure Changers have the following construction arguments:

- compressor - argument indicates whether the unit should be considered a compressor (True, default) or an expander/turbine (False). This determines how unit efficiency is calculated.
- thermodynamic_assumption - indicates which thermodynamic assumption should be used when constructing the model. Options are:
  - 'isothermal' - (default) assumes no temperature change occurs between the inlet and outlet of the unit.
  - 'isentropic' - assumes isentropic behavior. This requires an additional set of property calculations for the isentropic outlet conditions.
  - 'pump' - assumes that the fluid work is proportional to the pressure difference and flow rate of fluid. This is suitable for incompressible fluids.
- 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Pressure Changers 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	True
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	True
has_pressure_change	True

Additional Constraints

In addition to the Constraints written by the Holdup 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:

$$T_{out} = T_{in}$$

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:

$$F_{in,t,p,j} = F_{out,t,p,j}$$

$$s_{in,t} = s_{isentropic,t}$$

$$P_{in,t} \times P_{ratio,t} = P_{isentropic,t}$$

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, $W_{fluid,t} \times \eta_t = W_{mechanical,t}$

Variables

Pressure Changers contain the following Variables (not including those contained within the Holdup Block):

Variable	Name	Notes
P_{ratio}	ratioP	
V_t	volume	Only if has_rate_reactions = True, reference to holdup.rate_reaction_extent
$W_{mechanical,t}$	work_mechanical	Reference to holdup.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, not the Holdup Block).

PressureChangerData Class

class idaes.models.pressure_changer.**PressureChangerData**(*component*)

Compressor/Expander Unit Class

build()

Begin building model (pre-DAE transformation)

Parameters **None** –

Returns None

init_isentropic(*state_args*, *outlvl*, *solver*, *optarg*)

Initialisation 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 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

initialize(*state_args={}*, *routine=None*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)

General wrapper for pressure changer initialisation 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 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()

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

post_transform_build()

Continue model construction after DAE transformation

Parameters **None** –

Returns None

Temperature Changer

The IDAES Temperature Changer model represents a unit operation with a single stream of material which undergoes a change in temperature due to the application of a heat duty.

Degrees of Freedom

Temperature Changers generally have 1 degree of freedom .

Typical fixed variables are:

- heat duty, ΔT or outlet temperature.

Model Structure

The core Temperature Changer unit model consists of a single Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

Temperature Changers have the following construction arguments:

- heater - argument indicates whether the unit should be considered a heater (True, default) or a cooler (False).
- 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Temperature Changers 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	True
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	False

Additional Constraints

In addition to the Constraints written by the Holdup Block, Temperature Changer writes one additional Constraint:

$$\Delta T = T_{out} - T_{in}$$

Variables

Temperature Changers contain the following Variables (not including those contained within the Holdup Block):

Variable	Name	Notes
ΔT	deltaT	
V_t	volume	Only if include_holdup = True, reference to holdup.volume
Q_t	heat	Only if has_heat_transfer = True, reference to holdup.heat

TemperatureChangerData Class

class idaes.models.temperature_changer.**TemperatureChangerData**(*component*)

Standard Temperature Changer Unit Model Class

build()

Begin building model (pre-DAE transformation)

Parameters **None** –

Returns None

model_check()

Check that temperature change matches with heater argument (i.e. if heater = True, temperature should increase).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation

Parameters **None** –

Returns None

0-D Heat Exchanger (HeatExchanger)

Heat Exchanger models represents a unit operation with two material streams which exchange heat. HeatExchanger (0-D) is used for modeling heat exchangers using an average heat transfer coefficient and driving force. For more complex models involving spatial domains, see HeatExchanger1D.

Degrees of Freedom

0-D Heat Exchangers generally have 2 degrees of freedom.

Typical fixed variables are:

- heat transfer area,
- heat transfer coefficient.

Alternatives include:

- outlet temeprature of one stream,
- heat duty.

Model Structure

The 0-D Heat Exchanger unit model consists of two Holdup0D Blocks (named side_1 and side_2), each with one Inlet Port (named side_1_inlet and side_2_inlet) and one Outlet Port (named side_1_outlet and side_2_outlet).

Construction Arguments

The 0-D Heat Exchanger model has the following construction arguments:

- flow_type - argument indicating the flow arrangement within the Heat Exchanger. Currently only supports one option:
  - 'counter-current' - (default) counter current flow arrangement.
- side_1_property_package - property package to use when constructing Property Blocks for side_1 of the Heat Exchanger (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.
- side_1_property_package_args - set of arguments to be passed to the Property Blocks for side_1 when they are created.
- side_1_inlet_list - list of names to be passed to the build_inlets method for side_1 (default = None).
- side_1_num_inlets - number of inlets argument to be passed to the build_inlets method for side_1 (default = None).
- side_1_outlet_list - list of names to be passed to the build_outlets method for side_1 (default = None).
- side_1_num_outlets - number of outlets argument to be passed to the build_outlets method for side_1 (default = None).
- side_2_property_package - property package to use when constructing Property Blocks for side_2 of the Heat Exchanger (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.
- side_2_property_package_args - set of arguments to be passed to the Property Blocks for side_2 when they are created.
- side_2_inlet_list - list of names to be passed to the build_inlets method for side_2 (default = None).
- side_2_num_inlets - number of inlets argument to be passed to the build_inlets method for side_2 (default = None).
- side_2_outlet_list - list of names to be passed to the build_outlets method for side_2 (default = None).
- side_2_num_outlets - number of outlets argument to be passed to the build_outlets method for side_2 (default = None).

Additionally, 0-D Heat Exchangers have the following construction arguments which are passed to the Holdup Blocks for determining which terms to construct in the balance equations.

Argument	Default Value
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	True
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	False

Additional Constraints

In addition to the Constraints written by the Holdup Blocks, 0-D Heat Exchanger models write the following Constraints:

$$Q_{1,t} = -Q_{2,t}$$

where $Q_{1,t}$ and $Q_{2,t}$ are the heat transfer in the side_1 and side_2 Holdup Blocks respectively.

$$Q_{1,t} = U \times A \times \Delta T_{m,t}$$

where U is the average heat transfer coefficient (assumed constant for now), A is the heat transfer area (also assumed constant) and $\Delta T_{m,t}$ is the mean temperature driving force at time t. The mean temperature driving force is represented using the log-mean temperature difference:

$$(\Delta T_{1,t} - \Delta T_{2,t}) = \Delta T_{m,t} \times log(\frac{\Delta T_{1,t}}{\Delta T_{2,t}})$$

where $\Delta T_{1,t}$ and $\Delta T_{2,t}$ are the temperature difference at the side_1 inlet and outlet respectively. The Constraints defining these depend on the flow configuration of the unit.

Counter-Current Flow

If flow_type is set to 'counter-current', the following Constraints are used to define $\Delta T_{1,t}$ and $\Delta T_{2,t}$:

$$\Delta T_{1,t} = T_{side_1 in,t} - T_{side_2 out,t}$$

$$\Delta T_{2,t} = T_{side_1 out,t} - T_{side_2 in,t}$$

Variables

0-D Heat Exchangers contain the following Variables (not including those contained within the Holdup Blocks):

Variable	Name	Notes
$V_{1,t}$	side_1_volume	Only if include_holdup = True, reference to side_1.volume
$V_{2,t}$	side_2_volume	Only if include_holdup = True, reference to side_2.volume
Q_t	heat	Reference to side_1.heat
A	heat_transfer_area	
U	heat_transfer_coefficient	
$\Delta T_{m,t}$	temperature_driving_force	
$\Delta T_{1,t}$	side_1_inlet_dT	
$\Delta T_{2,t}$	side_1_outlet_dT	

HeatExchangerData Class

class idaes.models.heat_exchanger.**HeatExchangerData**(*component*)

Standard Heat Exchanger Unit Model Class

build()

Begin building model (pre-DAE transformation)

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 Holdup Blocks.

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation

Parameters **None** –

Returns None

1-D Heat Exchanger (HeatExchanger1D)

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 tube,
- 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 Holdup1D 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 the following construction arguments:

- 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).
- shell_property_package - property package to use when constructing shell 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 Holdup Block will try to use the default property package if one is defined.
- shell_property_package_args - set of arguments to be passed to the shell side Property Blocks when they are created.
- shell_inlet_list - list of names to be passed to the shell side build_inlets method (default = None).
- shell_num_inlets - number of inlets argument to be passed to the shell side build_inlets method (default = None).
- shell_outlet_list - list of names to be passed to the shell side build_outlets method (default = None).
- shell_num_outlets - number of outlets argument to be shell side passed to the build_outlets method (default = None).
- tube_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 Holdup Block will try to use the default property package if one is defined.

- tube_property_package_args - set of arguments to be passed to the tube side Property Blocks when they are created.
- tube_inlet_list - list of names to be passed to the tube side build_inlets method (default = None).
- tube_num_inlets - number of inlets argument to be passed to the tube side build_inlets method (default = None).
- tube_outlet_list - list of names to be passed to the tube side build_outlets method (default = None).
- tube_num_outlets - number of outlets argument to be passed to the tube side build_outlets method (default = None).
- discretization_method_shell - indicates which method to use when discretizing shell side length domain. Note that this should be compatible with the tube side method. Options are:
  - 'OCLR' - orthogonal collocation on finite elements (Radau roots)
  - 'OCLL' - orthogonal collocation on finite elements (Legendre roots)
  - 'BFD' - backwards finite difference (1st order)
  - 'FFD' - forwards finite difference (1st order)
- discretization_method_tube - indicates which method to use when discretizing tube side length domain. Note that this should be compatible with the shell side method. Options are:
  - 'OCLR' - orthogonal collocation on finite elements (Radau roots)
  - 'OCLL' - orthogonal collocation on finite elements (Legendre roots)
  - 'BFD' - backwards finite difference (1st order)
  - 'FFD' - forwards finite difference (1st order)
- 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 = 3, collocation methods only). This is used for both shell and tube side domains.
- has_mass_diffusion - indicates whether mass diffusion terms should be included in the material balance equations (default = False)
- has_energy_diffusion - indicates whether energy conduction terms should be included in the energy balance equations (default = False)

Additionally, 1-D Heat Exchanger 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	True
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
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:

$$Q_{shell,t,x} = -N_{tubes} \times L_{shell} \times (\pi \times U_{shell,t,x} \times D_tube \times (T_{shell,t,x} - T_{wall,t,x}))$$

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.

$$Q_{tube,t,x} = -N_{tubes} \times L_{shell} \times (\pi \times U_{tube,t,x} \times D_tube \times (T_{wall,t,x} - T_{tube,t,x}))$$

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.

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 Holdup Block.

Variable	Name Notes	
L_{shell}	length_shell	Reference to shell.length
A_{shell}	area_shell	Reference to shell.area
D_{shell}	d_shell	
L_{tube}	length_tube	Reference to tube.length
A_{tube}	area_tube	Reference to tube.area
D_{tube}	d_tube	
N_{tubes}	N_tubes	
$T_{wall,t,x}$	temperature_wall	
$U_{shell,t,x}$	heat_transfer_coefficient_shell	
$U_{tube,t,x}$	heat_transfer_coefficient_tube	

HeatExchanger1dData Class

class idaes.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={}*, *tube_state_args={}*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)

Initialisation routine for isothermal 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 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 (call model check on holdups)

Parameters **None** –

Returns None

post_transform_build()
Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Distillation and Flash Separation

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 holdup.properties_out[t].temperature. Setting the temperature in one of the outlet streams directly results in a much harder problme ot solve, and may be degenerate in some cases.

Model Structure

The core Flash unit model consists of a single Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet, default with two indexes ('Vap' and 'Liq')).

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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = ['Vap', 'Liq']).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	False
has_equilibrium_reactions	False
has_phase_equilibrium	True
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	True

Additional Constraints

Flash units write no additional Constraints beyond those written by the Holdup Block.

However, Flash units automatically fix the split fractions of the outlets such that all the liquid phase goes to the outlet named "Liq" and all the vapor phase goes to the outlet named "Vap". This is done as follows for all t in the time domain:

- split_fraction(t, "Liq", "Liq").fix(1.0)
- split_fraction(t, "Vap", "Vap").fix(1.0)

Variables

Flash units add one additional Variable beyond those created by the Holdup Block.

Name	Notes
split_fraction	Reference to holdup.outlet_splitter.split_fraction

FlashData Class

class idaes.models.flash.**FlashData**(*component*)
Standard Flash Unit Model Class

build()
Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()
Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Reactor Unit Models

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 Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

Stoichiometric reactor 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.

- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Stoichiometric reactor 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	True
has_equilibrium_reactions	True
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	False

Additional Constraints

Stoichiometric reactor units write no additional Constraints beyond those written by the Holdup Block.

Variables

Stoichiometric reactors units add the following additional Variables beyond those created by the Holdup Block.

Variable	Name	Notes
V_t	volume	Only if include_holdup = True, reference to holdup.volume
$X_{t,r}$	rate_reaction_extent	Only if has_rate_reactions = True, reference to holdup.rate_reaction_extent
Q_t	heat	Only if has_heat_transfer = True, reference to holdup.heat

StoichiometricReactorData Class

class idaes.models.stoichiometric_reactor.**StoichiometricReactorData**(*component*)

Standard Stoichiometric/Yield reactor Unit Model Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

CSTR 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 2 degrees of freedom.

Typical fixed variables are:

- reactor volume,
- reactor heat duty (has_heat_transfer = True only).

Model Structure

The core CSTR unit model consists of a single Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

CSTR 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, CSTR 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	True
has_equilibrium_reactions	True
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	False

Additional Constraints

CSTR units write the following additional Constraints beyond those written by the Holdup 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 Holdup Block.

Variable	Name	Notes
V_t	volume	If include_holdup = True this is a reference to holdup.volume, otherwise a Var attached to the Unit Model
Q_t	heat	Only if has_heat_transfer = True, reference to holdup.heat

CSTRData Class

class idaes.models.cstr.**CSTRData**(*component*)

Standard CSTR Unit Model Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

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 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 Holdup1D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

PFR 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).
- discretization_method - indicates which method to use when discretizing length domain. Options are:
  - 'OCLR' - orthogonal collocation on finite elements (Radau roots)
  - 'OCLL' - orthogonal collocation on finite elements (Legendre roots)
  - 'BFD' - backwards finite difference (1st order)
  - 'FFD' - forwards finite difference (1st order)
- finite_elements - sets the number of finite elements to use when discretizing the spatial domain (default = 20).
- collocation_points - sets the number of collocation points to use when discretizing the spatial domain (default = 3, collocation methods only).
- has_mass_diffusion - indicates whether mass diffusion terms should be included in the material balance equations (default = False)
- has_energy_diffusion - indicates whether energy conduction terms should be included in the energy balance equations (default = False)

Additionally, PFR 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
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False
include_holdup	False
has_rate_reactions	True
has_equilibrium_reactions	True
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	False
has_work_transfer	False
has_pressure_change	False

Additional Constraints

PFR units write the following additional Constraints beyond those written by the Holdup Block 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 property package).

Variables

PFR units add the following additional Variables beyond those created by the Holdup Block.

Variable	Name	Notes
L	length	Reference to holdup.length
A	area	Reference to holdup.area
V	volume	Reference to holdup.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

PFRData Class

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 Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

Equilibrium reactor 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Equilibrium reactor units have the following construction arguments which are passed to the Holdup Block for determining which terms to construct in the balance equations. Note that Equilibrium reactors do not support dynamic = True as holdups (and thus accumulations) are undefined by definition.

Argument	Default Value
material_balance_type	'component_phase'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False (cannot be True)
include_holdup	False
has_rate_reactions	True
has_equilibrium_reactions	True
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	False

Additional Constraints

Equilibrium reactors units write the following additional Constraints beyond those written by the Holdup Block 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 Holdup Block.

Variable	Name	Notes
V_t	volume	Only if include_holdup = True, reference to holdup.volume
Q_t	heat	Only if has_heat_transfer = True, reference to holdup.heat

EquilibriumReactorData Class

class idaes.models.equilibrium_reactor.**EquilibriumReactorData**(*component*)

Standard Equilibrium Reactor Unit Model Class

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Gibbs Reactor

The IDAES Gibbs reactor model represents a unit operation where a material stream undergoes some set of reactions such that the Gibbs energy of the material 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 1 degree of freedom.

Typical fixed variables are:

- reactor heat duty (has_heat_transfer = True only).

Model Structure

The core Gibbs reactor unit model consists of a single Holdup0D (named holdup) with one Inlet Port (named inlet) and one Outlet Port (named outlet).

Construction Arguments

CSTR 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.
- inlet_list - list of names to be passed to the build_inlets method (default = None).
- num_inlets - number of inlets argument to be passed to the build_inlets method (default = None).
- outlet_list - list of names to be passed to the build_outlets method (default = None).
- num_outlets - number of outlets argument to be passed to the build_outlets method (default = None).

Additionally, Gibbs reactor units have the following construction arguments which are passed to the Holdup Block for determining which terms to construct in the balance equations. Note that Gibbs reactors do not support dynamic = True as holdups (and thus accumulations) are undefined by definition.

Argument	Default Value
material_balance_type	'element_total'
energy_balance_type	'total'
momentum_balance_type	'total'
dynamic	False (cannot be True)
include_holdup	False
has_rate_reactions	True
has_equilibrium_reactions	True
has_phase_equilibrium	False
has_mass_transfer	False
has_heat_transfer	True
has_work_transfer	False
has_pressure_change	False

Additional Constraints

Gibbs reactor models write the following additional constraints to calculate the state that corresponds to the minimum Gibbs energy of the system.

$$0 = g_{pc,t,j} + R \times T_t \times ln(y_{t,j} + \sum_e L_{t,e} \times \alpha_{j,e})$$

where $g_{pc,t,j}$ is the pure component molar Gibbs energy of component j at time t, R is the gas constant, T_t is the final temperature of the material, $y_{t,j}$ is the mole fraction of component j at time :math't' in the final material stream, $L_{t,e}$ 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. $g_{pc,t,j}$, R, $y_{t,j}$ and $\alpha_{j,e}$ all come from the outlet Property Block.

Variables

Gibbs reactor units add the following additional Variables beyond those created by the Holdup Block.

Variable	Name	Notes
$L_{t,e}$	lagrange_mult	
V_t	volume	If include_holdup = True this is a reference to holdup.volume, otherwise a Var attached to the Unit Model
Q_t	heat	Only if has_heat_transfer = True, reference to holdup.heat

GibbsReactorData Class

class idaes.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

post_transform_build()

Continue model construction after DAE transformation.

Parameters **None** –

Returns None

Translator Block

In complex process flowsheets, it may be necessary or desirable to used different property packages in different parts of the flowsheet. In these cases, there will need to be a Block to link the two parts of the flowsheet together, which will need to "translate" between the different property packages. The IDAES Translator block provides a framework to the user for creating this link.

The Translator block allows the user to specify the two property packages to be linked, and constructs a Block containing two Property Blocks and Ports, one for the incoming property package (inlet) and one for the outgoing property package (outlet). Users will then need to provide a set of Constraints linking the necessary states between the incoming and outgoing property packages.

Degrees of Freedom

Degrees of freedom in Translator blocks will depend on the property packages being used. Users will need to read the documentation on the associated property packages to decide on how to link the two.

Model Structure

The core Translator block consists of two Property Blocks (inlet_properties and outlet_properties) and two Port objects (inlet and outlet).

Construction Arguments

Translator blocks have the following construction arguments:

- inlet_property_package - property package to use when constructing the Property Block for the incoming stream. This is provided as a Property Parameter Block by the Flowsheet when creating the model. If a value is not provided, the Translator Block will try to use the default property package if one is defined.
- inlet_property_package_args - set of arguments to be passed to the inlet Property Block when it is created.
- outlet_property_package - property package to use when constructing the Property Block for the outgoing stream. This is provided as a Property Parameter Block by the Flowsheet when creating the model. If a value is not provided, the Translator Block will try to use the default property package if one is defined.
- outlet_property_package_args - set of arguments to be passed to the outlet Property Block when it is created.
- has_equilibrium_reactions - argument to indicate whether the Property Blocks should enforce constraints for chemical equilibrium (default = False).
- has_phase_equilibrium - argument to indicate whether the Property Blocks should enforce constraints for phase equilibrium (default = False).

Additional Constraints

Translator blocks write no additional Constraints. Users should provide the necessary Constraints to link states between the two Property Blocks.

Variables

Translator Blocks create no additional Variables.

TranslatorData Class

class idaes.models.translator.**TranslatorData**(*component*)

This class constructs the basic framework for an IDAES block to "translate" between differnt property packages. This class constructs two property blocks using two different property packages - one for the incoming package and one for the outgoing package - along with the assoicated inlet and outlet Connector objects. Users will then need to provide a set of Constraints to connect the states in the incoming properties to those in the outgoing properties.

build()

Begin building model (pre-DAE transformation).

Parameters **None** –

Returns None

initialize(*outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)
This is a basic initialization routine for translator blocks. This method assumes both property blocks have good initial values and calls the initialization method of each block.

Users will liekly need to overload this method with a customised routine suited to their particular translation.

Keyword Arguments

- **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()
This is a general purpose model_check routine for translator blocks. This method tries to call the model_check method of the inlet and outlet property blocks. If an AttributeError is raised, the check is passed.

Parameters **None** –

Returns None

post_transform_build()
Continue model construction after DAE transformation.

Parameters **None** –

Returns None

3.2.6 Visualization

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.

The IDAES visualization module is built on top of bokeh and is meant to support multiple plot types. The data for these plots is input through dataframes defined via data-frame schemas.

Contents

- *Visualization*
- *The Plot class*
- *Plot utilities*
- *Examples*
  - *Drawing heat exchanger network diagrams*

– *Plotting profile plots from the MEA example*

3.2.7 The Plot class

The plot class implements different IDAES plots. Each of its methods acts like a factory method returning an instance of the Plot that can then be modified by calls to helper methods (e.g: saving a plot to disk). Currently only heat exchanger network diagrams are supported.

class idaes.vis.plot.**Plot**(*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

classmethod goodness_of_fit(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)

Draw y against predicted value (y^) and display (calculate?) value of R^2.

Parameters

- **data_frame** – a data frame with keys contained in x and y.
- **x** – Key in data-frame to use as x-axis.
- **y** – Keys in data-frame to plot on y-axis.
- **title** – Title for a plot.
- **xlab** – Label for x-axis.
- **ylab** – Label for y-axis.
- **y_axis_type** – Specify "log" to pass logarithmic scale.
- **legend** – List of strings matching y.

Returns Plot object on success.

Raises

- MissingVariablesException – Dependent variable or their data not passed.
- BadDataFrameException – No data-frame was generated for the model object.

classmethod heat_exchanger_network(*exchangers*, *stream_list*, *mark_temperatures_with_tooltips=False*, *mark_modules_with_tooltips=False*, *stage_width=2*, *y_stream_step=1*)

Plot a heat exchanger network diagram.

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. The *utility_type*, if present, will specify if we plot 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}}
```

The value of this key is a dictionary where each key is a module area and each value is how many modules of that area are in the exchanger. It's indicated as a tooltip on the resulting diagram.

If a stream is involved in multiple exchanges in teh same stage, the stream will split into multiple sub-streams with each sub-stream carrying one of the exchanges.

- **`stream_list`** – List of dicts representing streams where each item is a dict of the form:

```
{'name':'H1', 'temps': [443, 435, 355, 333], 'type':␣
↪HENStreamType.hot}
```

- **`mark_temperatures_with_tooltips`** – if True, we plot the stream temperatures and assign hover tooltips to them. Otherwise, we label them with text labels.
- **`mark_modules_with_tooltips`** – if True, we plot markers for modules (if present) and assign hover tooltips to them. Otherwise, we don't add module info.
- **`stage_width`** – How many units to use for each stage in the diagram (defaults to 2).
- **`y_stream_step`** – How many units to use to separate each stream/sub-stream from the next (defaults to 1).

`classmethod isobar`(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)

Need more information.

`classmethod profile`(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)

A profile plot includes 2 dependent variables and a single independent variable. Based on the Jupyter notebook here.

Parameters

- **`data_frame`** – a data frame with keys contained in x and y.
- **`x`** – Key in data-frame to use as x-axis.
- **`y`** – Keys in data-frame to use as y-axis.
- **`title`** – Title for a plot.
- **`xlab`** – Label for x-axis.
- **`ylab`** – Label for y-axis.

- **y_axis_type** – Specify "log" to pass logarithmic scale.
- **legend** – List of strings matching y.

Returns Plot object on success.

Raises

- MissingVariablesException – Dependent variable or their data not passed.
- BadDataFrameException – No data-frame was generated for the model object.

classmethod property_model(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)

Draw pressure/enthalpy plots for different levels of temperature.

Parameters

- **data_frame** – a data frame with keys contained in x and y.
- **x** – Key in data-frame to plot on x-axis.
- **y** – Keys in data-frame to plot on y-axis.
- **title** – Title for a plot.
- **xlab** – Label for x-axis.
- **ylab** – Label for y-axis.
- **y_axis_type** – Specify "log" to pass logarithmic scale.
- **legend** – List of strings matching y.

Returns Plot object on success.

Raises

- MissingVariablesException – Dependent variable or their data not passed.
- BadDataFrameException – No data-frame was generated for the model object.

classmethod residual(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)

Plot x, some continuous value (e.g: T, P), against Y (% residual value). Is this %-value calculated from variables in the idaes_model_object?

Parameters

- **data_frame** – a data frame with keys contained in x and y.
- **x** – Key in data-frame to use as x-axis.
- **y** – Keys in data-frame to plot on y-axis.
- **title** – Title for a plot.
- **xlab** – Label for x-axis.
- **ylab** – Label for y-axis.
- **y_axis_type** – Specify "log" to pass logarithmic scale.
- **legend** – List of strings matching y.

Returns Plot object on success.

Raises

- MissingVariablesException – Dependent variable or their data not passed.

- `BadDataFrameException` – No data-frame was generated for the model object.

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`

classmethod sensitivity(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)
Need more information.

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`

classmethod stream_table(*data_frame*, *title=''*)
Display a table for all names in the idaes_model_object_names indexing rows according to row_start and row_stop.

Parameters

- **data_frame** – a data frame with keys contained in x and y.
- **title** – Title for a plot.

Returns Plot object on success.

Raises

- `MissingVariablesException` – Dependent variable or their data not passed.
- `BadDataFrameException` – No data-frame was generated for the model object.

classmethod tradeoff(*data_frame*, *x=''*, *y=[]*, *title=''*, *xlab=''*, *ylab=''*, *y_axis_type='auto'*, *legend=[]*)
Draw some parameter varying and the result on the objective value.

Parameters

- **data_frame** – a data frame with keys contained in x and y.
- **x** – Key in data-frame to use as x-axis.

- **y** – Keys in data-frame to plot on y-axis.
- **title** – Title for a plot.
- **xlab** – Label for x-axis.
- **ylab** – Label for y-axis.
- **y_axis_type** – Specify "log" to pass logarithmic scale.
- **legend** – List of strings matching y.

Returns Plot object on success.

Raises

- `MissingVariablesException` – Dependent variable or their data not passed.
- `BadDataFrameException` – No data-frame was generated for the model object.

3.2.8 Plot utilities

The plot utilities module implements helper/utility methods used for visualization.

class idaes.vis.plot_utils.**HENStreamType**
Enum type defining hot and cold streams

idaes.vis.plot_utils.**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.plot_utils.**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.plot_utils.**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.plot_utils.**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

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.plot_utils.**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.plot_utils.**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.plot_utils.**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.plot_utils.**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

idaes.vis.plot_utils.**validate**(*data_frame*, *x*, *y*, *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) Raises exceptions otherwise.

Raises

- MissingVariablesException - on bad legend labels (if passed)
- BadDataFrameException - on invalid data frame.

3.2.9 Examples

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.

```
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!)

```
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:

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

(continues on next page)

(continued from previous page)

```
    {'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}},
    {'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

Note: 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'],
```

(continues on next page)

(continued from previous page)

```
                                  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')
```

3.2.10 Examples

Contents

- *Examples*
  - *Hello, world!*

The examples are stored in Jupyter notebooks.

Hello, world!

This example was generated by the command: *jupyter nbconvert –to html examples/notebooks/HelloWorldExample.ipynb*

3.2.11 Core API Documentation

Information on specific functions, classes, and methods for the IDAES core framework.

idaes.core package

Subpackages

idaes.core.plugins package

Submodules

idaes.core.plugins.pyosyn module

class idaes.core.plugins.pyosyn.**Pyosyn**(***kwds*)
Bases: pyomo.core.base.plugin.Transformation

Transformation to create a process synthesis model.

idaes.core.util package

Subpackages

idaes.core.util.convergence package

Subpackages

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.PressureChangerConvergenceE

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.**main**()

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 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).

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.

class idaes.core.util.convergence.convergence_base.**ConvergenceEvaluation**

Bases: object

get_initialized_model()

User should override this method to return an initialized model that is ready to solve. The convergence evaluation methods will change the values of parameters or variables according to the sampling specifications.

Returns **Pyomo model** – model object that will be used in the evaluation.

Return type return a Pyomo model object that is initialized and ready to solve. This is the

get_solver()
User should create and return the solver that will be used for the convergence evaluation (including any necessary options)

Returns

Return type Pyomo solver

get_specification()
User should override this method to return an instance of the ConvergenceEvaluationSpecification for this particular model and test set.

The basic flow for this method is:

- Create a ConvergenceEvaluationSpecification
- Call add_sampled_input for every input that should be varied.
- return the ConvergenceEvaluationSpecification

Returns

Return type *ConvergenceEvaluationSpecification*

class idaes.core.util.convergence.convergence_base.**ConvergenceEvaluationSpecification**
Bases: object

add_sampled_input(*name*, *pyomo_path*, *lower*, *upper*, *mean*, *std*)
Add an input that should be sampled when forming the set of specific points that need to be run for the convergence evaluation

The input will be sampled assuming a normal distribution (with given mean and standard devation) truncated to the values given by lower and upper bounds

Parameters

- **name** (*str*) – The name of the input.
- **pyomo_path** (*str*) – A string representation of the path to the variable or parameter to be sampled. This string will be executed to retrieve the Pyomo component.
- **lower** (*float*) – A lower bound on the input variable or parameter.
- **upper** (*float*) – An upper bound on the input variable or parameter
- **mean** (*float*) – The mean value to use when generating samples
- **std** (*float*) – The standard deviation to use when generating samples

Returns

Return type N/A

inputs

class idaes.core.util.convergence.convergence_base.**Stats**(*results*)
Bases: object

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.**run_convergence_evaluation_from_sample_file**(*sa*

idaes.core.util.convergence.convergence_base.**save_convergence_statistics**(*inputs*, *results*, *dmf=None*)

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*, *filename*, *convergence_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

class idaes.core.util.convergence.mpi_utils.**MPIInterface**

Bases: object

comm

have_mpi

rank

size

class idaes.core.util.convergence.mpi_utils.**ParallelTaskManager**(*n_total_tasks*, *mpi_interface=None*)

Bases: object

allgather_global_data(*local_data*)

gather_global_data(*local_data*)

global_to_local_data(*global_data*)

is_root()

Submodules

idaes.core.util.compare module

Provides a utility function for comparing two models

idaes.core.util.compare.**compare**(*m1*, *m2*, ***kwargs*)

The idea here is to go block by block through these two models, printing out the differences in the variables. Some recursion to be expected. If m1name and/or m2name is assigned, the corresponding model will be renamed to the given string

idaes.core.util.compare.**compare_block**(*b1*, *b2*, ***kwargs*)

idaes.core.util.compare.**compare_var**(*v1*, *v2*, *p1names*, *p2names*, ***kwargs*)

idaes.core.util.concave module

Utility functions for implementing piecewise linear underestimators for concave univariate expressions.

Implementation of the delta formulation from: Bergamini, M. L., Grossmann, I., Scenna, N., & Aguirre, P. (2008).

> An improved piecewise outer-approximation algorithm for the global optimization of MINLP models involving concave and bilinear terms. Computers and Chemical Engineering, 32, 477–493. http://doi.org/10.1016/j.compchemeng.2007.03.011

idaes.core.util.concave.**add_concave_linear_underest**(*b*, *name*, *nsegs*, *x*, *f*, *f_expr*, **sets*, ***kwargs*)

idaes.core.util.concave.**add_concave_relaxation**(*b*, *z*, *x*, *f_expr*, *df_expr*, *nsegs*, *indx*, *exists*, *block_bounds=(None*, *None)*, *bound_contract=None*)

idaes.core.util.concave.**try_eval**(*f*, *x*)

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_parameter_block**(*val*)
Domain validator for property package attributes

Parameters **val** – value to be checked

Returns TypeError if val is not an instance of PropertyParameterBase, 'use_parameter_block' or None

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.**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.convex module

Utility functions for generating envelopes for convex nonlinear functions

idaes.core.util.convex.**add_convex_relaxation**(*b*, *z*, *x*, *f_expr*, *df_expr*, *nsegs*, *indx*, *exists*, *block_bounds=(None*, *None)*, *bound_contract=None*)
Constructs a linear relaxation to bound a convex equality function

Parameters

- **b** (*Block*) – PyOMO block in which to generate variables and constraints
- **z** (*Expression*) – PyOMO expression for the convex function output
- **x** (*Expression*) – PyOMO expression for the convex function input
- **f_expr** (*function*) – convex function
- **df_expr** (*function*) – function giving first derivative of convex function with respect to x
- **exists** (*_VarData*) – Variable corresponding to existence of unit
- **block_bounds** (*dict, optional*) – dictionary describing disjunctive bounds present for variables associated with current block

Returns None

idaes.core.util.convex.**try_eval**(*f*, *x*)

idaes.core.util.cut_gen module

This module provides utility functions for cut generation.

idaes.core.util.cut_gen.**clone_block**(*old_block*, *new_block*, *var_set*, *lbda*)
This function acts similarly to the built-in Pyomo clone function, but excludes entries that are undesired for this platform.

idaes.core.util.cut_gen.**clone_block_constraints**(*old_block*, *new_block*, *var_set*)

idaes.core.util.cut_gen.**clone_block_params**(*old_block*, *new_block*)

idaes.core.util.cut_gen.**clone_block_sets**(*old_block*, *new_block*)

idaes.core.util.cut_gen.**clone_block_vars**(*old_block*, *new_block*, *var_set*, *lbda*)

idaes.core.util.cut_gen.**copy_var_data**(*new_var_data*, *old_var_data*)

idaes.core.util.cut_gen.**count_vars**(*old_block*)
Count the number of non-fixed variables to clone

idaes.core.util.cut_gen.**get_sum_sq_diff**(*old_block*, *new_block*)

idaes.core.util.cut_gen.**self_proj_var_rule**(*main_var*, *disagg_var*, *var_set*, *lbda*)

idaes.core.util.debug module

Debugging utility functions.

idaes.core.util.debug.**display_infeasible_bounds**(*m*, *tol=1e-06*)
Print the infeasible variable bounds in the model.

Parameters

- **m** (*Block*) – Pyomo block or model to check
- **tol** (*float*) – feasibility tolerance

idaes.core.util.debug.**display_infeasible_constraints**(*m*, *tol=1e-06*)
Print the infeasible constraints in the model.

Uses the current model state. Prints to standard out.

Parameters

- **m** (*Block*) – Pyomo block or model to check
- **tol** (*float*) – feasibility tolerance

idaes.core.util.debug.**log_disjunct_values**(*m*, *integer_tolerance=0.001*, *logger=<Logger idaes.debug (INFO)>*, *selected_only=False*)

Display logical value of model disjuncts.

Parameters

- **integer_tolerance** (*float*) – tolerance on integrality test.
- **logger** – logger to use for output. Otherwise, default debug module logger is used.
- **selected_only** – only log the selected disjuncts

idaes.core.util.expr module

Utility functions for working with Pyomo expressions.

idaes.core.util.expr.**is_linear**(*expr*)

Check if the Pyomo expression is linear.

TODO: There is the possibility for false negatives (if there is a nonlinear expression of a mutable parameter). Need to test this.

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

Returns True if expression is linear; False otherwise.

Return type bool

Raises ValueError – if polynomial degree is negative

idaes.core.util.expr.**log_active_nonlinear_constraints**(*model*, *logger=<RootLogger root (WARNING)>*)

Log names of the active nonlinear constraints.

idaes.core.util.initialization module

Library of utility functions for initialization

idaes.core.util.initialization.**get_port_value**(*port*, *port_idx=None*)

idaes.core.util.mccormick module

Utility functions for implementing piecewise (or standard) McCormick envelopes

Implementation of the McCormick envelope formulation given in: Misener, R., Thompson, J. P., & Floudas, C. A. (2011). Apogee:

> Global optimization of standard, generalized, and extended pooling problems via linear and logarithmic partitioning schemes. Computers and Chemical Engineering, 35(5), 876–892. http://doi.org/10.1016/j.compchemeng.2011.01.026

idaes.core.util.mccormick.**add_mccormick_cut**(*b*, *name*, *indx*, *z*, *x*, *y*, *exists*, ***kwargs*)

idaes.core.util.mccormick.**add_mccormick_relaxation**(*b*, *z*, *x*, *y*, *nsegs*, *indx*, *exists*, *block_bounds=(None, None)*)

Adds McCormick envelopes for a bilinear term z = x * y

Parameters

- **b** (*Block*) – PyOMO block in which to put constraints and helper variables
- **z** (*Expression*) – PyOMO expression for the bilinear product
- **x** (*Expression*) – expression for the bilinear operand to be divided into segments for the piecewise case
- **y** (*Expression*) – expression for the other bilinear operand
- **nsegs** (*integer*) – number of piecewise segments (normal is 1)
- **indx** (*tuple or singleton*) – index for an indexed bilinear term
- **exists** (*Var*) – variable corresponding to equipment existence
- **block_bounds** (*dict, optional*) – dictionary describing disjunctive bounds present for variables associated with current block

Returns None

idaes.core.util.mccormick.**setup_mccormick_cuts**(*b*, *name*, *nsegs*, **sets*)

idaes.core.util.mccormick.**squish**(*tup*)

Squishes a singleton tuple ('A',) to 'A'

If tup is a singleton tuple, return the underlying singleton. Otherwise, return the original tuple.

Parameters **tup** (*tuple*) – Tuple to squish

idaes.core.util.mccormick.**squish_concat**(*a*, **b*)

idaes.core.util.misc module

This module contains miscellaneous utility functions that of general use in IDAES models.

idaes.core.util.misc.**add_object_ref**(*local_block*, *local_name*, *external_component*)

Add a reference in a model to non-local Pyomo component. This is used when one Block needs to make use of a component in another Block as if it were part of the local block.

Parameters

- **local_block** – Block in which to add reference
- **local_name** – str name for referenced object to use in local_block
- **external_component** – external component being referenced

Returns None

idaes.core.util.misc.**category**(**args*)

Decorate tests to enable tiered testing.

Suggested categories:

1. frequent
2. nightly
3. expensive
4. research

Parameters ***args** (*tuple of strings*) – categories to which the test belongs

Returns Either the original test function or skip

Return type function

idaes.core.util.misc.**dict_set**(*v*, *d*, *pre_idx=None*, *post_idx=None*, *fix=False*)
Set the values of array variables based on the values stored in a dictionary. There may already be a better way to do this. Should look into it.

The value of Pyomo variable element with index key is set to d[key]

Arguments: v: Indexed Pyomo variable d: dictonary to set the variable values from, keys should match a subset

of Pyomo variable indexes.

pre_idx: fixed indexes before elements to be set or None post_idx: fixed indexes after elements to be set or None fix: bool, fix the variables (otional)

idaes.core.util.misc.**doNothing**(**args*, ***kwargs*)
Do nothing.

This function is useful for instances when you want to call a function, if it exists. For example: getattr(unit, 'possibly_defined_function', getNothing)()

Parameters

- ***args** (*anything*) – accepts any argument
- ****kwargs** (*anything*) – accepts any keyword arguments

Returns None

idaes.core.util.misc.**fix_port**(*port*, *var*, *comp=None*, *value=None*, *port_idx=None*)
Method for fixing Vars in Ports.

Parameters

- **port** – Port object in which to fix Vars
- **var** – variable name to be fixed (as str)
- **comp** – index of var to be fixed (if applicable, default = None)
- **value** – value to use when fixing var (default = None)
- **port_idx** – list of Port elements at which to fix var. Must be list of valid indices,

Returns None

idaes.core.util.misc.**get_pyomo_tmp_files**()
Make Pyomo write it's temporary files to the current working directory, useful for checking nl, sol, and log files for ASL solvers without needing to track down the temporary file location.

idaes.core.util.misc.**get_time**(*results*)
Retrieve the solver-reported elapsed time, if available.

idaes.core.util.misc.**hhmmss**(*sec_in*)
Convert elapsed time in seconds to "d days hh:mm:ss.ss" format. This is nice for things that take a long time.

idaes.core.util.misc.**requires_solver**(*solver*)
Decorate test to skip if a solver isn't available.

idaes.core.util.misc.**round_**(*n*, **args*, ***kwargs*)
Round the number.

This function duplicates the functionality of round, but when passed positive or negative infinity, simply returns the argument.

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

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.misc.**smooth_max**(*a*, *b*, *eps=0.0001*)
Smooth maximum operator.

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.misc.**smooth_min**(*a*, *b*, *eps=0.0001*)
Smooth minimum operator.

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.misc.**smooth_minmax**(*a*, *b*, *eps=0.0001*, *sense='max'*)
General function for creating an expression for a smooth minimum or maximum.

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.**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

- **solve** – 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.misc.**unfix_port**(*port*, *var*, *comp=None*, *port_idx=None*)
Method for unfixing Vars in Ports.

Parameters

- **port** – Port object in which to unfix Vars
- **var** – variable name to be unfixed (as str)
- **comp** – index of var to be unfixed (if applicable, default = None)
- **port_idx** – list of Port elements at which to unfix var. Must be list of valid indices,

Returns None

idaes.core.util.model_serializer module

Functions for saving and loading Pyomo objects to json

class idaes.core.util.model_serializer.**Counter**
Bases: object

class idaes.core.util.model_serializer.**StoreSpec**(*classes=((<class 'pyomo.core.base.param.Param'>, ()), (<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 '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*)
Bases: object

A StoreSpec object tells the load/save JSON 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.

Initialize an object to specify what parts of a model are saved. Classes and data classes are checked in order. So the more specific classes should go first and fallback cases should go last. Since classes like component catch pretty muh everything, you can also specify classes to skip by adding classes to the skip class list. Classes and data classes can also be skipped by setting there attribute list to None.

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 an empty list is

okay. If none skip this 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.

- **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.
- **ignore_missing** – If True will ignore a componet 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 saced state. Extra items in the saved state will not raise an exception regaurdless of this arguments
- **suffix** – If True store suffixes an component ids. If false, don't stored suffixes, also don't store the compoent ids because they are only needed to read back 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 is 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 suffix(*suffix_filter=None*)

classmethod value()
Returns a StoreSpec object to store variable values only.

classmethod value_isfixed(*only_fixed*)
Retur a StoreSpec object to store variable values and if fixed. :param 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. :param 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=False*)
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, if None, check for s
- **s** – JSON string to load
- **wts** – StoreSpec object specifying what to load

Returns

- 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

Return type Dictionary with some perfomance information. The keys are

idaes.core.util.model_serializer.**load_json**(**args*, ***kwargs*)
Deprecated function will add warning soon. See from_json.

idaes.core.util.model_serializer.**save_json**(**args*, ***kwargs*)
Deprecated function will add warning eventually. See to_json.

idaes.core.util.model_serializer.**to_json**(*o*, *fname=None*, *human_read=False*, *wts=None*, *metadata={}*, *gz=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

- **fname** – json file name to save model state, if None only create python dict
- **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
- **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.

Returns A Python dictionary containing the state of the model. If fname is given this dictionary is also dumped to a json file.

idaes.core.util.mpdisagg module

Utility functions for implementing multi-parametric disaggregation lower bounds

Implementation of lower-bounding formulation given in: Kolodziej, S., Castro, P. M., & Grossmann, I. E. (2013). Global

optimization of bilinear programs with a multiparametric disaggregation technique. Journal of Global Optimization, 57(4), 1039–1063. http://doi.org/10.1007/s10898-012-0022-1

idaes.core.util.mpdisagg.**add_mpDisagg_cut**(*b*, *name*, *indx*, *w*, *x*, *y*, *exists*)

idaes.core.util.mpdisagg.**setup_multiparametric_disagg**(*b*, *name*, *minPow*, *maxPow*, **sets*)

idaes.core.util.oa module

idaes.core.util.partial_surrogate module

idaes.core.util.stream module

Stream Utilities

idaes.core.util.stream.**make_stream_table**(*model*, *attrs*, *t=0*, *idx={}*, *streams=None*, *ignore_missing=False*)

This function makes a stream table contained in a Pandas data frame.

Parameters

- **model** – Pyomo block to search for streams in.
- **attrs** – Pyomo port components to add to stream table
- **t** – Time index for the stream table data
- **idx** – Dictionary of index lists for indexed attributes
- **streams** – Explicit list of steam objects to include. If this is None all streams in model will be included
- **ignore_missing** – If this is True and streams do not have all the attributes requested, the steam will be included anyway with missing data. If this option is false, streams that do not contain all required information will be excluded.

Returns A pandas data frame with port variables in columns and streams in rows.

Examples

table = make_stream_table(model=model, attrs=["flow_component", "temperature", "pressure"], idx={"flow_component":["CO2", "H2O", "N2", "O2", "Ar"]}, t=0, ignore_missing=False)

idaes.core.util.var module

This module provides utility functions and classes related to Pyomo variables

class idaes.core.util.var.**SliceVar**(*var*, *slices*)

Bases: object

This class provides a way to pass sliced variables into other utility functions

class idaes.core.util.var.**Wrapper**(*obj*)
Bases: object

This class provides a wrapper around a Pyomo Component so that a Block does not try to attach and construct it.

TODO: this might be a good place for a weakref implementation

idaes.core.util.var.**is_fixed_by_bounds**(*expr*, *block_bounds=(None*, *None)*, *tol=1e-14*)

idaes.core.util.var.**lb**(*expr*, *block_bounds=(None*, *None)*)
Returns the lower bound of the expression or variable

idaes.core.util.var.**max_ub**(*expr1*, *expr2*)

idaes.core.util.var.**max_ubb**(*expr1*, *expr2*, *block_bounds*)

idaes.core.util.var.**min_lb**(*expr1*, *expr2*)

idaes.core.util.var.**min_lbb**(*expr1*, *expr2*, *block_bounds*)

idaes.core.util.var.**none_if_empty**(*tup*)
Returns None if passed an empty tuple This is helpful since a SimpleVar is actually an IndexedVar with a single index of None rather than the more intuitive empty tuple.

idaes.core.util.var.**tighten_block_bound**(*var*, *newbound*, *block_bounds*)

idaes.core.util.var.**tighten_mc_var**(*vardata*, *x_expr*, *y_expr*, *block_bounds*)

idaes.core.util.var.**tighten_var_bound**(*var*, *newbound*)
Tightens the variable bounds for one variable

This function does not blindly apply the bounds passed in. Rather, it evaluates whether the new proposed bounds are better than the existing variable bounds.

Parameters

- **var** (*_VarData*) – single Pyomo variable object (not indexed like Var)
- **newbound** (*tuple*) – a tuple of (new lower bound, new upper bound)

Returns None

idaes.core.util.var.**ub**(*expr*, *block_bounds=(None*, *None)*)
Returns the upper bound of the expression or variable

idaes.core.util.var.**unwrap**(*obj*)
Unwraps the wrapper, if one exists.

idaes.core.util.var.**wrap_var**(*obj*)
Provides a Wrapper around the obj if it is not a constant value; otherwise, returns the constant value.

idaes.core.util.var_test module

idaes.core.util.var_test.**assert_var_equal**(*test_case*, *var*, *expected_val*, *tolerance*)

idaes.core.util.var_test.**value_correct**(*var*, *expected_val*, *tolerance*)

Submodules

idaes.core.flowsheet_model module

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

idaes.core.holdup module

Core IDAES holdup models.

class idaes.core.holdup.**Holdup**(**args*, ***kwargs*)
Bases: *idaes.core.process_block.ProcessBlock*

Holdup is a specialized Pyomo block for IDAES holdup blocks, and contains instances of HoldupData. In most cases, users will want to use one of the derived holdup classes (Holdup0D, Holdup1D or HoldupStatic for their models, however this class can be used to create the framework of a holdup block to which the user can then add custom constraints.

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model}
- **include_holdup** – Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = True), **True** - construct holdup terms, **False** - do not construct holdup terms}
- **material_balance_type** – Indicates what type of mass balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'component'), **'none'** - exclude material balances, **'component_phase' - use phase component balances, **'component_total' - use total component balances, **'element_total' - use total element balances.}
- **energy_balance_type** – Indicates what type of energy balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude energy balances, **'total'** - single energy balance for all phases.}
- **momentum_balance_type** – Indicates what type of momentum balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude momentum balances, **'total'** - single momentum balance for all phases.}
- **has_rate_reactions** – Indicates whether terms for rate controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include kinetic reaction terms, **False** - exclude kinetic reaction terms.}
- **has_equilibrium_reactions** – Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** {

'use_parent_value' - get flag from parent (default = False), **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}

- **has_phase_equilibrium** – Indicates whether terms for phase equilibrium should be

constructed (default = 'use_parent_value')

- 'use_parent_value' - get flag from parent (default = False)
- True - include phase equilibrium terms
- False - exclude phase equilibrium terms

has_mass_transfer: Indicates whether terms for mass transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include mass transfer terms, **False** - exclude mass transfer terms.}

has_heat_transfer: Indicates whether terms for heat transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_work_transfer: Indicates whether terms for work transfer should be constructed, default - 'use_parent_value'. **Valid values** { **'use_parent_value'** - get flag from parent (default = False), **True** - include work transfer terms, **False** - exclude work transfer terms.}

has_pressure_change: Indicates whether terms for pressure change should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package: Property parameter object used to define property calculations, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **a ParameterBlock object**.}

property_package_args: A dict of arguments to be passed to a property block and used when constructing these, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **dict** - see property package for documentation.}

Returns New Holdup instance

class idaes.core.holdup.**Holdup0D**(**args*, ***kwargs*)

Bases: *idaes.core.process_block.ProcessBlock*

Holdup0D is a specialized Pyomo block for IDAES non-discretized holdup blocks, and contains instances of Holdup0dData.

Holdup0D should be used for any holdup 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** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model}

- **include_holdup** – Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = True), **True** - construct holdup terms, **False** - do not construct holdup terms}
- **material_balance_type** – Indicates what type of mass balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'component'), **'none'** - exclude material balances, **'component_phase' - use phase component balances, **'component_total' - use total component balances, **'element_total' - use total element balances.}
- **energy_balance_type** – Indicates what type of energy balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude energy balances, **'total'** - single energy balance for all phases.}
- **momentum_balance_type** – Indicates what type of momentum balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude momentum balances, **'total'** - single momentum balance for all phases.}
- **has_rate_reactions** – Indicates whether terms for rate controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include kinetic reaction terms, **False** - exclude kinetic reaction terms.}
- **has_equilibrium_reactions** – Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}
- **has_phase_equilibrium** – Indicates whether terms for phase equilibrium should be

constructed (default = 'use_parent_value')

- 'use_parent_value' - get flag from parent (default = False)
- True - include phase equilibrium terms
- False - exclude phase equilibrium terms

has_mass_transfer: Indicates whether terms for mass transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include mass transfer terms, **False** - exclude mass transfer terms.}

has_heat_transfer: Indicates whether terms for heat transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_work_transfer: Indicates whether terms for work transfer should be constructed, default - 'use_parent_value'. **Valid values** { **'use_parent_value'** - get flag from parent (default = False), **True** - include work transfer terms, **False** - exclude work transfer terms.}

has_pressure_change: Indicates whether terms for pressure change should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package: Property parameter object used to define property calculations, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **a ParameterBlock object**.}

property_package_args: A dict of arguments to be passed to a property block and used when constructing these, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **dict** - see property package for documentation.}

Returns New Holdup0D instance

class idaes.core.holdup.**Holdup0dData**(*component*)

Bases: *idaes.core.holdup.HoldupData*

0-Dimensional (Non-Discretised) Holdup Class

This class forms the core of all non-discretized IDAES models. It builds property blocks and adds mass, energy and momentum balances. The form of the terms used in these constraints is specified in the chosen property package.

build()

Build method for Holdup0D blocks. This method calls submethods to setup the necessary property blocks, distributed variables, material, energy and momentum balances based on the arguments provided by the user.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)

Initialisation routine for holdup (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 initialisation routine. **Valid values: 0** - no output (default), **1** - return solver state for each step in routine, **2** - include solver output infomation (tee=True)
- **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 varaibles 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 relase_state method.

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()

This method exectues the model_check methods on the associated property blocks (if they exist). This method is generally called by a unit model as part of the unit's model_check method.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)

Method to relase state variables fixed during initialisation.

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 of logging

Returns None

class idaes.core.holdup.**Holdup1D**(**args*, ***kwargs*)

Bases: *idaes.core.process_block.ProcessBlock*

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model}
- **include_holdup** – Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = True), **True** - construct holdup terms, **False** - do not construct holdup terms}
- **material_balance_type** – Indicates what type of mass balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'component'), **'none'** - exclude material balances, **'component_phase' - use phase component balances, **'component_total' - use total component balances, **'element_total' - use total element balances.}
- **energy_balance_type** – Indicates what type of energy balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude energy balances, **'total'** - single energy balance for all phases.}
- **momentum_balance_type** – Indicates what type of momentum balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude momentum balances, **'total'** - single momentum balance for all phases.}
- **has_rate_reactions** – Indicates whether terms for rate controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include kinetic reaction terms, **False** - exclude kinetic reaction terms.}
- **has_equilibrium_reactions** – Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}
- **has_phase_equilibrium** – Indicates whether terms for phase equilibrium should be

constructed (default = 'use_parent_value')

- 'use_parent_value' - get flag from parent (default = False)
- True - include phase equilibrium terms
- False - exclude phase equilibrium terms

has_mass_transfer: Indicates whether terms for mass transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include mass transfer terms, **False** - exclude mass transfer terms.}

has_heat_transfer: Indicates whether terms for heat transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_work_transfer: Indicates whether terms for work transfer should be constructed, default - 'use_parent_value'. **Valid values** { **'use_parent_value'** - get flag from parent (default = False), **True** - include work transfer terms, **False** - exclude work transfer terms.}

has_pressure_change: Indicates whether terms for pressure change should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package: Property parameter object used to define property calculations, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **a ParameterBlock object**.}

property_package_args: A dict of arguments to be passed to a property block and used when constructing these, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **dict** - see property package for documentation.}

inherited_length_domain: A Pyomo ContinuousSet to use as the length domain in the holdup. length_domain_set: List of floats between 0 and 1 to used to initialize length domain, if

inherited_length_domain not set (default = [0.0, 1.0]

flow_direction: Flag indicating the direction of flow within the length

domain (default = ***forward*****).**

- 'forward' - flow from 0 to 1
- 'backward' - flow from 1 to 0

discretization_method: Method to be used by DAE transformation when discretizing length

domain (default = ***OCLR*****).**

- 'OCLR' - orthogonal collocation (Radau roots)
- 'OCLL' - orthogonal collocation (Legendre roots)
- 'BFD' - backwards finite difference (1st order)
- 'FFD' - forwards finite difference (1st order)

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)

has_mass_diffusion: Flag indicating whether mass diffusion/dispersion should be included in material balance equations (default=False)

has_energy_diffusion: Flag indicating whether energy diffusion/dispersion should be included in energy balance equations (default=False)

Returns New Holdup1D instance

class idaes.core.holdup.**Holdup1dData**(*component*)
Bases: *idaes.core.holdup.HoldupData*

1-Dimensional Holdup Class

This class is designed to be the core of all 1D discretized IDAES models. It builds property blocks, inlet/outlet ports and adds mass, energy and momentum balances. The form of the terms used in these constraints is specified in the chosen property package.

Assumes constant reactor dimensions

CONFIG = <pyutilib.misc.config.ConfigBlock object>

build()
Build method for Holdup1D blocks. This method calls submethods to setup the necessary property blocks, distributed variables, material, energy and momentum balances based on the arguments provided by the user.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *hold_state=True*, *solver='ipopt'*, *optarg=None*)
Initialisation routine for holdup (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 initialisation routine
  - 0 = no output
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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). - True - states varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()
This method exectues the model_check methods on the associated property blocks (if they exist). This method is generally called by a unit model as part of the unit's model_check method.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging

Returns None

class idaes.core.holdup.**HoldupData**(*component*)
Bases: *idaes.core.process_base.ProcessBlockData*

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

The most signfiicant role of the Holdup class is to set up the build arguments for the holdup block, automatically link to the time domain of the parent block, and to get the information about the property package.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

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

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

Parameters **None** –

Returns None

get_property_package()
This method gathers the necessary information about the property package to be used in the holdup block.

If a property package has not been provided by the user, the method searches up the model tree until it finds an object with the 'default_property_package' attribute and uses this package for the holdup block.

The method also gathers any default construction arguments specified for the property package and combines these with any arguments specified by the user for the holdup block (user specified arguments take priority over defaults).

Parameters **None** –

Returns None

class idaes.core.holdup.**HoldupStatic**(**args*, ***kwargs*)
Bases: *idaes.core.process_block.ProcessBlock*

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent, **True** - set as a dynamic model, **False** - set as a steady-state model}
- **include_holdup** – Indicates whether holdup terms should be constructed or not. Must be True if dynamic = True, **default** - 'use_parent_value'. **Valid values:** {

'use_parent_value' - get flag from parent (default = True), **True** - construct holdup terms, **False** - do not construct holdup terms}

- **material_balance_type** – Indicates what type of mass balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'component'), **'none'** - exclude material balances, **'component_phase' - use phase component balances, **'component_total' - use total component balances, **'element_total' - use total element balances.}
- **energy_balance_type** – Indicates what type of energy balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude energy balances, **'total'** - single energy balance for all phases.}
- **momentum_balance_type** – Indicates what type of momentum balance should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = 'total'), **'none'** - exclude momentum balances, **'total'** - single momentum balance for all phases.}
- **has_rate_reactions** – Indicates whether terms for rate controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include kinetic reaction terms, **False** - exclude kinetic reaction terms.}
- **has_equilibrium_reactions** – Indicates whether terms for equilibrium controlled reactions should be constructed, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include equilibrium reaction terms, **False** - exclude equilibrium reaction terms.}
- **has_phase_equilibrium** – Indicates whether terms for phase equilibrium should be

constructed (default = 'use_parent_value')

- 'use_parent_value' - get flag from parent (default = False)
- True - include phase equilibrium terms
- False - exclude phase equilibrium terms

has_mass_transfer: Indicates whether terms for mass transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include mass transfer terms, **False** - exclude mass transfer terms.}

has_heat_transfer: Indicates whether terms for heat transfer should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include heat transfer terms, **False** - exclude heat transfer terms.}

has_work_transfer: Indicates whether terms for work transfer should be constructed, default - 'use_parent_value'. **Valid values** { **'use_parent_value'** - get flag from parent (default = False), **True** - include work transfer terms, **False** - exclude work transfer terms.}

has_pressure_change: Indicates whether terms for pressure change should be constructed, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get flag from parent (default = False), **True** - include pressure change terms, **False** - exclude pressure change terms.}

property_package: Property parameter object used to define property calculations, default - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **a ParameterBlock object**.}

property_package_args: A dict of arguments to be passed to a property block and used when constructing these, **default** - 'use_parent_value'. **Valid values:** { **'use_parent_value'** - get package from parent (default = None), **dict** - see property package for documentation.}

Returns New HoldupStatic instance

class idaes.core.holdup.**HoldupStaticData**(*component*)

Bases: *idaes.core.holdup.HoldupData*

Static Holdup Class

This class is designed to be used for unit operations zero volume or holdups with no through flow (such as dead zones). This type of holdup has only a single PropertyBlock index by time (Holdup0D has two).

build()

Build method for HoldupStatic blocks. This method calls submethods to setup the necessary property blocks, distributed variables, material, energy and momentum balances based on the arguments provided by the user.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)

Initialisation routine for holdup (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 initialisation routine
  - 0 = no output (default)
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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). - True - states varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()

This method exectues the model_check methods on the associated property blocks (if they exist). This method is generally called by a unit model as part of the unit's model_check method.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging

Returns None

idaes.core.ports module

These classes handle the mixing and splitting of multiple inlets/outlets to a single holdup block.

class idaes.core.ports.**InletMixer**(**args*, ***kwargs*)
Bases: *idaes.core.process_block.ProcessBlock*

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **has_material_balance** – Indicates whether material mixing constraints should be constructed.
- **has_energy_balance** – Indicates whether energy mixing constraints should be constructed.
- **has_momentum_balance** – Indicates whether momentum mixing constraints should be constructed.
- **inlets** – A list of strings to be used to name inlet streams.

Returns New InletMixer instance

class idaes.core.ports.**InletMixerData**(*component*)
Bases: *idaes.core.ports.Port*

Inlet Mixer Class

This class builds a mixer to allow for multiple inlets to a single holdup block. The class constructs property blocks for each inlet and creates mixing rules to connect them to the property block within the associated holdup block.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

build()
Build method for Mixer blocks. This method calls a number of methods to construct the necessary balance equations for the Mixer.

Parameters **None** –

Returns None

initialize(*state_args=None*, *outlvl=0*, *optarg=None*, *solver='ipopt'*, *hold_state=True*)
Initialisation routine for InletMixer (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 initialisation routine
  - 0 = no output (default)
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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).

  - **True = state varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()
Calls model checks on all associated Property Blocks.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging (default=0)

Returns None

class idaes.core.ports.**OutletSplitter**(**args*, ***kwargs*)
Bases: *idaes.core.process_block.ProcessBlock*

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **has_material_balance** – Indicates whether material mixing constraints should be constructed.
- **has_energy_balance** – Indicates whether energy mixing constraints should be constructed.

- **has_momentum_balance** – Indicates whether momentum mixing constraints should be constructed.
- **outlets** – A list of strings to be used to name outlet streams.
- **split_type** – A list of strings to be used to name outlet streams.

Returns New OutletSplitter instance

class idaes.core.ports.**OutletSplitterData**(*component*)

Bases: *idaes.core.ports.Port*

Outlet Mixer Class

This class builds a splitter to allow for multiple outlets to a single holdup block. The class constructs property blocks for each outlet and creates splitting rules to connect them to the property block within the associated holdup block.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

build()

Build method for Splitter blocks. This method calls a number of methods to construct the necessary balance equations for the Splitter.

Parameters **None** –

Returns None

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

Initialisation routine for OutletSplitter (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 initialisation routine
  - 0 = no output (default)
  - 1 = return solver state for each step in routine
  - 2 = include solver output infomation (tee=True)
- **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).

  - **True = state varaibles 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 relase_state method

Returns If hold_states is True, returns a dict containing flags for which states were fixed during initialization.

model_check()

Calls model checks on all associated Property Blocks.

Parameters **None** –

Returns None

release_state(*flags*, *outlvl=0*)
Method to relase state variables fixed during initialisation.

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 of logging (default=0)

class idaes.core.ports.**Port**(*component*)
Bases: *idaes.core.process_base.ProcessBlockData*

Base Port Class

This class contains methods common to all Port classes.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

build()
General build method for Ports. This method calls a number of methods common to all Port blocks.

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

Parameters **None** –

Returns None

idaes.core.process_base module

Base for IDAES process model objects.

class idaes.core.process_base.**ProcessBlockData**(*component*)
Bases: pyomo.core.base.block._BlockData

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.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

build()
Default build method for all Classes inheriting from ProcessBlockData. Currently empty, but left in place to allow calls to super().build and for future compatability.

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

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 to create modular process model blocks.

class idaes.core.process_block.**ProcessBlock**(**args*, ***kwargs*)
Bases: pyomo.core.base.block.Block

Process block.

Process block behaves like a Pyomo Block. The important differences are listed below.

- There is a default rule that calls the build() method for _BlockData subclass ojects, so subclass of _BlockData used in a ProcessBlock should have a build() method. A different rule or no rule (None) can be set with the usual rule argument, if additional steps are required to build an element of a block. A example of such a case is where different elements of an indexed block require addtional information to construct.
- Some of the arguments to __init__, which are not expected arguments of Block, are split off and stored in self._block_data_config. If the _BlockData subclass inherits ProcessBlockData, self._block_data_config is sent to the self.config ConfigBlock.

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 _BlockData. It creates a ProcessBlock subclass to contain it. For example (where ProcessBlockData is a subclass of _BlockData):

@declare_process_block_class(name=MyUnitBlock) class MyUnitBlockData(ProcessBlockData):

This class is a _BlockData subclass contained in a Block subclass # MyUnitBlock ....

The only requirment is that the subclass of _BlockData contain a build() method.

Parameters

- **name** – class name for the model.

- **block_class** – ProcessBlock or a subclass of ProcessBlock, this allows you to use a subclass of ProcessBlock if needed.
- **doc** – Documentation for the class. This should play nice with sphinx.

idaes.core.property_base module

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

class idaes.core.property_base.**PropertyBlockDataBase**(*component*)
Bases: *idaes.core.process_base.ProcessBlockData*

This is the base class for property block data objects. These are blocks that contain the Pyomo components associated with calculating a set of thermophysical, transport and reacion properties for a given material.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

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

Parameters **None** –

Returns None

class idaes.core.property_base.**PropertyParameterBase**(*component*)
Bases: *idaes.core.process_base.ProcessBlockData*

This is the base class for property parameter blocks. These are blocks that contain a set of parameters associated with a specific property package, and are linked to by all instances of that property package.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

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

Parameters **None** –

Returns None

get_package_units()
Method to return a dictionary of default units of measurement used in the property package. This is used to populate doc strings for variables which derive from the property package (such as flows and volumes). This method should return a dict with keys for the quantities used in the property package (as strs) and values of their default units as strs.

The quantities used by the framewokr are (all optional):

- 'time'
- 'length'
- 'mass'
- 'amount'
- 'temperature'
- 'energy'
- 'current'
- 'luminous intensity'

This default method is a placeholder and should be overloaded by the package developer. This method will return an Exception if not overloaded.

Parameters **None** –

Returns A dict with supported properties as keys and tuples of (method, units) as values.

get_supported_properties()
Method to return a dictionary of properties supported by this package and their assoicated construction methods and units of measurement. This method should return a dict with keys for each supported property.

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.

This default method is a placeholder and should be overloaded by the package developer. This method will return an Exception if not overloaded.

Parameters **None** –

Returns A dict with supported properties as keys.

idaes.core.stream module

Base clase for IDAES streams

class idaes.core.stream.**Stream**(**args*, ***kwargs*)
Bases: *idaes.core.process_block.ProcessBlock*

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **source** – Pyomo Port object representing the source of the process stream.
- **source_idx** – Key of indexed Port to use for source (if applicable). default = None.
- **destination** – Pyomo Port object representing the destination of the process stream.
- **destination_idx** – Key of indexed Port to use for destination (if applicable). default = None.

Returns New Stream instance

class idaes.core.stream.**StreamData**(*component*)
Bases: *idaes.core.process_base.ProcessBlockData*

This is the class for process streams. These are blocks that connect two unit models together.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

activate(*var=None*)
Method for activating Constraints in Stream. If not provided with any arguments, this activates the entire Stream block. Alternatively, it may be provided with the name of a variable in the Stream, in which case only the Constraint associated with that variable will be activated.

Parameters **var** – name of a variable in the Stream for which the corresponding Constraint should be activated (default = None).

Returns None

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

Parameters **None** –

Returns None

converged(*tolerance=1e-06*)
Check if the values on both sides of a Stream are converged.

Parameters **tolerance** – tolerance to use when checking if Stream is converged. (default = 1e-6).

Returns A Bool indicating whether the Stream is converged

deactivate(*var=None*)
Method for deactivating Constraints in Stream. If not provided with any arguments, this deactivates the entire Stream block. Alternatively, it may be provided with the name of a variable in the Stream, in which case only the Constraint associated with that variable will be deactivated.

Parameters **var** – name of a variable in the Stream for which the corresponding Constraint should be deactivated (default = None).

Returns None

display(*side='source'*, *display_constraints=False*, *tolerance=1e-06*, *ostream=None*, *prefix=''*)
Display the contents of Stream Block.

Parameters

- **side** – side of Stream to display values from (default = 'soruce'). Valid values are 'source' and 'destination'.
- **display_constraints** – indicates whether to display Constraint information (default = False).
- **tolerance** – tolerance to use when checking if Stream is converged. (default = 1e-6).
- **ostream** – output stream (default = None)
- **prefix** – str to append to each line of output (default = '')

class idaes.core.stream.**VarDict**
Bases: dict

This class creates an object which behaves like a Pyomo IndexedVar. It is used to contain the separate Vars contained within IndexedPorts, and make them look like a single IndexedVar. This class supports the fix, unfix and display attributes.

display(*side='source'*, *ostream=None*, *prefix=''*)
Print component information

Parameters

- **side** – which side of port to display (default = 'source'). Valid values are 'source' or 'destination'.
- **ostream** – output stream (default = None)
- **prefix** – str to append to each line of output (default = '')

Returns None

fix(*value=None*, *side='destination'*)
Method to fix Vars.

Parameters

- **value** – value to use when fixing Var (default = None).
- **side** – side of port to fix (default = 'destination'). Valid values are 'source', 'destination' or 'all'.

Returns None

unfix(*side='destination'*)
Method to unfix Vars.

Parameters

- **value** – value to use when fixing Var (default = None)
- **side** – side of port to fix (default = 'destination'). Valid values are 'source', 'destination' or 'all'.

Returns None

idaes.core.unit_model module

Base clase for unit models

class idaes.core.unit_model.**UnitBlockData**(*component*)
Bases: *idaes.core.process_base.ProcessBlockData*

This is the class for process unit operations models. These are models that would generally appear in a process flowsheet or superstructure.

CONFIG = <pyutilib.misc.config.ConfigBlock object>

build()
General build method for UnitBlockData. 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

build_inlets(*holdup=None*, *inlets=None*, *num_inlets=None*)
This is a method to build inlet Port objects in a unit model and connect these to holdup blocks as needed. This method supports an arbitary number of inlets and holdup blocks, and works for both simple (0D) and 1D IDAES holdup blocks.

Keyword Arguments

- **= holdup block to which inlets are associated. If left None,** (holdup) – assumes a default holdup (default = None).
- **= argument defining inlet names (default** (*inlets*) – None). inlets may be None or list. - None - assumes a single inlet. - list - use names provided in list for inlets (can be

  other iterables, but not a string or dict)
- **= argument indication number** (*num_inlets*) – construct (default = None). Not used if inlets arg is provided. - None - use inlets arg instead - int - Inlets will be named with sequential numbers from 1

  to num_inlets.

Returns A Pyomo Port object and assoicated components.

build_outlets(*holdup=None*, *outlets=None*, *num_outlets=None*, *split_type='flow'*)
This is a method to build outlet Port objects in a unit model and connect these to holdup blocks as needed. This method supports an arbitary number of outlets and holdup blocks, and works for both simple (0D) and 1D IDAES holdup blocks.

Keyword Arguments

- **= holdup block to which inlets are associated. If left None,** (*holdup*) – assumes a default holdup (default = None).
- **= argument defining outlet names (default** (*outlets*) – None). outlets may be None or list. - None - assumes a single outlet. - list - use names provided in list for outlets (can be

  other iterables, but not a string or dict)
- **= argument indication number** (*num_outlets*) – construct (default = None). Not used if outlets arg is provided. - None - use outlets arg instead - int - Outlets will be named with sequential numbers from

  1 to num_outlets.
- **= argument defining method to use to split outlet flow** (*split_type*) – in case of multiple outlets (default = 'flow'). - 'flow' - outlets are split by total flow - 'phase' - outlets are split by phase - 'component' - outlets are split by component - 'total' - outlets are split by both phase and

  component

  – **'duplicate' - all outlets are duplicates of the** total outlet stream.

Returns A Pyomo Port object and assoicated components.

display_P()
Display pressure variables associated with the UnitBlockData.

display_T()
Display temperature variables associated with the UnitBlockData.

display_flows()
Display component flow variables associated with the UnitBlockData.

display_total_flows()
Display total flow variables associated with the UnitBlockData.

display_variables(*simple=True*, *descend_into=True*)
Display all variables associated with the UnitBlockData.

Parameters **simple** (*bool, optional*) – Print a simplified version showing only variable values.

initialize(*state_args=None*, *outlvl=0*, *solver='ipopt'*, *optarg={'tol': 1e-06}*)
This is a general purpose initialization routine for simple unit models. This method assumes a single Holdup block called holdup, 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 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

is_process_unit()
Tag to indicate that this object is a process unit.

model_check()
This is a general purpose initialization routine for simple unit models. This method assumes a single Holdup block called holdup and tries to call the model_check method of the holdup 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 Holdup blocks present.

Parameters **None** –

Returns None

class idaes.core.unit_model.**UnitBlock**(**args*, ***kwargs*)
Bases: *idaes.core.process_block.ProcessBlock*

Parameters

- **rule** – (Optional) A rule function or None. Default rule calls build().
- **concrete** – If True, make this a toplevel model. **Default** - False.
- **ctype** – (Optional) Pyomo ctype of the Block.
- **dynamic** – Indicates whether this model will be dynamic or not (default = 'use_parent_value'). 'use_parent_value' - get flag from parent (default = False) True - set as a dynamic model False - set as a steady-state model

Returns New UnitBlock instance

3.2.12 Conceptual Design using the IDAES Pyosyn Framework

This guide to the IDAES conceptual design framework will illustrate how to develop and optimize models to solve chemical engineering design problems. This draft of the guide was written based upon commit *d4f6059*, though much of it will also be true for the stable commit *48cc07e*.

To checkout a specific commit, you can simply use `git checkout 48cc07e`. This will put you in a detached head mode. To make a new local branch at that commit, use `git checkout -b new_branch_name`.

Contents

- *Conceptual Design using the IDAES Pyosyn Framework*
  - *Base Classes*
  - *Examples*

Base Classes

A handful of base classes provide necessary and useful functionality when building your models. The most important of these base classes are listed below:

FlowsheetModel

Each superstructure synthesis problem should declare a class inheriting from FlowsheetModel that provides information on how to calculate the objective function.

UnitModel

Process units such as reactors, flash drums, compressors, etc. should inherit from UnitModel

TODO Highlight some key attributes here equip_exists equip block lin_cuts oa_cuts units

LOA

The LOA module provides code for automatically executing the logic-based outer approximation optimization algorithm.

Examples

Water Treatment Model

The water treatment model uses three unit models: Feed, Reactor, and Sink. In this example, the objective is to minimize the cost of treating fixed contaminant loads from incoming water streams. The decision is between one or multiple treatment units in parallel or sequential order. Here, for convenience, the treatment units are modeled by a unit named *Reactor*. However, in practice, various forms of filtration, digestion, etc. may be used. The modeler is free to name their units as they please.

The steps for setting up and solving the water treatment model are thus:

1. Define problem

First, we create the class WaterModel, inheriting from FlowsheetModel. The objective functions for the problem are defined here. All modeling objects for the problem are also stored in this class.

2. Create unit models

Next, we create and define the Feed, Reactor, and Sink unit models.

3. Build and connect the model

The next step is to build and connect the model. Most of the code for this can be found in the build_model function:

The code within this function consists of a section importing data from external sources followed by the creation of the WaterModel, addition of units, and construction of the units. After data import, the first commands are:

```
m = WaterModel()
m.comps = comps
m.max_flow = 300
```

This creates a new WaterModel instance, specifies the chemical components that are relevant to the model, and introduces a maximum flowrate for all streams in the flowsheet.

Next, the command `m.add_unit()` is used to add the appropriate unit models to the superstructure. Notice that each unit takes arguments *name* and *parent*. The name should be unique for all units belonging to the same parent component. The parent should be the object to which the unit is being added. It may seem redundant to have `m.add_unit(Unit(parent=m))`, but it is currently necessary.

class idaes_models.core.process_base.**ProcessBase**

Next, connections between units are defined using `m.connect(from_unit, to_unit)`. By default, *connect* looks for a port named *outlet* on the from_unit, and a port named *inlet* on the to_unit. If a unit has multiple possible inlets or outlets, the correct port can be specified using the optional *to_port* or *from_port* arguments:

class idaes_models.core.process_base.**ProcessBase**

The next four commands invoke methods on the unit models to create the optimization model objects:

```
m.build_units()
m.build_links()
m.expand_streams()
propagate_var_fix(m)
```

The first command executes the `build()` command on each unit model. The next command creates Pyomo Port objects to link together the unit models. These Port objects are then expanded to normal equality constraints by the next command. Finally, the model is examined for `a = b` constraints in which one of the two variables is fixed. In that case, the other variable linked by equality is also fixed to the same value.

4. Solve the model

Finally, we solve the conceptual design problem. The high-level code for doing this can be found in the `main()` function:

The first command `m = build_model()` calls the previously described code. Next, the code iterates through the superstructure units and applies a linearization strategy:

```
for o in itervalues(m.units):
    o.apply_linear_relaxations()
    o.apply_OA_strategy(oa_ports=True)
```

The first command tells each unit to generate rigorous linear relaxations of nonlinear functions. The second command tells each unit that OA cuts should be generated for nonlinear equations, including for the interconnection units (ports).

Next, are invocations to the logic-based algorithms:

```
do_LOA(m, tol=100)
do_GLOA(m, tol=100, iterlim=32, contract_bounds=True, do_self_proj=False)
```

These functions call the respective algorithms in the loa module described above. Note that in this configuration, the solution to the LOA algorithm is used to initialize the GLOA algorithm, providing an initial upper bound (assuming minimization). However, either one of the two algorithms can be started and will execute independently of the other.

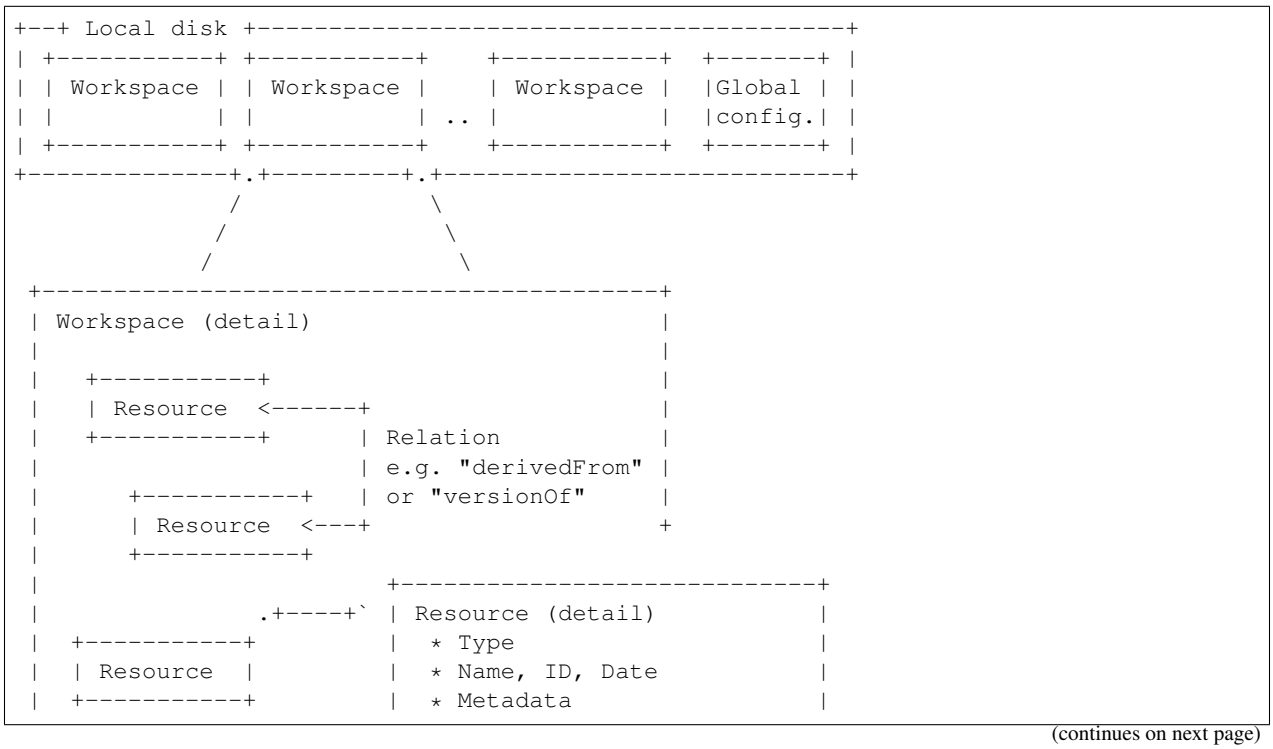
3.3 Data Management Framework (DMF)

3.3.1 About IDAES and the DMF

The Data Management Framework (DMF) is part of the Institute for the Design of Advanced Energy Systems (IDAES, pronounced "eye-DAY-iss") project. IDAES' goal is to 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." You can read more about IDAES on the project website.

The core functionality of IDAES is building and executing mathematical models of energy systems. The DMF complements this functionality by providing the ability to track the inputs (parameters, property models, etc.), processing (model code, Jupyter notebooks, scripts, etc.) and outputs (images, tables, etc.) of IDAES experiments. The DMF does not attempt to record all aspects of the environment, such as operating system and library versions. Instead, it focuses on tracking the environment-agnostic elements of the analysis that are specific to process engineering within the IDAES framework.

Below is a conceptual diagram of the DMF, showing the *Workspaces* residing on the local disk, and the contents of the workspaces being the tracked *Resources*, linked by *Relations* and containing metadata and (optionally) data files.



(continues on next page)

(continued from previous page)

```
|               `+----+. |  * Data files               |
|                        |                             |
+------------------------+  | +------+ +-----+   +------+ |
                           | | File | |File |...| File | |
                           | +------+ +-----+   +------+ |
                           +----------------------------+
```

The DMF provides a command line (terminal) interface and a Python library to create, retrieve, update, and delete resources and workspaces. The Python library is designed to be relatively easy to use interactively from within, e.g., Jupyter notebooks.

3.3.2 Getting started

Installation

The Data Management Framework (DMF) is installed automatically as part of the IDAES framework installation. Please refer to the general IDAES installation instructions for details.

Initialization

To use the DMF you will need to first create a *Workspace*. The instructions below show how to do this with the command-line interface, which is installed as the `dmf` command. Use `dmf create -i <directory>` to create a new workspace in the given directory, with interactive prompts for initial values:

```
$ dmf create -i /tmp/newdir
Interactive mode for workspace configuration
--
Field      : htmldocs
Description: HTML documentation paths. Each item is a directory containing Sphinx-
↪generated HTML docs
Enter value(s) separated by commas [{dmf_root}/doc/build/html]:
--
Field      : description
Description: A human-readable description of the workspace
Enter value []: This is an example workspace
--
Field      : name
Description: A short name for the workspace
Enter value []: example1
```

After you enter the final value, the workspace will be created. The command will then print the results of the `dmf info` command for the newly created workspace:

```
Workspace
  example1 - This is an example workspace

General information
 - Location = /tmp/newdir
 - Workspace identifier (_id) = b3a53e3eaf11421fb51e21d5baa5a9d1
 - Created = 2018-07-17T04:06:03.246855
 - Modified = 2018-07-17T04:06:03.246855
 - Num. resources = 0
```

(continues on next page)

(continued from previous page)

```
Configuration
 - conf = None
 - htmldocs = ['/home/dang/src/idaes/dangunter/idaes/doc/build/html']
 - showver = False
```

Note: The *htmldocs* configuration parameter is used internally for interactive help in the Jupyter Notebook. The value of this parameter does not usually need to be changed by the user.

Setting default workspace

If you are working on the command-line, it is convenient to set the current workspace for all commands. This value is recorded in the global configuration at `~/.dmf` (the file ".dmf" in your home directory), and can be viewed or changed with the `dmf init` command:

```
dmf init /tmp/newdir
```

Like the "create" command, this will also print out the current settings when it finishes. You can get this same printout by invoking the command with no arguments:

```
DMF global configuration </home/dang/.dmf>
> workspace = /tmp/newdir
```

Now all the other commands will use this workspace by default (the *-p/–path* option can override this default for most commands).

Adding a resource

You can add resources with the `dmf import` command or programmatically with the *Python API*. Importing files directly from the command line is convenient for things like Jupyter Notebooks and images. For example:

```
dmf import examples/dmf/generate_resources.ipynb
```

You can list resources in a workspace with `dmf ls`.

Getting help

For help on the command-line tools, the `dmf` command and its subcommands all use the `-h` option. For the main `dmf` command, the help will also list all the available subcommands.

If you are working in the Jupyter Notebook or another interactive Python environment, you can get details on the API calls through the standard help or "?" keywords. For example:

```
from idaes.dmf.dmf import DMF
help(DMF.add)   # generic Python help
DMF.add?        # IPython / Jupyter help
```

Next steps

There are more details and more commands in the full *command-line interface*.

For the Python API, you could start with some of the *example Jupyter notebooks*. and look at the *Python API* for details.

3.3.3 DMF Global Configuration

class idaes.dmf.dmf.**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 wthe 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 .dmf are formatted as YAML, with the following keys defined:

workspace Path to the default workspace directory.

An example file is shown below:

```
{workspace: /tmp/newdir}
```

3.3.4 DMF Workspace

As shown in the overview diagram on the *About IDAES and the DMF* page, the DMF *Workspace* is the container for all DMF *Resources* that can be worked on as a unit. Below is the documentation for the Workspace main class:

class idaes.dmf.workspace.**Workspace**(*path*, *create=False*, *add_defaults=False*)

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 DMF 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.

3.3.5 DMF Command-line interface

Overview

The DMF command-line interface (CLI) is used to perform common start-up, browsing, etc. tasks.

Getting started

Note: In the examples that follow the terminal shell prompt will be represented by a `$`. It will be omitted in cases where there is no terminal output shown.

To use the CLI, open a shell window and run "dmf <command>". For example, to see the help message, run:

```
dmf -h
```

All operations in the DMF occur in the context of a base container called a "workspace". This container has a configuration and all the metadata and files, called "resources". A workspace corresponds to a directory on the filesystem. You can have many workspaces, but only one is active at a time. To choose the currently active workspace, you can specify it explicitly for each command, e.g.:

```
dmf -p /path/to/workspace <commands...>
```

But this quickly becomes tedious. It is easier to use the DMF's *global configuration* to set the current workspace with the "init" command:

```
dmf init /path/to/workspace
```

After you run this command, subsequent commands will operate within this workspace. To create a blank workspace to run in, use "dmf create":

```
dmf create /path/to/workspace --name "MyWorkspace" --description "My first workspace"
```

You'll notice that this command prints out some information about the workspace at the end. To see that same information at any time for your current workspace, use "dmf info":

```
$ dmf info

Workspace
  MyWorkspace - My first workspace

General information
 - Location = /home/dang/src/workspace
 - Workspace identifier (_id) = 2f0a82e03e5747ccbcb04f83cb417f5a
 - Created = 2018-07-07 07:10:02
 - Modified = 2018-07-07 07:10:02
 - Num. resources = 0

Configuration
 - conf = None
 - htmldocs = ['/home/dang/src/idaes/doc/build/html']
 - showver = False
```

To see the *contents* of a workspace, a.k.a. the workspace **resources**, you can use the "dmf ls". This will be quite boring if you just created the workspace, since there are no resources in it yet:

```
$ dmf ls
$
```

To see some results from the listing, use "dmf import" to add a generic file resouce to the workspace and then try again with "ls":

```
$ echo 'hello' > hello.txt  # create the text file
$ dmf import hello.txt
Imported 1 resource
$ dmf ls
hello.txt:data
```

Command reference

The DMF command-line interface (CLI) uses the main command / subcommand interface that many people will know from tools such as git and subversion. The main command is `dmf`. The subcommands are:

- *dmf create*
- *dmf import*
- *dmf info*
- *dmf init*
- *dmf ls*
- *dmf ws*

dmf create

```
usage: dmf create [-h] [-v] [-c PATH] [--interactive] [--htmldocs VALUE]
                  [--description VALUE] [--name VALUE]
                  path
```

Positional Arguments

path Directory for new workspace. In this directory, a directory "config.yaml" will be created to hold metadata.

Named Arguments

-v, --verbose Default: 0

-c, --conf Use global config at PATH (default = ~/.dmf)

--interactive, -i Interactively ask for configuration values

Default: False

--htmldocs HTML documentation paths. Each item is a directory containing Sphinx-generated HTML docs (repeatable)

Default: ['{dmf_root}/doc/build/html']

--description A human-readable description of the workspace

Default: ""

--name A short name for the workspace

Default: ""

Examples:

Interactively create a new workspace: `dmf create -i $HOME/data/my-workspace`

---

dmf import

```
usage: dmf import [-h] [-v] [-c PATH] [-p PATH] [-x] [FILE [FILE ...]]
```

Positional Arguments

FILE One or more files, or patterns of files, to import (default=empty)

Default: []

Named Arguments

-v, --verbose	Default: 0
-c, --conf	Use global config at PATH (default = ~/.dmf)
-p, --path	Use workspace at PATH (default = from config or ".")
-x, --exitfirst	Exit on first error
	Default: False

---

dmf info

```
usage: dmf info [-h] [-v] [-c PATH] [PATH]
```

Positional Arguments

PATH	Search for DMF configuration at PATH (default=".")

Named Arguments

-v, --verbose	Default: 0
-c, --conf	Use global config at PATH (default = ~/.dmf)

Examples:

Get info about current workspace: `dmf info`

Get info about "~/data" directory: `dmf info -p ~/data`

---

dmf init

```
usage: dmf init [-h] [-v] [-c PATH] [WORKSPACE]
```

Positional Arguments

WORKSPACE	Set path to default workspace

Named Arguments

-v, --verbose	Default: 0
-c, --conf	Use global config at PATH (default = ~/.dmf)

Examples:

Print current settings: `dmf init`

Print settings from "/opt/dmf" `dmf init -c /opt/dmf`

Set workspace to "~/data/project/my-dmf" `dmf init ~/data/project/my-dmf`

---

dmf ls

```
usage: dmf ls [-h] [-v] [-c PATH] [-l] [-r] [PATH]
```

Positional Arguments

PATH	List resources in workspace at PATH (default = from config or ".")

Named Arguments

-v, --verbose	Default: 0
-c, --conf	Use global config at PATH (default = ~/.dmf)
-l, --long	Use a long listing format
	Default: False
-r, --relations	With long listing, show relationships
	Default: False

Examples:

List objects in current workspace: `dmf ls`

List objects, in long format, in workspace at $HOME/foo: `dmf ls -l $HOME/foo`

List objects and their relations (in long format): `dmf ls -lr`

---

dmf ws

```
usage: dmf ws [-h] [-v] [-c PATH] [-p PATH]
```

Named Arguments

-v, --verbose	Default: 0
-c, --conf	Use global config at PATH (default = ~/.dmf)

-p, --path List workspaces at/below PATH (default = ".")

Default: "."

Examples:

List workspaces below current directory: `dmf ws`

List workspaces below user $HOME directory: `dmf ws -p ~`

3.3.6 DMF API Examples

The *DMF API* allows you to add, find, retrieve, and delete *Resources* <*resources*> from the DMF. See the *dmf.DMF* class docs for the full API reference.

The rest of this section has a brief description, with examples, of the major operations of the API.

Initialize

To perform these functions, you should first create an instance of the DMF object, which is linked of course to its configuration and the underlying file system:

```
from idaes.dmf import dmf
my_dmf = dmf.DMF(path='/path/to/my-workspace')
```

The path given to the constructor is the DMF's "workspace" directory, that holds the configuration files, resource metadata and data. This directory is laid out like this:

```
my-workspace: Root DMF "workspace" directory
 |
 +- config.yaml: Configuration file
 +- resourcedb.json: Resource metadata "database" (if using TinyDB)
 +- files: Data files for all resources
    |
    +- 195d6e5e-73da-4a0e-85f4-129bfcfda64b: Unique hash of directory
    |  with the datafiles (this is in the `datafiles_dir` attribute
    |  of the Resource object).
    |  |
    |  +- <fileA>: Datafile for that resource
    |  +- <fileB>: Another datafile, etc.
    |
    +- 36206516-88ce-400f-ac72-e42918e34aaf: Another directory, etc.
```

The "resourcedb.json" is a JSON file that is used by TinyDB. If you configured the DMF to use another database backend (not currently an option, but definitely in the roadmap), this file will not exist.

Find resources

You can *find* resources by searching on any of the fields that are part of the resource metadata. The basic syntax is a filter consisting of key/value pairs. The key is the name of the attribute in the *Resource* class; sub-attributes should be referred to with dotted notation. For example, the query for looking for version 1.0.0 would be:

```
spec = {'version.revision': '1.0.0'}
resources = my_dmf.find(spec)
```

For dates, you should pass a *datetime.datetime* or an instance the *Pendulum* class, from the pendulum package. The pendulum package is pretty easy to use, as shown here:

```
import pendulum
last_week = pendulum.now().subtract(weeks=1)
spec = {'version.created': last_week}
```

Combining expressions. You can combine multiple filter expressions by simply putting them together in the same dictionary. A given record must match all fields to match. There is currently no built-in way to select records matching less than all the fields:

```
halloween = pendulum.Pendulum(2017, 10, 31)
scary _query = {'version.revision': '13.13.13',
                'version.created': halloween}
```

Inequalities. There are modifiers for inequalities in the style of MongoDB, i.e. are nested dictionaries with the key being the inequality prefixed with a "$" and the value being the end of the range. An example should clarify the general idea, see the documentation for *find* for details:

```
from pendulum import Pendulum
# see if 'version.created' is in October 2017
#   $ge   :  greater than or equal
#   $le   :  less than or equal
spec = {'version.created': {'$ge': Pendulum('2017-10-1'),
                            '$le': Pendulum('2017-10-31')}
       }
```

Lists. There is some special notation to help with lists of values. If the resource's attribute value is a list, e.g. the *tags* attribute, then you need to pass a list of values to match against. By default, the method will return resources that match *any* of the provided values. If you want to only get resources that match *all* of the provided values, then add a '!' after the attribute name. For example:

```
# match any resource with tags "MEA" or "model"
spec = {'tags': ['MEA', 'model']}
# match resources that have both tags "MEA" and "model"
spec = {'tags!': ['MEA', 'model']}
```

Find related resources

You may to want to find a group of resources that are connected by some (set of) relationships. You can imagine the resources in a mathematical graph, where the resource information is contained at the vertices and the relations are the edges, and you want to find vertices reachable from some given starting point. The *find_related* method is used to navigate all "relations" (as they are called in the DMF code) from a given resource.

We adopt the Resource Description Format (RDF) terminology for relations, where in a relation between two resources, the starting resource is called the "subject", the ending resource is called the "object" and the type of relation between them is called the "predicate". See examples for clarification.

There are a only a few predicates, i.e. types of relations, in the DMF. These are enumerated in the *resource.RelationType* class.

derived The object is derived from the subject, as in a property model being derived from property data. See *uses* description for the difference between the two.

contains The object is part of the subject. This is used for describing "in", or container/contained, relationships such as some type of resource being part of an `experiment.Experiment`, or an Experiment being part of another Experiment.

uses The object was/is used by the subject in some way. This is like *derived* in that there is a dependency on the subject, but different because the object was created before that dependency existed. If the object depends on the subject in order to be created, then you should express that as a *derived* relationship instead.

version The object is a (newer) version of the subject. The "version" field should be different in the two resources. You should use this relation where the object represents changes in the subject, and use *derived* where the subject and object represent different resources.

The `find_related` method starts with a resource and then navigates through relations to other resources, to some maximum depth (distance) from the starting resource. The parameters control which relations are included, which types of resources, and how deep the search can go. For example:

```
from idaes.dmf import DMF
from idaes.dmf.resource import ResourceTypes
ws = DMF(path='/my/workspace')
for exp in ws.find({'type': ResourceTypes.xp}):
    println ('Experiment {}:'.format(exp.name))
    info_fields = ['name', 'type']
    for depth, rel, info in ws.find_related(exp, meta=info_fields):
        indent = '    ' * depth
        println('{}* {} -> {} [{}]'.format(indent, rel.predicate,
            info['name'], info['type']))
```

Retrieve resources

Resources can be retrieved directly, and reasonably quickly, by their identifiers. The methods that do that start with the word "fetch". The *fetch_one* method gets a single resource and `fetch_many` gets a number of them at once:

```
from idaes.dmf import DMF
from idaes.dmf.resource import Resource

my_dmf = DMF(path='/my/workspace')

def add_test_resource():
    r = Resource(desc='test resource')
    dmf.add(r)  # Resource id is assigned at this point
    return r.id_

# Demonstrate, in a somewhat contrived way, how "fetch_one" works
rid = add_test_resource()
r = my_dmf.fetch_one(rid)  # returns the new resource

# Demonstrate, in a somewhat contrived way, how "fetch_many" works
rids = [add_test_resource() for i in range(10)]
rlist = my_dmf.fetch_many(rids)  # returns the new resources
```

Add resources

The `add` method puts a resource into the DMF. This is mostly simply adding the metadata in the resource.

Datafiles. At this time, files in the *datafiles* attribute are copied and, if they were marked as temporary, the original is deleted. The default is to copy, and not delete the original. If a file is not copied, **it is the user's responsibility** to update the record if the original file moves. Below are examples of the three possibilities:

```
from idaes_dmf import dmf
from idaes_dmf.resource import Resource, FilePath

my_dmf = dmf.DMF(path='/where/my/config/lives/')
r = Resource(desc='test resource')

# (1) Copy, and don't remove
r.datafiles.append(FilePath(path='/tmp/my-data.csv'))

# (2) Copy, and remove original
r.datafiles.append(FilePath(path='/tmp/temp-data.csv',
                            tempfile=True))
# (3) Neither copy nor remove
r.datafiles.append(FilePath(path='/home/bigfile', copy=False))

# Att this point, and not before, the copies occur
dmf.add(r)

# This is an error! It makes no sense to ask the file
# to be removed, but not copied (just a file delete?!)
r.datafiles.append(FilePath(path='foo', copy=False, tempfile=True))
# ^^ raises ValueError
```

Update resources

Resources are updated by providing a new Resource object to the `update` method. However, you cannot change the "type" of a Resource in the process of updating it:

```
from idaes_dmf import dmf
from idaes_dmf.resource import Resource

my_dmf = dmf.DMF(path='/where/my/config/lives/')
r = Resource(desc='test resource')
my_dmf.add(r)

# Valid update
r.aliases.append('tester')
my_dmf.update(r)

# Invalid update
r.type = 'free to be me'
my_dmf.update(r)  # !! Raises: errors.DMFError
```

Delete resources

Resources are deleted by their *id*, or a filter, using the *remove* method:

```
from idaes_dmf import dmf
from idaes_dmf.resource import Resource, FilePath

my_dmf = dmf.DMF(path='/where/my/config/lives/')
r = Resource(desc='test resource')
my_dmf.add(r)
```

(continues on next page)

(continued from previous page)

```
# (a) delete by ID:
my_dmf.remove(identifier=r.id_)
# (b) alternatively, delete by filter:
my_dmf.remove(filter_dict={'desc':'test_resource'})
```

3.3.7 DMF API Documentation

Information on specific functions, classes, and methods for the IDAES Data Management Framework (DMF).

idaes.dmf package

IDAES Data Management Framework (DMF)

The DMF lets you save, search, and retrieve provenance related to your models.

This package is documented with Sphinx. To build the documentation, change to the 'docs' directory and run, e.g., 'make html'.

Resource representaitons.

Subpackages

idaes.dmf.schemas package

Submodules

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.**cat_resources**(*path*, *objects=()*, *color=True*)

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_info**(*dirname*)

idaes.dmf.commands.**workspace_init**(*dirname*, *metadata*)
Initialize from root at *dirname*, set environment variable for other commands, and parse config file.

idaes.dmf.dmf module

Data Management Framework

class idaes.dmf.dmf.**DMF**(*path=''*, *name=None*, *desc=None*, ***ws_kwargs*)
Bases: *idaes.dmf.workspace.Workspace*, traitlets.traitlets.HasTraits

Data Management Framework (DMF).

Expected usage is to instantiate this class, once, and then use it for storing, searching, and retrieve *resource* s 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*.

CONF_DATA_DIR = 'datafile_dir'

CONF_DB_FILE = 'db_file'

CONF_HELP_PATH = 'htmldocs'

add(*rsrc*)
Add a resource and associated files.

Parameters **rsrc** (resource.Resource) – The resource

Returns (str) Resource ID

Raises DMFError, DuplicateResourceError

count()

datafile_dir
A trait for unicode strings.

db_file
A trait for unicode strings.

fetch_many(*rid_list*)
Fetch multiple resources, by their identifiers.

Parameters **rid_list** (*list*) – List of integer resource identifers

Returns (list of resource.Resource) List of found resources (may be empty)

fetch_one(*rid*)
Fetch one resource, from its identifier.

Parameters **rid** (*int*) – Resource identifier

Returns (resource.Resource) The found resource, or None if no match

find(*filter_dict=None*, *id_only=False*)
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 four possible types of values:

1. scalar string or number (int, float): Match resources that have this exact value for the given attribute.
2. date, as datetime.datetime or pendulum.Pendulum instance: Match resources that have this exact date for the given attribute.
3. 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.
4. 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 (!=)

Parameters

- **filter_dict** (*dict*) – Search filter.
- **id_only** (*bool*) – If true, return only the identifier of each resource; otherwise a Resource object is returned.

Returns (list of int|Resource) Depending on the value of *id_only*.

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** (*int*) – Identifier in *id_* attribute of 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.dmf.`**DMFConfig**(*defaults=None*)

Bases: `object`

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 wthe 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 .dmf are formatted as YAML, with the following keys defined:

workspace Path to the default workspace directory.

An example file is shown below:

```
{workspace: /tmp/newdir}
```

DEFAULTS = {'workspace': '/home/ksb/Projects/IDAES/github/IDAES/idaes/doc'}

```
    WORKSPACE = 'workspace'
    filename = '/home/ksb/.dmf'
    save()
    workspace
```

idaes.dmf.dmf.**get_propertydb_table**(*rsrc*)

idaes.dmf.errors module

Exception classes.

exception idaes.dmf.errors.**AlamoDisabledError**
Bases: *idaes.dmf.errors.AlamoError*

exception idaes.dmf.errors.**AlamoError**(*msg*)
Bases: *idaes.dmf.errors.DmfError*

exception idaes.dmf.errors.**BadResourceError**
Bases: *idaes.dmf.errors.ResourceError*

exception idaes.dmf.errors.**CommandError**(*command*, *operation*, *details*)
Bases: Exception

exception idaes.dmf.errors.**DMFBadWorkspaceError**(*path*, *why*)
Bases: *idaes.dmf.errors.DMFError*

exception idaes.dmf.errors.**DMFError**(*detailed_error*)
Bases: Exception

exception idaes.dmf.errors.**DMFWorkspaceNotFoundError**(*path*)
Bases: *idaes.dmf.errors.DMFError*

exception idaes.dmf.errors.**DataFormatError**(*dtype*, *err*)
Bases: *idaes.dmf.errors.DmfError*

exception idaes.dmf.errors.**DmfError**
Bases: Exception

exception idaes.dmf.errors.**DuplicateResourceError**(*op*, *id_*)
Bases: *idaes.dmf.errors.ResourceError*

exception idaes.dmf.errors.**FileError**
Bases: Exception

exception idaes.dmf.errors.**InvalidRelationError**(*subj*, *pred*, *obj*)
Bases: *idaes.dmf.errors.DmfError*

exception idaes.dmf.errors.**ModuleFormatError**(*module_name*, *type_*, *what*)
Bases: Exception

exception idaes.dmf.errors.**NoSuchResourceError**(*name=None*, *id_=None*)
Bases: *idaes.dmf.errors.ResourceError*

exception idaes.dmf.errors.**ParseError**
Bases: Exception

exception idaes.dmf.errors.**ResourceError**
Bases: Exception

exception idaes.dmf.errors.**SearchError**(*spec*, *problem*)
Bases: Exception

exception idaes.dmf.errors.**WorkspaceConfMissingField**(*path*, *name*, *desc*)
Bases: *idaes.dmf.errors.WorkspaceError*

exception idaes.dmf.errors.**WorkspaceConfNotFoundError**(*path*)
Bases: *idaes.dmf.errors.WorkspaceError*

exception idaes.dmf.errors.**WorkspaceError**
Bases: Exception

exception idaes.dmf.errors.**WorkspaceNotFoundError**(*from_dir*)
Bases: *idaes.dmf.errors.WorkspaceError*

idaes.dmf.experiment module

The 'experiment' is a root container for a coherent set of 'resources'.

class idaes.dmf.experiment.**Experiment**(*dmf*, ***kwargs*)
Bases: *idaes.dmf.resource.Resource*

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(***kwargs*)
Get a copy of this experiment. The returned object will have been added to the DMF.

Parameters **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*

dmf

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*, ***kw*)
Get one or more files with HTML documentation for the given object, in paths referred to by the dmf instance.

idaes.dmf.help.**get_html_docs**(*dmf*, *module_*, *name*, *sphinx_version=(1*, *5*, *5)*)

idaes.dmf.magics module

Jupyter magics for the DMF.

class idaes.dmf.magics.**DmfMagics**(*shell*)
Bases: IPython.core.magic.Magics

NEED_INIT_CMD = {'help': '+', 'info': '*'}

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 or more arguments looks for help in the docs on the given objects or classes.

dmf_info(**topics*)
Provide information about DMF current state for whatever 'topics' are provided. With no topic, provide general information about the configuration.

Parameters **topics** (*(List[str])*) – List of topics

Returns None

dmf_init(*path*, **extra*)
Initialize DMF (do this before most other commands).

Parameters **path** (*str*) – Full path to DMF home

dmf_list()
List resources in the current workspace.

dmf_workspaces(**paths*)
List DMF workspaces.

Parameters **paths** (*List[str]*) – Paths to search, use "." by default

idaes(*line*)
%idaes magic

idaes_help(**names*)
Provide help on IDAES objects and classes.

Invoking with no arguments gives general help. Invoking with one or more arguments looks for help in the docs on the given objects or classes.

magics = {'cell': {}, 'line': {'dmf': 'dmf', 'idaes': 'idaes'}}

registered = True

idaes.dmf.magics.**register**()

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=''*)
Bases: KeyError

Error for :meth:PropertyData.add_csv()

class idaes.dmf.propdata.**Fields**
Bases: *idaes.dmf.tabular.Fields*

Constants for fields.

C_PROP = 'property'

C_STATE = 'state'

class idaes.dmf.propdata.**PropertyColumn**(*name*, *data*)
Bases: *idaes.dmf.tabular.Column*

Data column for a property.

data()

type_name = 'Property'

class idaes.dmf.propdata.**PropertyData**(*data*)
Bases: *idaes.dmf.tabular.TabularData*

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

embedded_units = '(.*)\\((.*)\\)'

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, excluse property data

Returns List of column names.

Return type list[str]

properties

states

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*)
Bases: *idaes.dmf.tabular.Metadata*

Class to import property metadata.

class idaes.dmf.propdata.**PropertyTable**(*data=None*, *metadata=None*)
Bases: *idaes.dmf.tabular.Table*

Property data and metadata together (at last!)

static 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 ..."}
    ],
```

(continues on next page)

(continued from previous page)

```
    "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*)
Bases: *idaes.dmf.tabular.Column*

Data column for a state.

data()

type_name = 'State'

idaes.dmf.propdata.**convert_csv**(*meta_csv*, *datatype*, *data_csv*, *nstates*, *output*)

idaes.dmf.resource module

Resource representaitons.

class idaes.dmf.resource.**Code**(**args*, ***kwargs*)
Bases: *idaes.dmf.resource.TraitContainer*

Some source code, such as a Python module or C file.

This can also refer to packages or entire Git repositories.

desc
Description of the code

idhash
Git or other unique hash

language
Programming language, e.g. "Python" (the default).

location
Flie path or URL location for the code

name
Name of the code object, e.g. Python module name

release
Version of the release, default is '0.0.0'

type
Type of code resource, must be one of – 'method', 'function', 'module', 'class', 'file', 'package', 'repository', or 'notebook'.

class idaes.dmf.resource.**Contact**(**args*, ***kwargs*)
Bases: *idaes.dmf.resource.TraitContainer*

Person who can be contacted.

email
Email of the contact

name
Name of the contact

class idaes.dmf.resource.**DateTime**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
Bases: traitlets.traitlets.TraitType

A trait type for a datetime.

Input can be a string, float, or tuple. Specifically:

- string, ISO8601: YYYY[-MM-DD[Thh:mm:ss[.uuuuuu]]]
- float: seconds since Unix epoch (1/1/1970)
- tuple: format accepted by datetime.datetime()

No matter the input, validation will transform it into a floating point number, since this is the easiest form to store and search.

default_value = 0

info_text = 'a datetime'

classmethod isoformat(*ts*)

validate(*obj*, *value*)

class idaes.dmf.resource.**FilePath**(*tempfile=False*, *copy=True*, ***kwargs*)
Bases: *idaes.dmf.resource.TraitContainer*

Path to a file, plus optional description and metadata.

So that the DMF does not break when data files are moved or copied, the default is to copy the datafile into the DMF workspace. This behavior can be controlled by the *copy* and *tempfile* keywords to the constructor.

For example, if you have a big file you do NOT want to copy when you create the resource:

```
FilePath(path='/my/big.file', desc='100GB file', copy=False)
```

On the other hand, if you have a file that you want the DMF to manage entirely:

```
FilePath(path='/some/file.txt', desc='a file', tempfile=True)
```

CSV_MIMETYPE = 'text/csv'

desc
Description of the file's contents

do_copy

fullpath

is_tmp

metadata
Metadata to associate with the file

mimetype
MIME type

open(*mode='r'*)

path
Path to file

read(**args*)

root

subdir
Unique subdir

class idaes.dmf.resource.**FlowsheetResource**(**args*, ***kwargs*)
Bases: *idaes.dmf.resource.Resource*

Flowsheet resource & factory.

classmethod from_flowsheet(*obj*, ***kw*)

class idaes.dmf.resource.**Identifier**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
Bases: traitlets.traitlets.TraitType

Unique identifier.

Will set it itself automatically to a 32-byte unique hex string. Can only be set to strings

default_value = None

expr = re.compile('[0-9a-f]{32}')

info_text = 'Unique identifier'

validate(*obj*, *value*)

class idaes.dmf.resource.**PropertyDataResource**(*property_table=None*, ***kwargs*)
Bases: *idaes.dmf.resource.TabularDataResource*

Property data resource & factory.

idaes.dmf.resource.**R_DERIVED = 'derived'**
Constants for RelationType predicates

class idaes.dmf.resource.**RelationType**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
Bases: traitlets.traitlets.TraitType

Traitlets type for RDF-style triples relating resources to each other.

Predicates = {'derived', 'uses', 'contains', 'version'}

info_text = 'triple of (subject-id, predicate, object-id), all strings, with a predica

validate(*obj*, *value*)

class idaes.dmf.resource.**Resource**(**args*, ***kwargs*)
Bases: *idaes.dmf.resource.TraitContainer*

A dynamically typed resource.

Resources have metadata and (same for all resoures) a type-specific "data" section (unique to that type of resource).

ID_FIELD = 'id_'

TYPE_FIELD = 'type'

aliases
List of aliases for the resource

codes
List of code objects (including repositories and packages) associated with the resource. Each value is a *Code*.

collaborators
List of other people involved. Each value is a *Contact*.

copy (***kwargs*)
Get a copy of this Resource.

As a convenience, optionally set some attributes in the copy.

Parameters **kwargs** – Attributes to set in new instance after copying.

Returns: Resource: A deep copy.

The copy will have an empty (zero) *identifier* and a new unique value for *uuid*. The relations are not copied.

static create_relation (*subj*, *pred*, *obj*)
Create a relationship between two Resource instances.

Parameters

- **subj** (Resource) – Subject
- **pred** (*str*) – Predicate
- **obj** (Resource) – Object

Returns None

Raises TypeError – if subject & object are not Resource instances.

created
Date and time when the resource was created. This defaults to the time when the object was created. Value is a *DateTime*.

creator
Creator of the resource. Value is a *Contact*.

data
An instance of a Python dict.

datafiles
List of data files associated with the resource. Each value is a *FilePath*.

datafiles_dir
Datafiles subdirectory (single directory name)

desc
Description of the resource

help (*name*)
Return descriptive 'help' for the given attribute.

Parameters **name** (*str*) – Name of attribute

Returns Help string, or error starting with "Error: "

Return type str

id_
Integer identifier for this Resource. You should not set this yourself. The value will be automatically overwritten with the database's value when the resource is added to the DMF (with the *.add()* method).

modified
Date and time the resource was last modified. This defaults to the time when the object was created. Value is a *DateTime*.

name
Human-readable name for the resource (optional)

property_table
For property data resources, this property builds and returns a PropertyTable object.

Returns

A representation of metadata and data in this resource.

Return type *propdata.PropertyTable*

Raises TypeError – if this resource is not of the correct type.

relations
Validate values in a list as belonging to a given TraitType.

This can be used in place of the Traitlets.List class.

sources
Sources from which resource is derived, i.e. its provenance. Each value is a *Source*.

table

For tabular data resources, this property builds and returns a Table object.

Returns

A representation of metadata and data in this resource.

Return type *tabular.Table*

Raises TypeError – if this resource is not of the correct type.

tags
List of tags for the resource

type
Type of this Resource. See *ResourceTypes* for standard values for this attribute.

uuid
Universal identifier for this resource

version
Version of the resource. Value is a *SemanticVersion*.

class idaes.dmf.resource.**ResourceTypes**
Bases: object

Standard resource type names.

Use these as opaque constants to indicate standard resource types. For example, when creating a Resource:

```
rsrc = Resource(type=ResourceTypes.property_data, ...)
```

data = 'data'
Data (e.g. result data)

experiment = 'experiment'
Experiment

fs = 'flowsheet'
Flowsheet resource.

jupyter = 'notebook'

jupyter_nb = 'notebook'

nb = 'notebook'
Jupyter Notebook

property_data = 'propertydb'
Property data resource, e.g. the contents are created via classes in the *idaes.dmf.propdata* module.

python = 'python'
Python code

surrmod = 'surrogate_model'
Surrogate model

tabular_data = 'tabular_data'
Tabular data

xp = 'experiment'

class idaes.dmf.resource.**SemanticVersion**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
Bases: traitlets.traitlets.TraitType

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:

- 1 => valid => (1, 0, 0, '')
- rc3 => invalid: no number
- 1.1 => valid => (1, 1, 0, '')
- 1a => valid => (1, 0, 0, 'a')
- 1.a.1 => invalid: non-numeric can only go at end
- 1.12.1 => valid => (1, 12, 1, '')
- 1.12.13-1 => valid => (1, 12, 13, '1')
- 1.12.13.x => invalid: too many parts

default_value = (0, 0, 0, '')

info_text = 'semantic version major, minor, patch, & modifier'

classmethod pretty(*values*)

validate(*obj*, *value*)

class idaes.dmf.resource.**Source**(**args*, ***kwargs*)
Bases: *idaes.dmf.resource.TraitContainer*

A work from which the resource is derived.

date
Date associated with resource

doi
Digital object identifier

isbn
ISBN

language
The primary language of the intellectual content of the resource

source
The work, either print or electronic, from which the resource was derived

class idaes.dmf.resource.**TabularDataResource**(*table=None*, ***kwargs*)
Bases: *idaes.dmf.resource.Resource*

Tabular data resource & factory.

class idaes.dmf.resource.**TraitContainer**(**args*, ***kwargs*)
Bases: traitlets.traitlets.HasTraits

Base class for Resource, that knows how to serialize and parse its traits.

as_dict()

classmethod from_dict(*d*)

class idaes.dmf.resource.**Triple**(*subject*, *predicate*, *object*)
Bases: tuple

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

class idaes.dmf.resource.**ValidatingList**(**args*, ***kwargs*)
Bases: traitlets.traitlets.List

Validate values in a list as belonging to a given TraitType.

This can be used in place of the Traitlets.List class.

validate_elements(*obj*, *value=None*)
This is called when the initial value is set.

class idaes.dmf.resource.**Version**(**args*, ***kwargs*)
Bases: *idaes.dmf.resource.TraitContainer*

Version of something (code, usually).

created
When this version was created. Default "empty", which is encoded as the start of Unix epoch (1970/01/01).

name
Name given to version

revision
Revision, e.g. 1.0.0rc3

idaes.dmf.resource.**get_resource_structure**()

idaes.dmf.resourcedb module

Resource database.

class idaes.dmf.resourcedb.**ResourceDB**(*dbfile=None*, *connection=None*)
Bases: object

A database interface to all the resources within a given DMF workspace.

delete(*id_=None*, *idlist=None*, *filter_dict=None*)
Delete one or more resources with given identifiers.

Parameters

- **id** (*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.

Returns (list[str]) Identifiers

find(*filter_dict*, *id_only=False*)
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.

Returns (list of int|Resource) Depending on the value of *id_only*

find_related(*rsrc_id*, *filter_dict=None*, *outgoing=True*, *maxdepth=0*, *meta=None*)
Find all resources connected to the identified one.

Parameters

- **rsrc_id** –
- **filter_dict** –
- **outgoing** –
- **maxdepth** –
- **meta** (*List[str]*) –

Returns Generator of (depth, relation, metadata)

Raises KeyError if the resource is not found.

get(*identifier*)

put(*resource*)

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, e.g. result of *Resource.as_dict()*.

Returns The id_ of the resource, or None if it was not updated.

Return type int

Raises ValueError – If new resource is of wrong type

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*)

Bases: object

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*)
Bases: object

Generic, abstract column

data()

type_name = 'generic'

class idaes.dmf.tabular.**Fields**
Bases: object

Constants for field names.

AUTH = 'authors'

COLTYPE = 'type'

DATA = 'data'

DATA_ERRORS = 'errors'

DATA_ERRTYPE = 'error_type'

DATA_NAME = 'name'
Keys for data mapping

DATA_UNITS = 'units'

DATA_VALUES = 'values'

DATE = 'date'

DTYPE = 'datatype'

INFO = 'info'

META = 'meta'

ROWS = 'rows'

TITLE = 'title'

VALS = 'values'

class idaes.dmf.tabular.**Metadata**(*values=None*)

Bases: object

Class to import metadata.

as_dict()

author

Publication author(s).

datatype

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.

line_expr = re.compile('\\s*(\\w+)\\s*,\\s*(.*)\\s*')

source

Full publication info.

source_expr = re.compile('\\s*(.*)\\s*,\\s*"(.*)"\\s*,\\s*(.*)\\s*')

title

Publication title.

class idaes.dmf.tabular.**Table**(*data=None*, *metadata=None*)

Bases: *idaes.dmf.tabular.TabularObject*

Tabular data and metadata together (at last!)

add_metadata(*m*)

as_dict()
Represent as a Python dictionary.

Returns (dict) Dictionary representation

data

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"
        }
    ]
}
```

metadata

class idaes.dmf.tabular.**TabularData**(*data*, *error_column=False*)
Bases: object

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

columns

embedded_units = '(.*)\\((.*)\\)'

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**
Bases: object

Abstract Property data class.

as_dict()
Return Python dict representation.

idaes.dmf.util module

Utility functions.

class idaes.dmf.util.**CPrint**(*color=True*)
Bases: object

Colorized terminal printing.

Codes are below. To use:

cprint = CPrint() cprint('This has no colors') # just like print() cprint('This is @b[blue] and @_r[red underlined]')

You can use the same class as a no-op by just passing *color=False* to the constructor.

COLORS = {'*': '\x1b[1m', '-': '\x1b[2m', '.': '\x1b[0m', '_': '\x1b[4m', 'b': '\x

colorize(*s*)

println(*s*)

write(*s*)

class idaes.dmf.util.**TempDir**(**args*)
Bases: object

Simple context manager for mkdtemp().

idaes.dmf.util.**datetime_timestamp**(*v*)
Get numeric timestamp. This will work under both Python 2 and 3.

idaes.dmf.util.**find_process_byname**(*name*, *uid=None*)
Generate zero or more PIDs where 'name' is part of either the first or second token in the command line. Optionally also filter the returned PIDs to only those with a 'real' user UID (UID) equal to the provided uid. If None, the default, is given, then use the current process UID. Providing a value of < 0 will skip the filter.

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_logger**(*name=''*)
Create and return a DMF logger instance.

The name should be lowercase letters like 'dmf' or 'propdb'.

Leaving the name blank will get the root logger. Also, any non-string name will get the root logger.

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.**import_module**(*name*)

idaes.dmf.util.**is_jupyter_notebook**(*filename*)
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*)

idaes.dmf.util.**strlist**(*x*, *sep=', '*)

idaes.dmf.util.**terminate_pid**(*pid*, *waitfor=1*)

idaes.dmf.validate module

class idaes.dmf.validate.**InstanceGenerator**(*schema*, *params=None*)

Bases: object

bplate_div = 'DO NOT MODIFY BEYOND THIS POINT'

create_script(*output_file*, *preserve_old=True*, ***kwargs*)

default_arr_len = 1

get_script(*n=1*, *output_files='/tmp/file{i}.json'*)

Code to load & generate *n* schemas as a Python string template with the spot for the variables as '{variables}'.

Returns

Pair of strings, first is user-modifiable part and second is boilerplate with the template data. This allows separate modification of these 2 sections.

Return type (str, str)

get_template()

Generate a new template for the instance.

Returns JSON of the instance

Return type str

get_variables(*commented=True*)

indent = 2

keywords = ('$schema', 'id', 'definitions')

root_var = 'root'

class idaes.dmf.validate.**JsonSchemaValidator**(*modpath='idaes.dmf'*, *directory='schemas'*, *do_not_cache=False*)

Bases: object

Validate JSON documents against schemas defined in this package.

The schemas are in the "schemas/" directory of the package. They are first processed as Jinja2 templates, to allow for flexible re-use of common schema elements. The actual resulting schema is stored in a temporary directory that is removed when this class is deleted.

Example usage:

```
vdr = JsonSchemaValidator()
# Validate document against the "foobar" schema.
ok, msg = vdr.validate({'foo': '1', 'bar': 2}, 'foobar')
if ok:
    print("Success!")
else:
    print("Failed: {}".format(msg))
# Validate input YAML file against the "config" schema
ok, msg = vdr.validate('/path/to/my_config.yaml', 'config', yaml=True)
if ok:
    print("Success!")
else:
    print("Failed: {}".format(msg))
```

get_schema(*schema*)

Load the schema and return it as a Python (dict) object. See *validate()* for details.

Parameters **schema** (*str*) – Schema name. Same as *schema* arg to *validate()*

Returns Parsed schema

Return type dict

Raises

- IOError if file cannot be opened.
- ValueError if file cannot be parsed.

instances(*schema*, *param_file*)

reset()
Clear cached schemas, so that changes in the base templates are picked up by the validation code.

validate(*doc*, *schema*, *yaml=False*)
Validate a JSON file against a schema.

Parameters

- **doc** (*str|file|list|dict*) – Input filename or object. May be JSON or YAML. Also may be a list/dict, which is assumed to represent parsed JSON.
- **schema** (*str*) – Name of schema in this package. This will be the name, without the *.template* suffix, of a file in the 'schemas/' directory.
- **yaml** (*bool*) – If true, use the YAML parser instead of the JSON parser on the input file.

Returns

(bool, str) Pair whose first value is whether it validated and second is set to the error message if it did not.

Raises

- IOError if either file cannot be opened.
- ValueError if either file cannot be parsed.

idaes.dmf.workspace module

Workspace classes and functions.

class idaes.dmf.workspace.**Fields**
Bases: object

Workspace configuration fields.

DOC_HTML_PATH = 'htmldocs'

LOG_CONF = 'logging'

class idaes.dmf.workspace.**Workspace**(*path*, *create=False*, *add_defaults=False*)
Bases: object

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 DMF 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

description

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.

name

root
Root path for this workspace. This is the path containing the configuration file.

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

class idaes.dmf.workspace.**WorkspaceConfig**
Bases: object

DEFAULTS = {'array': [], 'boolean': False, 'number': 0, 'string': ''}

get_fields(*only_defaults=False*)
Get all possible metadata fields for workspace config.

These values come out of the configuration schema. Keys starting with a leading underscore, like '_id', are skipped.

Parameters **only_defaults** – Only include fields that have a default value in the schema.

Returns

Keys are field name, values are (field description, value). The 'value' gives a default value. Its type is either a list, a number, bool, or a string; the list may be empty.

Return type dict

idaes.dmf.workspace.**find_workspaces**(*root*)
Find workspaces at or below 'root'.

Parameters `root` – Path to start at

Returns List of paths, which are all workspace roots.

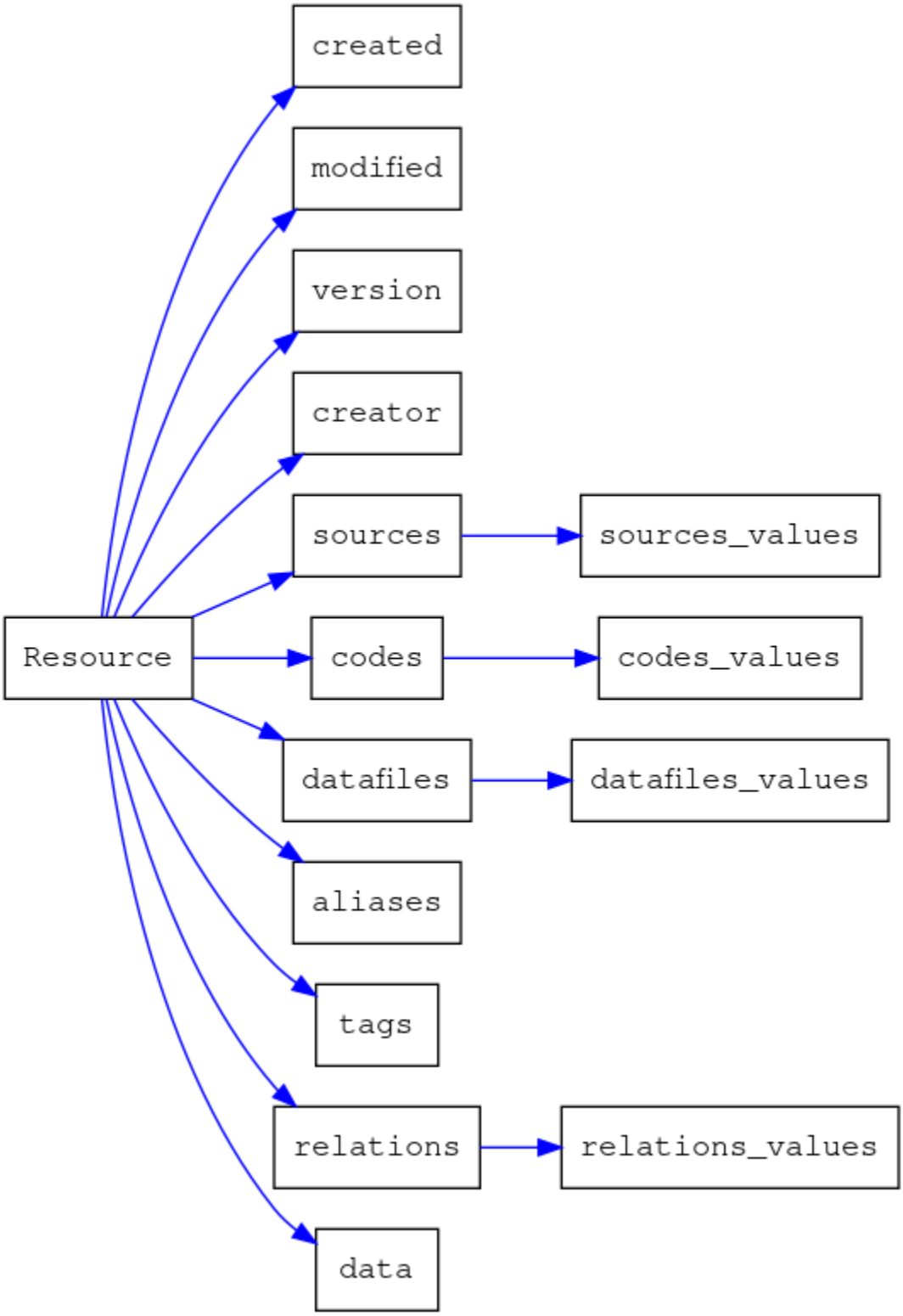
3.3.8 DMF Resources

Data items stored in the DMF, whether they be property data, property models, flowsheets, or other user-defined objects, are called "resources". They are represented in the Python API by a class, *Resource*, with attributes for metadata and a generic "data" section where resource-type-specific information can be held. You store Resources in the DMF with the *add* method on an instance of the *DMF* class. See the *DMF API Examples* documentation for all the available operations.

Resource classes documented on this page: *Resource*, *PropertyDataResource*, *FlowsheetResource*.

A "special" type of resource is an "experiment". The idea of experiments is that they anchor a set of resources that were the result of one coherent set of activities. Experiments are initialized with their associated DMF instance and have some convenience methods to make them a handy way to track a set of resources. See the *experiment documentation page <experiment>* for more details.

Here is a diagram of the resource structure:



Resources are populated by simply setting their attributes. The attributes are actually traits, using the Traitlets library, to make them "smarter" and easier to use. Some of the attributes, such as *creator* or *sources*, are lists of containers with more attributes. All the attributes are documented in the *API Reference* section.

Example

Below is an example of creating and then setting all (well, most) of the attributes for a resource that is a plot of the standard 'iris' dataset.

```
##############################################################################
# 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".
##############################################################################
from idaes.dmf import resource

# Create an empty resource
rsrc = resource.Resource(type='plot')

# Populate the resource. All the attributes are optional.
# This is set automatically to current time
rsrc.created = '2017-10-31'
# This is set automatically to current time
rsrc.modified = '2017-11-23'
# Description of the resource
rsrc.desc = 'Plots of the Iris dataset'
# Version of the resource
rsrc.version = resource.Version(revision='1.0.3-a7',
                                name='november-release')
# Name and contact email of creator of the resource
rsrc.creator = resource.Contact(name='Dan Gunter',
                                email='dang@science-lab.gov')
# Name and contact email of other people involved
rsrc.collaborators = [resource.Contact(name='John Eslick',
                       email='john@science-lab.gov'),
                      resource.Contact(name='David Miller',
                                       email='david@science-lab.gov')]
# Provenance: sources, such as publications -- but
# really anything is allowed -- for the resource.
rsrc.sources = [resource.Source(
             source='R. A. Fisher. "The use of multiple measurements '
                    'in taxonomic problems". '
                    'Annals of Eugenics. 7 (2): 179-188.',
             doi='10.1111/j.1469-1809.1936.tb02137.x', date='1936'),
           resource.Source(source='Edgar Anderson. '
                                  '"The species problem in Iris". '
                                  'Annals of the Missouri Botanical Garden. '
                                  '23 (3): 457-509. '
                                  'JSTOR 2394164',
                           date='1936')]
# Associated code files, including Jupyter notebooks.
# If these codes are worth making into their own Resources,
# you should not put them here, but instead link to them
# through the `relations` attribute.
rsrc.codes = [resource.Code(type='notebook',
```

(continues on next page)

(continued from previous page)

```
                         desc='Python plotting code',
                        name='plot_iris_dataset.ipynb',
                        language='Python',
                        location='http://scikit-learn.org/stable/_downloads/'
                                 'plot_iris_dataset.ipynb')]
# Associated data files. Same comment as for `codes` re: linking.
rsrc.datafiles = [resource.FilePath(path='iris-data.csv',
                                    desc='Iris data')]
# Short names that make it easier to find this resource.
rsrc.aliases = ['iris', 'iris-plots']
# Tags that make it easier to find this resource later.
rsrc.tags = ['iris', 'anderson', 'fisher']
# Arbitrary additional data to add into the resource.
# Note that larger data units should be made into files and
# added into the `datafiles` attributes.
rsrc.data = {
    'features': ['sepal length (cm)', 'sepal width (cm)',
                 'petal length (cm)', 'petal width (cm)'],
    # the data is already defined in the datafile,
    # but we could also put it here, inline
    'array': [[5.1, 3.5, 1.4, 0.2], [4.9, 3.0, 1.4, 0.2],
              [4.7, 3.2, 1.3, 0.2], [4.6, 3.1, 1.5, 0.2],
              [5.0, 3.6, 1.4, 0.2], [5.4, 3.9, 1.7, 0.4],
              # ... you get the idea..
             ]
}
```

API Reference

This section shows the documentation from the source code for the *Resource* and *ResourceType* classes, as well as Python classes that inherit from *Resource* in order to add custom functionality: *PropertyDataResource* and *FlowsheetResource*.

class idaes.dmf.resource.**Resource**(**args*, ***kwargs*)

A dynamically typed resource.

Resources have metadata and (same for all resoures) a type-specific "data" section (unique to that type of resource).

id_

Integer identifier for this Resource. You should not set this yourself. The value will be automatically overwritten with the database's value when the resource is added to the DMF (with the *.add()* method).

uuid

Universal identifier for this resource

type

Type of this Resource. See *ResourceTypes* for standard values for this attribute.

name

Human-readable name for the resource (optional)

desc

Description of the resource

created

Date and time when the resource was created. This defaults to the time when the object was created. Value is a *DateTime*.

modified
Date and time the resource was last modified. This defaults to the time when the object was created. Value is a *DateTime*.

version
Version of the resource. Value is a *SemanticVersion*.

creator
Creator of the resource. Value is a *Contact*.

collaborators
List of other people involved. Each value is a *Contact*.

sources
Sources from which resource is derived, i.e. its provenance. Each value is a *Source*.

codes
List of code objects (including repositories and packages) associated with the resource. Each value is a *Code*.

datafiles
List of data files associated with the resource. Each value is a *FilePath*.

datafiles_dir
Datafiles subdirectory (single directory name)

aliases
List of aliases for the resource

tags
List of tags for the resource

help(*name*)
Return descriptive 'help' for the given attribute.

Parameters **name** (*str*) – Name of attribute

Returns Help string, or error starting with "Error: "

Return type str

static create_relation(*subj*, *pred*, *obj*)
Create a relationship between two Resource instances.

Parameters

- **subj** (Resource) – Subject
- **pred** (*str*) – Predicate
- **obj** (Resource) – Object

Returns None

Raises TypeError – if subject & object are not Resource instances.

copy(***kwargs*)
Get a copy of this Resource.

As a convenience, optionally set some attributes in the copy.

Parameters **kwargs** – Attributes to set in new instance after copying.

Returns: Resource: A deep copy.

The copy will have an empty (zero) *identifier* and a new unique value for *uuid*. The relations are not copied.

table

For tabular data resources, this property builds and returns a Table object.

Returns

A representation of metadata and data in this resource.

Return type *tabular.Table*

Raises `TypeError` – if this resource is not of the correct type.

property_table

For property data resources, this property builds and returns a PropertyTable object.

Returns

A representation of metadata and data in this resource.

Return type *propdata.PropertyTable*

Raises `TypeError` – if this resource is not of the correct type.

class idaes.dmf.resource.**DateTime**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)

A trait type for a datetime.

Input can be a string, float, or tuple. Specifically:

- string, ISO8601: YYYY[-MM-DD[Thh:mm:ss[.uuuuuu]]]
- float: seconds since Unix epoch (1/1/1970)
- tuple: format accepted by datetime.datetime()

No matter the input, validation will transform it into a floating point number, since this is the easiest form to store and search.

class idaes.dmf.resource.**SemanticVersion**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)

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:

- 1 => valid => (1, 0, 0, ‘’)
- rc3 => invalid: no number
- 1.1 => valid => (1, 1, 0, ‘’)
- 1a => valid => (1, 0, 0, ‘a’)
- 1.a.1 => invalid: non-numeric can only go at end
- 1.12.1 => valid => (1, 12, 1, ‘’)
- 1.12.13-1 => valid => (1, 12, 13, ‘1’)

- 1.12.13.x => invalid: too many parts

class idaes.dmf.resource.**Source**(**args*, ***kwargs*)
A work from which the resource is derived.

doi
Digital object identifier

isbn
ISBN

source
The work, either print or electronic, from which the resource was derived

language
The primary language of the intellectual content of the resource

date
Date associated with resource

class idaes.dmf.resource.**Contact**(**args*, ***kwargs*)
Person who can be contacted.

name
Name of the contact

email
Email of the contact

class idaes.dmf.resource.**Code**(**args*, ***kwargs*)
Some source code, such as a Python module or C file.

This can also refer to packages or entire Git repositories.

type
Type of code resource, must be one of – 'method', 'function', 'module', 'class', 'file', 'package', 'repository', or 'notebook'.

desc
Description of the code

name
Name of the code object, e.g. Python module name

language
Programming language, e.g. "Python" (the default).

release
Version of the release, default is '0.0.0'

idhash
Git or other unique hash

location
Flie path or URL location for the code

class idaes.dmf.resource.**FilePath**(*tempfile=False*, *copy=True*, ***kwargs*)
Path to a file, plus optional description and metadata.

So that the DMF does not break when data files are moved or copied, the default is to copy the datafile into the DMF workspace. This behavior can be controlled by the *copy* and *tempfile* keywords to the constructor.

For example, if you have a big file you do NOT want to copy when you create the resource:

```
FilePath(path='/my/big.file', desc='100GB file', copy=False)
```

On the other hand, if you have a file that you want the DMF to manage entirely:

```
FilePath(path='/some/file.txt', desc='a file', tempfile=True)
```

path
Path to file

subdir
Unique subdir

desc
Description of the file's contents

mimetype
MIME type

metadata
Metadata to associate with the file

__init__(*tempfile=False*, *copy=True*, ***kwargs*)
Constructor.

Parameters **tempfile** (*bool*) – if True, when copying the file, remove original.

class idaes.dmf.resource.**RelationType**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)
Traitlets type for RDF-style triples relating resources to each other.

class idaes.dmf.resource.**ResourceTypes**
Standard resource type names.

Use these as opaque constants to indicate standard resource types. For example, when creating a Resource:

```
rsrc = Resource(type=ResourceTypes.property_data, ...)
```

data = 'data'
Data (e.g. result data)

experiment = 'experiment'
Experiment

fs = 'flowsheet'
Flowsheet resource.

nb = 'notebook'
Jupyter Notebook

property_data = 'propertydb'
Property data resource, e.g. the contents are created via classes in the *idaes.dmf.propdata* module.

python = 'python'
Python code

surrmod = 'surrogate_model'
Surrogate model

tabular_data = 'tabular_data'
Tabular data

class idaes.dmf.resource.**PropertyDataResource**(*property_table=None*, ***kwargs*)
Property data resource & factory.

class idaes.dmf.resource.**FlowsheetResource**(**args*, ***kwargs*)
Flowsheet resource & factory.

Resource schema

Below are HTML and raw JSON versions of the resource "schema". This describes the structure of a resource. Having a schema is useful for other programs to be able to independently manipulate the resource representation.

Resource Schema (HTML)

Resouce Schema (JSON)


```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://idaes.org",
  "definitions": {
    "SemanticVersion": {
      "type": "array",
      "items": [
        {
          "type": "integer"
        },
        {
          "type": "integer"
        },
        {
          "type": "integer"
        },
        {
          "type": "string"
        }
      ],
      "minItems": 4
    }
  },
  "type": "object",
  "properties": {
    "aliases": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "codes": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "type": {
            "type": "string",
            "enum": [
              "method",
```


(continues on next page)

(continued from previous page)

```
            "function",
            "module",
            "class",
            "file",
            "package",
            "repository",
            "notebook"
          ]
        },
        "desc": {
          "type": "string"
        },
        "name": {
          "type": "string"
        },
        "language": {
          "type": "string"
        },
        "idhash": {
          "type": "string"
        },
        "location": {
          "type": "string"
        },
        "version": {
          "$ref": "#/definitions/SemanticVersion"
        }
      }
    }
  },
  "collaborators": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "email": {
          "type": "string",
          "format": "email"
        },
        "name": {
          "type": "string"
        }
      },
      "required": [
        "name"
      ]
    }
  },
  "created": {
    "type": "number"
  },
  "creator": {
    "type": "object",
    "properties": {
      "email": {
        "type": "string",
        "format": "email"
```

(continues on next page)

(continued from previous page)

```
        },
        "name": {
          "type": "string"
        }
      },
      "required": [
        "name"
      ]
    },
    "data": {
      "type": "object"
    },
    "datafiles": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "desc": {
            "type": "string"
          },
          "metadata": {
            "type": "object"
          },
          "mimetype": {
            "type": "string"
          },
          "path": {
            "type": "string"
          },
          "subdir": {
            "type": "string"
          }
        },
        "required": [
          "desc",
          "metadata",
          "mimetype",
          "path",
          "subdir"
        ]
      }
    },
    "datafiles_dir": {
      "type": "string"
    },
    "desc": {
      "type": "string"
    },
    "id_": {
      "type": "integer"
    },
    "modified": {
      "type": "number"
    },
    "relations": {
      "type": "array",
      "items": {
```

(continues on next page)

(continued from previous page)

```
        "type": "object",
        "properties": {
          "predicate": {
            "type": "string",
            "enum": [
              "WasGeneratedBy",
              "Used",
              "WasDerivedFrom",
              "WasTriggeredBy",
              "WasControlledBy",
              "WasRevisionOf"
            ]
          },
          "identifier": {
            "type": "string"
          },
          "role": {
            "type": "string",
            "enum": [
              "subject",
              "object"
            ]
          }
        },
        "required": [
          "predicate",
          "identifier",
          "role"
        ]
      }
    },
    "sources": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "date": {
            "type": "number"
          },
          "doi": {
            "type": "string"
          },
          "isbn": {
            "type": "string"
          },
          "language": {
            "type": "string"
          },
          "source": {
            "type": "string"
          }
        }
      }
    },
    "tags": {
      "type": "array",
      "items": {
```

(continues on next page)

(continued from previous page)

```
        "type": "string"
      }
    },
    "type": {
      "type": "string"
    },
    "version": {
      "type": "object",
      "properties": {
        "created": {
          "type": "number"
        },
        "name": {
          "type": "string"
        },
        "revision": {
          "$ref": "#/definitions/SemanticVersion"
        }
      }
    }
  },
  "required": [
    "id_"
  ],
  "additionalProperties": false
}
```

3.3.9 Experiments

Experiments are a type of *resource* intended to be the "anchor" resource that groups other resources into logical units that can be repeated, versioned, etc.

When you create an `Experiment` instance, it is immediately added to the given DMF instance. For example, the following code] adds a new experiment to the DMF workspace at "workspace/path":

```
from idaes.dmf import DMF, experiment
mydmf = DMF(path='workspace/path')
exp = experiment.Experiment(mydmf, name='try1', desc='Nice try')
```

You can create new versions of an experiment with the *`Experiment.copy()`* method:

```
from idaes.dmf import DMF, experiment
from idaes.dmf.resource import R_VERSION
mydmf = DMF(path='workspace/path')
exp1 = experiment.Experiment(mydmf, name='try', version='0.0.1')
exp2 = exp1.copy(version='0.0.2')
# add a relation indicating that exp2 is a revision of exp1
exp1.link(exp2, R_VERSION)
```

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(***kwargs*)
Get a copy of this experiment. The returned object will have been added to the DMF.

Parameters **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*

update()
Update experiment to current values.

remove()
Remove this experiment from the associated DMF instance.

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

3.3.10 DMF Tabular Data

The DMF has some classes and methods specifically for adding tabular data, e.g. data from a spreadsheet. Before importing external data, you will usually convert it to comma-separated values (CSV), although the DMF also has its own internal JSON format that you could use. There are two related modules for handling tabular data. The `tabular` module is designed for any set of labeled columns, whereas the `propdata` module has some additional bells and whistles for IDAES "property" data. In the description that follows, both will be dealt with together, with differences highlighted in the text and notes.

Note: In the Python code, the *propdata.PropertyTable* class *inherits* from the base *tabular.Table* class, meaning that methods expecting a Table can also take a PropertyTable object.

Tabular data can be imported into Python objects from comma-separated values (CSV) files, JSON files, or Python dictionaries. Data files can be merged and the data can be changed through the Python API. From the Python objects,

the data can be exported to CSV or JSON, or saved to the DMF as a *TabularDataResource* (for property data, *PropertyDataResource*).

Steps to get new property data into the DMF:

0. Initialize the DMF (see the "Initialize" section of the *API examples*).
1. Load the data into a *tabular.Table* (for property data, *propdata.PropertyTable*) object, using the import methods for *CSV*, *JSON*, or a *Python dictionary*.
2. Create a *TabularDataResource* (for property data, *PropertyDataResource*), passing the object from the previous step into its constructor.
3. Add the new resource to the DMF with *dmf.DMF.add()*.

Below are more details about importing from different data sources.

Import from CSV

The expected form of CSV data is one or **two** files: one for metadata, the other for the data. The metadata indicates the provenance of the data, *e.g.*, a publication or website.

Note: For property data, the metadata is especially important to identify the associated sources of the data.

Tabular data example

Here is a simple example of importing from CSV to a *Table* object, and from there to the DMF:

```
from idaes_dmf import dmf, resource, tabular

# Note: Working directory is inside the DMF `examples` tree

# Initialize the DMF in the current directory,
# creating '.dmf' tree if necessary.
d = dmf.DMF(create=True)

# Load tabular data
pd = tabular.TabularData.from_csv('wwc_input_liq_vap.csv')

# Load tabular metadata
pm = tabular.Metadata.from_csv('wwc_input_liq_vap_sources.csv')

# Create table with data/metadata combined
tbl = tabular.Table(data=pd, metadata=pm)

# Make tabular resource with table
# This will automatically save the DMF representation to
# a temporary file
rsrc = resource.TabularDataResource(tbl)

# Add the resource to the DMF
# This will copy the temporary file with the converted
# tabular data into the DMF workspace.
# (Note that for very large files this behavior can be
#  overridden to avoid the copy overhead.)
d.add(rsrc)
```

Property data example

This is the same as the example above, but for property data, import using *PropertyTable*:

```
from idaes_dmf import dmf, resource, propdata

# Note: Working directory is inside the DMF `examples` tree

# Initialize the DMF in the current directory,
# creating '.dmf' tree if necessary.
d = dmf.DMF(create=True)

# Load property data
# `12` is number of non-measurement columns, i.e. most of them
pd = propdata.PropertyData.from_csv('wwc_input_liq_vap.csv', 12)

# Load property metadata
# `MEA` is the user-selected "type" of the property data
pm = propdata.PropertyMetadata.from_csv('wwc_input_liq_vap_sources.csv', 'MEA')

# Create PropertyTable from data and metadata
tbl = propdata.PropertyTable(data=pd, metadata=pm)

# Create resource from PropertyTable.
#
# At this point, you can modify the resource metadata to reflect
# links to other objects, aliases, tags, a detailed description, etc.
rsrc = resource.PropertyDataResource(tbl)

# Add the resource to the DMF
d.add(rsrc)
```

CSV Formats

Tabular metadata

The metadata file has two columns, name and value. One name, "Source", is required. All other names are user-defined. Below is an example CSV metadata file (line continuations added for legibility).

```
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"
Notes,r is MEA weight fraction in aqueous soln. (CO2-free basis)
```

This might be exported from a spreadsheet that looks like the following:

Table 3: Sample Property Metadata

Name	Value
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
Notes	r is MEA weight fraction in aqueous soln. (CO2-free basis)

The "Source" will be parsed according to the regular expression in the *source_expr* attribute of *tabular.Metadata*, which by default is just three parts: {authors}, "title", {venue and year}. The year is inferred from a comma followed by optional whitespace and 4 digits.

Tabular data

The data file is a standard CSV file, with a "header" row followed by one or more value rows. The first column is always an identifier, whose value is largely ignored. Columns can either be interpreted as values or pairs of `<value column>,<error column>`. The default behavior differs depending on whether you are importing a generic table or property data:

- For a generic table, the default is to see each column as a value. You can pass "error_column=True" to change this.
- For property data, error columns are the default. You need to pass error_column=False to change this.

Note: For property data, some number of initial column pairs are called "states", which means they are the independent variables, and the remaining columns are called "properties", which are dependent on the values for the states.

Tabular CSV data example

Thus, the overall layout of the file is like this, for generic tabular data:

```
id, value1, value2, ...

OR

id, value1, error1, value2, error2, ...
```

Property CSV data example

The overall layout is like this, for property data:

```
id, state-value1, state-error1, state-value2, state-error2, ..., \
    prop-value1, prop-error1, prop-value2, prop-error2, ...
```

For example, here is a property data CSV with two data rows, and two columns: one state value and one property value column.

```
Data No.,T (K),Absolute Error,Density (g/cm3),Absolute Error
1,303.15,0,0.2,0
2,304.15,0,0.3,0
```

And in a spreadsheet, this example may look like this:

Table 4: Sample Property Data

Data No.	T (K)	Absolute Error	Density (g/cm3)	Absolute Error
1	303.15	0	0.2	0
2	304.15	0	0.3	0

The format for the header columns is:

For state/property value columnns `<value name> (<units>)`, where the units are optional but recommended. Example: "Density (g/cm3)".

For state/property error columns `<type of error> Error`. Example: "Absolute Error".

Import from JSON

The JSON representation of the tabular data, unlike the CSV representation, combines the data and metadata in a single file.

JSON import example

Here is a simple example of importing tabular data (or property data) from JSON and adding to the DMF:

```
from idaes_dmf import dmf, resource, propdata, tabular

# Note: Working directory is inside the DMF `examples` tree

# Initialize the DMF in the current directory,
# creating '.dmf' tree if necessary.
d = dmf.DMF(create=True)

# Create generic Table from JSON file
tbl = tabular.Table.load(open('wwc_input_liq_vap.json'))
#
# OR
#
# Create PropertyTable from (same) JSON file
tbl2 = propdata.PropertyTable.load(open('wwc_input_liq_vap.json'))

# Create resource from PropertyTable.
#
# At this point, you can modify the resource metadata to reflect
# links to other objects, aliases, tags, a detailed description, etc.
rsrc = resource.PropertyDataResource(tbl)

# Add the resource to the DMF
d.add(rsrc)
```

Note: Because the JSON format explicitly labels the "states" and "properties" columns, and separates error values from non-error vallues, the only real difference between importing as PropertyTable and a tabular.Table is the type of the objects that are returned.

JSON format

The "schema" of the JSON format is shown at the end of this section. But, since nobody likes reading schemas, here is a JSON version of the CSV *metadata* and *data* given above (truncated to six rows).

```
{
  "data": [
```

(continues on next page)

(continued from previous page)


```
    {
      "name": "T",
      "units": "K",
      "values": [
        298.15,
        298.15,
        298.15,
        298.15,
        298.15,
        313.15,
      ],
      "errors": [
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
      ],
      "error_type": "absolute",
      "type": "property"
    },
    {
      "name": "CO2 Loading",
      "units": "mol CO2/MEA",
      "values": [
        0.1,
        0.21,
        0.32,
        0.44,
        0.56,
        0.1,
      ],
      "errors": [
        0.0,
        0.0,
        0.0,
        0.0,
        0.0,
      ],
      "error_type": "absolute",
      "type": "property"
    },
    {
      "name": "r",
      "units": "",
      "values": [
        0.3,
        0.3,
        0.3,
        0.3,
        0.3,
        0.3,
      ],
      "errors": [
        0.0,
        0.0,
```


(continues on next page)

(continued from previous page)


```
        0.0,
        0.0,
        0.0,
        0.0,
      ],
      "error_type": "absolute",
      "type": "property"
    },
    {
      "name": "Density Data",
      "units": "g/cm3",
      "values": [
        1.0333,
        1.0534,
        1.0756,
        1.0964,
        1.1142,
        1.0253,
      ],
      "errors": [
        5e-05,
        5e-05,
        5e-05,
        5e-05,
        5e-05,
        5e-05,
      ],
      "error_type": "absolute",
      "type": "property"
    }
  ],
  "meta": [
    {
      "authors": "Han, J., Jin, J., Eimer, D.A., Melaaen, M.C.",
      "date": "1970-01-01",
      "title": "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",
      "info": "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)"
    }
  ]
}
```


Import from Python dictionary

Creating a new table from a Python dictionary is similar to importing from JSON, except in this case the data and metadata dicts are created separately from Python dicts, and then combined.

Import from dictionary example

The import API is shown in the following example:

```
from idaes_dmf import dmf, resource, tabular, propdata

 # Note: Working directory is inside the DMF `examples` tree

 # Initialize the DMF in the current directory,
 # creating '.dmf' tree if necessary.
 d = dmf.DMF(create=True)

 # Create PropertyTable from two inputs:
 #  - `data` is a dict in the same format as the 'data' secion of the JSON
 #    schema
 #  - `metadata` is a list of metadata dicts, just like the `meta`
 #    section of the JSON schema.
 tbl = tabular.Table(data=data, metadata=metadata)
 #
 # OR, for property data:
 tbl = propdata.PropertyTable(data=data, metadata=metadata)

 # Add the property data resource to the DMF
 d.add(resource.PropertyDataResource(tbl))
```

JSON Schemas

Schema (HTML version)

Schema (raw JSON version)

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://idaes.org",
  "definitions": {
    "Column": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "examples": [
            "Density",
            "r"
          ]
        },
        "units": {
          "type": "string",
          "examples": [
            "mPa-s",
            "K"
          ]
        },
        "values": {
          "description": "Column of numeric values",
          "type": "array",
          "items": {
            "type": "number"
          },
```

(continues on next page)

(continued from previous page)

```
          "examples": [
            "[2.6, 6.21]"
          ]
        },
        "error_type": {
          "description": "Type for error values",
          "type": "string"
        },
        "errors": {
          "description": "Column of numeric errors",
          "type": "array",
          "items": {
            "type": "number"
          },
          "examples": [
            "[0.001, 0.035]"
          ]
        },
        "type": {
          "description": "Type of column",
          "enum": [
            "state",
            "property"
          ]
        }
      },
      "required": [
        "name",
        "units",
        "values"
      ],
      "additionalProperties": false
    },
    "Metadata": {
      "type": "object",
      "properties": {
        "datatype": {
          "description": "name of the data type",
          "type": "string",
          "examples": [
            "MEA"
          ]
        },
        "info": {
          "description": "Additional information about the source (i.e. publication)",
          "type": "string",
          "examples": [
            "J. Chem. Eng. Data, 2009, Vol 54, pg. 3096-30100"
          ]
        },
        "notes": {
          "description": "Free-form text with notes about the data",
          "type": "string",
          "examples": [
            "r is MEA weight fraction in aqueous soln."
          ]
        },
```

(continues on next page)

(continued from previous page)

```
        "authors": {
          "description": "Author list in format Last1, First1, Last2, First2, etc.",
          "type": "string",
          "examples": [
            "Amundsen, T.G., Lars, E.O., Eimer, D.A."
          ]
        },
        "title": {
          "description": "Title of the source (e.g. publication title)",
          "type": "string",
          "examples": [
            "Density and Viscosity of Monoethanolamine + .etc."
          ]
        },
        "date": {
          "description": "Date of source data",
          "type": "string",
          "examples": [
            "2009"
          ]
        }
      },
      "required": [
        "datatype",
        "authors",
        "title",
        "date"
      ],
      "additionalProperties": true
    }
  },
  "type": "object",
  "properties": {
    "meta": {
      "description": "List of information about the data source",
      "type": "array",
      "items": {
        "$ref": "#/definitions/Metadata"
      }
    },
    "data": {
      "description": "Measured data columns",
      "type": "array",
      "items": {
        "$ref": "#/definitions/Column"
      }
    }
  },
  "required": [
    "meta",
    "data"
  ],
  "additionalProperties": false
}
```

3.3.11 DMF Help

The DMF `help` function provides a link between objects that the user is working with in a Jupyter Notebook and the detailed documentation pages that you have created for those objects in Sphinx.

Creating help pages

The mechanism is relatively simple, once you have a documentation page already created. Simply add special "index" entries at the appropriate place in the page, with the following format:

```
.. index::
   pair: full.path.to.your.module;ClassName
```

For example, if you were to add help for the *Port* class of the *idaes.core.ports*, you could add the following to the page:

```
.. index::
   pair: idaes.core.ports;Port
```

You can have multiple index entries for the same page.

DMF Help Configuration

The DMF finds the help pages by searching a user-configured set of paths that point to the generated (note: *not* the source) HTML documentation. This information goes under the `htmldocs` keyword in the DMF configuration file (found in `<workspace_root>/config.yaml`).

Using the special key `{dmf_root}` to indicate the root of the DMF installation, the default configuration is:

```
htmldocs:
- '{dmf_root}/doc/build/html/dmf'
- '{dmf_root}/doc/build/html/models'
```

Sensible values for these documentation paths are automatically generated by the *dmf init* command, and most users will not need to modify this. Note that these paths are searched in the order given, and the first match is the one used.

DMF Help Usage

In the Jupyter notebook, the `%dmf help <object-or-class>` magic will invoke the help functionality. If a page is found, it will be automatically shown in a new window or tab of the browser. See *this Jupyter notebook* for an example of using the `%dmf help` magics.

3.3.12 Example Jupyter Notebooks

The Jupyter Notebook is a web-based interactive Python (and, actually, other languages as well) environment that can mix rich text, graphics, and running Python code. It is used widely within IDAES as a tool for performing and sharing analyses.

This page provides links to some example notebooks, rendered in their HTML version with the Jupyter Notebook viewer. You can run the original form of the notebook from the "examples" directory in the DMF source code, with a command like:

```
# if necessary setup your virtual environment first
# to include the installed DMF code, e.g.:
# 'source activate dmf_dev' for Anaconda and an env named 'dmf_dev'

cd examples
jupyter notebook <name-of-notebook>.ipynb
```

DMF Properties API example

July 30, 2017 - Dan Gunter dkgunter@lbl.gov

This example shows usage of the DMF "Properties" API in the DMF.

```
# Standard library imports
import json
import os
import tempfile
```

```
# Third-party imports
import pandas as pd
import numpy as np
```

Initialize the DMF

The DMF is initialized with a root directory, called the "workspace". For this example, we'll use a temporary directory that we create.

```
from idaes_dmf import dmf
from idaes_dmf import propdata
```

```
tmpdir = tempfile.mkdtemp()
mydmf = dmf.DMF(tmpdir, create=True)
```

Create property data from Python dictionary

Create some inline property data and write it to a temporary file. This file could have been created in other ways, e.g. by using Python or CLI functions to convert the data from comma-separated values.

```
amundsen_properties = {
                "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.",
                    "date": "2009"
                },
                "data": {
                    "properties" : [
                        {
                            "name": "Viscosity Value",
```

(continues on next page)

(continued from previous page)

```
                            "units": "mPa-s",
                            "values": [2.6, 6.2],
                            "error_type": "absolute",
                            "errors": [0.06, 0.004]
                        }
                    ],
                    "states": [
                        {
                            "name": "r",
                            "units": "",
                            "values": [0.2, 1000]
                        }
                    ]
                }
            }
```

```
# Make temporary directory in which to write the file
scratchdir = os.path.join(mydmf.root, 'scratch')
if not os.path.exists(scratchdir):
    os.mkdir(scratchdir)
# Dump the Python dictionary as JSON to the file
tmpf = open(os.path.join(scratchdir, 'resource.json'), 'w')
tmpf.write(json.dumps(amundsen_properties))
tmpf.close()
```

Load property data into DMF

The newly created property data is just a file, that could be anywhere on the file system. So the DMF can locate it and manipulate it, we need to explicitly "add" it.

```
# Add the resource
factory = dmf.ResourceFactory()
resource = factory.create_propertydb(tmpf.name)
rid = mydmf.add(resource, copy_files=(tmpf.name,))
```

Get the property data as an object

Now that the property data is in the DMF, we can retrieve it as an object. Since we just loaded it, we can take a shortcut and "get" the data using the resource ID returned by the "add_resource" method.

```
r2 = mydmf.fetch_one(rid)
table = dmf.get_propertydb_table(r2)
table.data.values_dataframe()
```

Extract values and errors as Pandas Dataframes

After extracting each dataframe, merge them together.

```
vf = table.data.values_dataframe()
ef = table.data.errors_dataframe()
vf.join(ef, rsuffix='.err').head()
```

The 'states=False' parameter means we only see measured properties, not the associated states – which in this case is temperature.

```
vf = table.data.values_dataframe(states=False)
ef = table.data.errors_dataframe(states=False)
vf.join(ef, rsuffix='.err').head()
```

Create property data from CSV

The DMF properties classes also understand how to convert data from specifically formatted CSV files into the JSON format shown above. Once that is done, the process for importing that data into the DMF is the same. In this example, we'll start with some example CSV data, which we write to a file and then convert before importing to the DMF.

Han properties as CSV

The CSV format has two files, one for "metadata" and one for "data".

```
han_meta = '''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)
'''
han_data = '''Data No.,T (K),Absolute Error,CO2 Loading (mol CO2/MEA),Absolute Error,
↪r,Absolute Error,Density Data (g/cm3),Absolute Error
1,298.15,0,0.1,0,0.3,0,1.0333,5E-05
2,298.15,0,0.21,0,0.3,0,1.0534,5E-05
3,298.15,0,0.32,0,0.3,0,1.0756,5E-05
4,298.15,0,0.44,0,0.3,0,1.0964,5E-05
5,298.15,0,0.56,0,0.3,0,1.1142,5E-05
6,313.15,0,0.1,0,0.3,0,1.0253,5E-05
7,313.15,0,0.21,0,0.3,0,1.0464,5E-05
8,313.15,0,0.32,0,0.3,0,1.0669,5E-05
9,313.15,0,0.44,0,0.3,0,1.0891,5E-05
10,313.15,0,0.56,0,0.3,0,1.1068,5E-05
11,323.15,0,0.1,0,0.3,0,1.0196,5E-05
12,323.15,0,0.21,0,0.3,0,1.0412,5E-05
13,323.15,0,0.32,0,0.3,0,1.0613,5E-05
14,323.15,0,0.44,0,0.3,0,1.0838,5E-05
15,323.15,0,0.56,0,0.3,0,1.1014,5E-05
16,333.15,0,0.1,0,0.3,0,1.0138,5E-05
17,333.15,0,0.21,0,0.3,0,1.0356,5E-05
18,333.15,0,0.32,0,0.3,0,1.0556,5E-05
19,333.15,0,0.44,0,0.3,0,1.0782,5E-05
20,333.15,0,0.56,0,0.3,0,1.0957,5E-05
21,343.15,0,0.1,0,0.3,0,1.0076,5E-05
22,343.15,0,0.21,0,0.3,0,1.0297,5E-05
23,343.15,0,0.32,0,0.3,0,1.0496,5E-05
24,343.15,0,0.44,0,0.3,0,1.0723,5E-05
25,343.15,0,0.56,0,0.3,0,1.0887,5E-05
26,353.15,0,0.1,0,0.3,0,1.0002,5E-05
27,353.15,0,0.21,0,0.3,0,1.0234,5E-05
28,353.15,0,0.32,0,0.3,0,1.0434,5E-05
```

(continues on next page)

(continued from previous page)

```
29,353.15,0,0.44,0,0.3,0,1.066,5E-05
30,353.15,0,0.56,0,0.3,0,1.0812,5E-05
31,298.15,0,0.1,0,0.4,0,1.0376,5E-05
32,298.15,0,0.21,0,0.4,0,1.0627,5E-05
33,298.15,0,0.33,0,0.4,0,1.0945,5E-05
34,298.15,0,0.45,0,0.4,0,1.1296,5E-05
35,313.15,0,0.1,0,0.4,0,1.0295,5E-05
36,313.15,0,0.21,0,0.4,0,1.0547,5E-05
37,313.15,0,0.33,0,0.4,0,1.0867,5E-05
38,313.15,0,0.45,0,0.4,0,1.1199,5E-05
39,323.15,0,0.1,0,0.4,0,1.0237,5E-05
40,323.15,0,0.21,0,0.4,0,1.049,5E-05
41,323.15,0,0.33,0,0.4,0,1.0811,5E-05
42,323.15,0,0.45,0,0.4,0,1.1138,5E-05
43,333.15,0,0.1,0,0.4,0,1.0178,5E-05
44,333.15,0,0.21,0,0.4,0,1.043,5E-05
45,333.15,0,0.33,0,0.4,0,1.0752,5E-05
46,333.15,0,0.45,0,0.4,0,1.1087,5E-05
47,343.15,0,0.1,0,0.4,0,1.011,5E-05
48,343.15,0,0.21,0,0.4,0,1.0367,5E-05
49,343.15,0,0.33,0,0.4,0,1.0686,5E-05
50,343.15,0,0.45,0,0.4,0,1.1032,5E-05
51,353.15,0,0.1,0,0.4,0,1.0048,5E-05
52,353.15,0,0.21,0,0.4,0,1.0292,5E-05
53,353.15,0,0.33,0,0.4,0,1.0626,5E-05
54,353.15,0,0.45,0,0.4,0,1.0963,5E-05
'''
```

```
# use 'scratchdir' from above to write the files,
# saving the names of these files in some local variables
prop_meta = os.path.join(scratchdir, 'prop-meta.csv')
open(prop_meta, 'w').write(han_meta)
prop_data = os.path.join(scratchdir, 'prop-data.csv')
open(prop_data, 'w').write(han_data)
```

```
2151
```

Convert from CSV to JSON

A single convenience function can convert the properties data from CSV to JSON. In addition to the files, it needs two parameters:

- The "type" of the data, which is a free-form string such as "MEA" to identify what kinds of property data this is. Right now, the vocabulary, etc. of this string is outside the scope of the DMF code.
- The number of initial columns of data (value,error = one "column") that are state variables (as opposed to measured variables). This information is necessary for good display of the data, as well as merging data from multiple sources. In the case of the dataset here, the first 3 values are "states" and only the last one (Density) is measured.

```
from idaes_dmf import propdata
output_file = os.path.join(scratchdir, 'prop.json')
propdata.convert_csv(prop_meta, 'MEA', prop_data, 3, output_file)
jdata = json.load(open(output_file))
println(json.dumps(jdata, indent=2))
```

```
{
  "data": {
    "properties": [
      {
        "name": "Density Data",
        "units": "g/cm3",
        "values": [
          1.0333,
          1.0534,
          1.0756,
          1.0964,
          1.1142,
          1.0253,
          1.0464,
          1.0669,
          1.0891,
          1.1068,
          1.0196,
          1.0412,
          1.0613,
          1.0838,
          1.1014,
          1.0138,
          1.0356,
          1.0556,
          1.0782,
          1.0957,
          1.0076,
          1.0297,
          1.0496,
          1.0723,
          1.0887,
          1.0002,
          1.0234,
          1.0434,
          1.066,
          1.0812,
          1.0376,
          1.0627,
          1.0945,
          1.1296,
          1.0295,
          1.0547,
          1.0867,
          1.1199,
          1.0237,
          1.049,
          1.0811,
          1.1138,
          1.0178,
          1.043,
          1.0752,
          1.1087,
          1.011,
          1.0367,
          1.0686,
          1.1032,
```


(continues on next page)

(continued from previous page)

```
          1.0048,
          1.0292,
          1.0626,
          1.0963
        ],
        "errors": [
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
          5e-05,
```

(continues on next page)

(continued from previous page)


```
          5e-05,
          5e-05,
          5e-05
        ],
        "error_type": "absolute"
      }
    ],
    "states": [
      {
        "name": "T",
        "units": "K",
        "values": [
          298.15,
          298.15,
          298.15,
          298.15,
          298.15,
          313.15,
          313.15,
          313.15,
          313.15,
          313.15,
          323.15,
          323.15,
          323.15,
          323.15,
          323.15,
          333.15,
          333.15,
          333.15,
          333.15,
          333.15,
          343.15,
          343.15,
          343.15,
          343.15,
          343.15,
          353.15,
          353.15,
          353.15,
          353.15,
          353.15,
          298.15,
          298.15,
          298.15,
          298.15,
          313.15,
          313.15,
          313.15,
          313.15,
          323.15,
          323.15,
          323.15,
          323.15,
          323.15,
          333.15,
          333.15,
          333.15,
```


(continues on next page)

(continued from previous page)

```
            333.15,
            343.15,
            343.15,
            343.15,
            343.15,
            353.15,
            353.15,
            353.15,
            353.15
          ]
        },
        {
          "name": "CO2 Loading",
          "units": "mol CO2/MEA",
          "values": [
            0.1,
            0.21,
            0.32,
            0.44,
            0.56,
            0.1,
            0.21,
            0.32,
            0.44,
            0.56,
            0.1,
            0.21,
            0.32,
            0.44,
            0.56,
            0.1,
            0.21,
            0.32,
            0.44,
            0.56,
            0.1,
            0.21,
            0.32,
            0.44,
            0.56,
            0.1,
            0.21,
            0.33,
            0.45,
            0.1,
            0.21,
            0.33,
            0.45,
            0.1,
            0.21,
            0.33,
            0.45,
```

(continues on next page)

(continued from previous page)


```
          0.1,
          0.21,
          0.33,
          0.45,
          0.1,
          0.21,
          0.33,
          0.45,
          0.1,
          0.21,
          0.33,
          0.45
        ]
      },
      {
        "name": "r",
        "units": "",
        "values": [
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.3,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
```


(continues on next page)

(continued from previous page)

```
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4,
          0.4
        ]
      }
    ]
  },
  "meta": {
    "datatype": "MEA",
    "authors": "Han, J., Jin, J., Eimer, D.A., Melaaen, M.C.",
    "date": 2012,
    "title": "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
↪",
    "info": "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)"
  }
}
```

Load PropertyTable

Once there is a standard JSON version of the data, you can simply load it back into a PropertyTable

```
table2 = propdata.PropertyTable.load(output_file)
```

Show the data

As before, we extract the data to a Pandas dataframe so it is easy to view in the notebook.

```
vf = table2.data.values_dataframe()
ef = table2.data.errors_dataframe()
vf.join(ef, rsuffix='.err').head(10)
```

DMF help example

```
# Import DMF
from idaes_dmf import dmf, util, magics
```

```
from idaes_model_contrib.mea_simple.flowsheet.flowsheet import MeaSheet
```

```
%dmf init ../WORKSPACE
```

You put the doc locations in the configuration

The paths can be relative to the location of the DMF "workspace", or to where the code is installed, or absolute paths. If they are wrong, then of course the DMF will have trouble finding the documentation.

```
%dmf info
```

Configuration

- created: 2017-08-11T17:52:16.350098
- htmldocs:
  - /home/dang/src/idaes/dangunter/model_contrib/doc/_build/html
  - /home/dang/src/idaes/dangunter/models/docs/html
  - /home/dang/src/idaes/dangunter/DMF/docs/build/html
- modified: 2017-08-11T17:52:16.350098
- property_data: resources/index.sqlite

You can get help on the class

```
%dmf help MeaSheet
```

You can also get help on instances

This takes you the same page you would get for the class. Note that the warning comes from a half-baked initialization of `MeaSheet()`.

```
mm = MeaSheet()

%dmf help mm
```

```
/home/dang/anaconda3/envs/idaes_dev/lib/python3.6/site-packages/idaes_models-0.1.0-
↪py3.6.egg/idaes_models/core/process_base.py:957: UserWarning: Component set␣
↪unspecified for unit Unnamed_Flowsheet
  'Component set unspecified for unit {}'.format(self.unit_name))
```

You can also get help on the DMF itself

```
%dmf help dmf
```

How does it work? Let's see the help page!

```
%dmf help help
```

Property Data from CSV

```
from idaes_dmf import dmf, propdata, resource
```

```
ls *.csv
```

```
amundsen-data.csv  han-data.csv  jayarathna-data.csv
amundsen-meta.csv  han-meta.csv  jayarathna-meta.csv
```

Load CSV files into a PropertyTable object

```
# load data
data3 = propdata.PropertyData.from_csv("han-data.csv", 2)
data3.add_csv("jayarathna-data.csv")
data3.add_csv("amundsen-data.csv")
table3 = propdata.PropertyTable(data=data3)
# add metadata (sources)
for f in "han-meta.csv", "jayarathna-meta.csv", "amundsen-meta.csv":
    table3.add_metadata(propdata.PropertyMetadata.from_csv(f, "MEA"))
```

Add PropertyTable object, containing all the property data, to the DMF

```
d = dmf.DMF()
```

```
rsrc = resource.PropertyDataResource(property_table=table3)
```

```
ident = d.add(rsrc)
println(ident)
```

```
4
```

```
d2 = d.fetch_one(ident)
```

```
data3 = d2.property_table
```

```
data3.data.values_dataframe()
```

3.3.13 Glossary

DMF Data Management Framework. This is the overarching component.

data files Files, text or binary, that are tracked by the DMF by being associated with one or more *resource*s.

flowsheet An object that encapsulates the connections between models to create a meaningful end to end description of a process.

property data A set of values relating to measured properties for chemical compounds or other components of interest. For example, pressure and temperature data.

property model A model, usable in *Pyomo*, for calculating property values based on *property data*.

Pyomo Python-based, open-source optimization modeling language used to perform all the underlying computation for the models. See the Pyomo website for more details.

relation A connection between two resources. These connections have labels, such as "contains" or "derivedfrom". They can be expressed as triples of the form (subject, predicate, object), which is not coincidentally the basic unit of the Resource Description Framework (RDF), a web data interchange standard.

resource A distinct object stored in the DMF, e.g., a unit model, flowsheet, property data package. Resources have a unique identifier, and may connect to each other.

unit model A Pyomo "block" that models one unit in a flowsheet.

workspace The container for a set of DMF resources and files. Associated with a configuration file. All DMF actions occur within a single workspace.

3.3.14 How to use this documentation

The goal of this documentation is to provide a guide and reference to using the DMF, for all kinds of users. It is organized into four major sections:

- Getting started: First steps in installing and configuring the DMF
- Topic guides: Step-by-step guides on using specific features of the DMF.
- Reference: Detailed reference for the Python API and command-line tools, as well as any data formats or schemas.
- Example notebooks: A collection of example Jupyter notebooks. These are also cross-referenced from the topic guides.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

idaes.core, 146
idaes.core.flowsheet_model, 162
idaes.core.holdup, 162
idaes.core.plugins, 146
idaes.core.plugins.pyosyn, 146
idaes.core.ports, 172
idaes.core.process_base, 175
idaes.core.process_block, 176
idaes.core.property_base, 177
idaes.core.stream, 178
idaes.core.unit_model, 180
idaes.core.util, 146
idaes.core.util.compare, 150
idaes.core.util.concave, 151
idaes.core.util.config, 151
idaes.core.util.convergence, 146
idaes.core.util.convergence.convergence, 146
idaes.core.util.convergence.convergence_base, 147
idaes.core.util.convergence.mpi_utils, 150
idaes.core.util.convex, 151
idaes.core.util.cut_gen, 152
idaes.core.util.debug, 152
idaes.core.util.expr, 153
idaes.core.util.initialization, 153
idaes.core.util.mccormick, 153
idaes.core.util.misc, 154
idaes.core.util.model_serializer, 157
idaes.core.util.mpdisagg, 159
idaes.core.util.stream, 160
idaes.core.util.var, 160
idaes.core.util.var_test, 161
idaes.dmf, 198
idaes.dmf.commands, 198
idaes.dmf.dmf, 199
idaes.dmf.errors, 202
idaes.dmf.experiment, 203
idaes.dmf.help, 204
idaes.dmf.magics, 204
idaes.dmf.propdata, 205
idaes.dmf.resource, 208
idaes.dmf.resourcedb, 215
idaes.dmf.schemas, 198
idaes.dmf.surrmod, 216
idaes.dmf.tabular, 217
idaes.dmf.util, 221
idaes.dmf.validate, 223
idaes.dmf.workspace, 224
idaes.models.cstr, 126
idaes.models.equilibrium_reactor, 130
idaes.models.feed, 100
idaes.models.flash, 123
idaes.models.gibbs_reactor, 132
idaes.models.heat_exchanger, 117
idaes.models.heat_exchanger_1D, 120
idaes.models.mixer, 104
idaes.models.pfr, 128
idaes.models.pressure_changer, 111
idaes.models.product, 102
idaes.models.separator, 108
idaes.models.splitter, 106
idaes.models.stoichiometric_reactor, 124
idaes.models.temperature_changer, 114
idaes.models.translator, 133
idaes.vis.plot_utils, 139
idaes_models.core.flowsheet_model, 183
idaes_models.core.process_base, 184
idaes_models.core.unit_model, 183
idaes_models.core.util.model_serializer, 95
idaes_models.process.conceptd.water.flowsheet, 184
idaes_models.process.conceptd.water.main, 184
idaes_models.unit.water_net.feed, 184
idaes_models.unit.water_net.reactor, 184
idaes_models.unit.water_net.sink, 184

INDEX

A

activate() (idaes.core.stream.StreamData method), 59, 178
add() (idaes.dmf.dmf.DMF method), 199
add() (idaes.dmf.experiment.Experiment method), 203
add_concave_linear_underest() (in module idaes.core.util.concave), 151
add_concave_relaxation() (in module idaes.core.util.concave), 151
add_convex_relaxation() (in module idaes.core.util.convex), 151
add_csv() (idaes.dmf.propdata.PropertyData method), 205
add_exchanger_labels() (in module idaes.vis.plot_utils), 139
add_mccormick_cut() (in module idaes.core.util.mccormick), 153
add_mccormick_relaxation() (in module idaes.core.util.mccormick), 153
add_metadata() (idaes.dmf.tabular.Table method), 218
add_module_markers_to_heat_exchanger_plot() (in module idaes.vis.plot_utils), 139
add_mpDisagg_cut() (in module idaes.core.util.mpdisagg), 160
add_object_ref() (in module idaes.core.util.misc), 96, 154
add_sampled_input() (idaes.core.util.convergence.convergence_base.ConvergenceEvaluationSpecification method), 148
AddedCSVColumnError, 205
AlamoDisabledError, 202
AlamoError, 202
aliases (idaes.dmf.resource.Resource attribute), 210
allgather_global_data() (idaes.core.util.convergence.mpi_utils.ParallelTaskManager method), 150
annotate() (idaes.vis.plot.Plot method), 135
as_arr() (idaes.dmf.propdata.PropertyData method), 206
as_arr() (idaes.dmf.tabular.TabularData method), 220
as_dict() (idaes.dmf.resource.TraitContainer method), 214
as_dict() (idaes.dmf.tabular.Metadata method), 218
as_dict() (idaes.dmf.tabular.Table method), 218
as_dict() (idaes.dmf.tabular.TabularObject method), 221
as_list() (idaes.dmf.tabular.TabularData method), 220
assert_var_equal() (in module idaes.core.util.var_test), 161
AUTH (idaes.dmf.tabular.Fields attribute), 217
author (idaes.dmf.tabular.Metadata attribute), 218

B

BadResourceError, 202
base_class_module() (idaes.core.process_block.ProcessBlock class method), 94, 176
base_class_name() (idaes.core.process_block.ProcessBlock class method), 94, 176
bound() (idaes.core.util.model_serializer.StoreSpec class method), 158
bplate_div (idaes.dmf.validate.InstanceGenerator attribute), 223
build() (idaes.core.flowsheet_model.FlowsheetBlockData method), 53
build() (idaes.core.holdup.Holdup0dData method), 69, 165
build() (idaes.core.holdup.Holdup1dData method), 77, 168
build() (idaes.core.holdup.HoldupData method), 64, 169
build() (idaes.core.holdup.HoldupStaticData method), 83, 171
build() (idaes.core.ports.InletMixerData method), 87, 172
build() (idaes.core.ports.OutletSplitterData method), 89, 174
build() (idaes.core.ports.Port method), 85, 175
build() (idaes.core.process_base.ProcessBlockData method), 93, 175
build() (idaes.core.property_base.PropertyBlockDataBase method), 92, 177
build() (idaes.core.property_base.PropertyParameterBase method), 91, 177
build() (idaes.core.stream.StreamData method), 59, 178
build() (idaes.core.unit_model.UnitBlockData method), 56, 180
build() (idaes.models.cstr.CSTRData method), 126
build() (idaes.models.equilibrium_reactor.EquilibriumReactorData method), 130
build() (idaes.models.feed.FeedData method), 100
build() (idaes.models.flash.FlashData method), 123

build() (idaes.models.gibbs_reactor.GibbsReactorData method), 132
build() (idaes.models.heat_exchanger.HeatExchangerData method), 117
build() (idaes.models.heat_exchanger_1D.HeatExchanger1dData method), 120
build() (idaes.models.mixer.MixerData method), 104
build() (idaes.models.pressure_changer.PressureChangerData method), 111
build() (idaes.models.product.ProductData method), 102
build() (idaes.models.separator.SeparatorData method), 108
build() (idaes.models.splitter.SplitterData method), 106
build() (idaes.models.stoichiometric_reactor.StoichiometricReactorData method), 124
build() (idaes.models.temperature_changer.TemperatureChangerData method), 114
build() (idaes.models.translator.TranslatorData method), 133
build_inlets() (idaes.core.unit_model.UnitBlockData method), 56, 180
build_outlets() (idaes.core.unit_model.UnitBlockData method), 56, 181

C

C_PROP (idaes.dmf.propdata.Fields attribute), 205
C_STATE (idaes.dmf.propdata.Fields attribute), 205
cat_resources() (in module idaes.dmf.commands), 198
category() (in module idaes.core.util.misc), 96, 154
clone_block() (in module idaes.core.util.cut_gen), 152
clone_block_constraints() (in module idaes.core.util.cut_gen), 152
clone_block_params() (in module idaes.core.util.cut_gen), 152
clone_block_sets() (in module idaes.core.util.cut_gen), 152
clone_block_vars() (in module idaes.core.util.cut_gen), 152
Code (class in idaes.dmf.resource), 208
codes (idaes.dmf.resource.Resource attribute), 210
collaborators (idaes.dmf.resource.Resource attribute), 211
colorize() (idaes.dmf.util.CPrint method), 221
COLORS (idaes.dmf.util.CPrint attribute), 221
COLTYPE (idaes.dmf.tabular.Fields attribute), 217
Column (class in idaes.dmf.tabular), 217
columns (idaes.dmf.tabular.TabularData attribute), 220
comm (idaes.core.util.convergence.mpi_utils.MPIInterface attribute), 150
CommandError, 202
compare() (in module idaes.core.util.compare), 150
compare_block() (in module idaes.core.util.compare), 150
compare_var() (in module idaes.core.util.compare), 150
component_data_from_dict() (in module idaes.core.util.model_serializer), 158
component_data_to_dict() (in module idaes.core.util.model_serializer), 158
CONF_CREATED (idaes.dmf.workspace.Workspace attribute), 225
CONF_DATA_DIR (idaes.dmf.dmf.DMF attribute), 199
CONF_DB_FILE (idaes.dmf.dmf.DMF attribute), 199
CONF_DESC (idaes.dmf.workspace.Workspace attribute), 225
CONF_HELP_PATH (idaes.dmf.dmf.DMF attribute), 199
CONF_MODIFIED (idaes.dmf.workspace.Workspace attribute), 225
CONF_NAME (idaes.dmf.workspace.Workspace attribute), 225
CONFIG (idaes.core.holdup.Holdup1dData attribute), 168
CONFIG (idaes.core.holdup.HoldupData attribute), 169
CONFIG (idaes.core.ports.InletMixerData attribute), 172
CONFIG (idaes.core.ports.OutletSplitterData attribute), 174
CONFIG (idaes.core.ports.Port attribute), 175
CONFIG (idaes.core.process_base.ProcessBlockData attribute), 175
CONFIG (idaes.core.property_base.PropertyBlockDataBase attribute), 177
CONFIG (idaes.core.property_base.PropertyParameterBase attribute), 177
CONFIG (idaes.core.stream.StreamData attribute), 178
CONFIG (idaes.core.unit_model.UnitBlockData attribute), 180
Contact (class in idaes.dmf.resource), 208
converged() (idaes.core.stream.StreamData method), 59, 179
ConvergenceEvaluation (class in idaes.core.util.convergence.convergence_base), 147
ConvergenceEvaluationSpecification (class in idaes.core.util.convergence.convergence_base), 148
convert_csv() (in module idaes.dmf.propdata), 208
copy() (idaes.dmf.experiment.Experiment method), 203
copy() (idaes.dmf.resource.Resource method), 211
copy_var_data() (in module idaes.core.util.cut_gen), 152
count() (idaes.dmf.dmf.DMF method), 199
count_vars() (in module idaes.core.util.cut_gen), 152
Counter (class in idaes.core.util.model_serializer), 157
CPrint (class in idaes.dmf.util), 221
create_relation() (idaes.dmf.resource.Resource static method), 211
create_script() (idaes.dmf.validate.InstanceGenerator method), 223
created (idaes.dmf.resource.Resource attribute), 211

created (idaes.dmf.resource.Version attribute), 214
creator (idaes.dmf.resource.Resource attribute), 211
CSTRData (class in idaes.models.cstr), 126
CSV_MIMETYPE (idaes.dmf.resource.FilePath attribute), 209

D

data (idaes.dmf.resource.Resource attribute), 211
data (idaes.dmf.resource.ResourceTypes attribute), 212
DATA (idaes.dmf.tabular.Fields attribute), 217
data (idaes.dmf.tabular.Table attribute), 219
data files, **262**
data() (idaes.dmf.propdata.PropertyColumn method), 205
data() (idaes.dmf.propdata.StateColumn method), 208
data() (idaes.dmf.tabular.Column method), 217
DATA_ERRORS (idaes.dmf.tabular.Fields attribute), 217
DATA_ERRTYPE (idaes.dmf.tabular.Fields attribute), 217
DATA_NAME (idaes.dmf.tabular.Fields attribute), 217
DATA_UNITS (idaes.dmf.tabular.Fields attribute), 217
DATA_VALUES (idaes.dmf.tabular.Fields attribute), 217
datafile_dir (idaes.dmf.dmf.DMF attribute), 199
datafiles (idaes.dmf.resource.Resource attribute), 211
datafiles_dir (idaes.dmf.resource.Resource attribute), 211
DataFormatError, 202
datatype (idaes.dmf.tabular.Metadata attribute), 218
date (idaes.dmf.resource.Source attribute), 213
DATE (idaes.dmf.tabular.Fields attribute), 217
date (idaes.dmf.tabular.Metadata attribute), 218
DateTime (class in idaes.dmf.resource), 209
datetime_timestamp() (in module idaes.dmf.util), 222
db_file (idaes.dmf.dmf.DMF attribute), 199
deactivate() (idaes.core.stream.StreamData method), 59, 179
declare_process_block_class() (in module idaes.core.process_block), 95, 176
default_arr_len (idaes.dmf.validate.InstanceGenerator attribute), 223
default_value (idaes.dmf.resource.DateTime attribute), 209
default_value (idaes.dmf.resource.Identifier attribute), 210
default_value (idaes.dmf.resource.SemanticVersion attribute), 213
DEFAULTS (idaes.dmf.dmf.DMFConfig attribute), 201
DEFAULTS (idaes.dmf.workspace.WorkspaceConfig attribute), 226
delete() (idaes.dmf.resourcedb.ResourceDB method), 215
desc (idaes.dmf.resource.Code attribute), 208
desc (idaes.dmf.resource.FilePath attribute), 209
desc (idaes.dmf.resource.Resource attribute), 211
description (idaes.dmf.workspace.Workspace attribute), 226
dict_set() (in module idaes.core.util.misc), 96, 155
display() (idaes.core.stream.StreamData method), 60, 179
display() (idaes.core.stream.VarDict method), 60, 179
display() (idaes.models.feed.FeedData method), 100
display() (idaes.models.product.ProductData method), 103
display_flows() (idaes.core.unit_model.UnitBlockData method), 57, 181
display_infeasible_bounds() (in module idaes.core.util.debug), 152
display_infeasible_constraints() (in module idaes.core.util.debug), 152
display_P() (idaes.core.unit_model.UnitBlockData method), 57, 181
display_T() (idaes.core.unit_model.UnitBlockData method), 57, 181
display_total_flows() (idaes.core.unit_model.UnitBlockData method), 57, 181
display_variables() (idaes.core.unit_model.UnitBlockData method), 57, 181
DMF, **262**
  idaes.dmf, 185
  idaes.dmf.dmf, 185, 194
  idaes_dmf.dmf, 185
Dmf
  dmf.help, 185
DMF (class in idaes.dmf.dmf), 199
dmf (idaes.dmf.experiment.Experiment attribute), 203
dmf() (idaes.dmf.magics.DmfMagics method), 204
dmf.help
  Dmf, 185
  Help, 249
dmf_help() (idaes.dmf.magics.DmfMagics method), 204
dmf_info() (idaes.dmf.magics.DmfMagics method), 204
dmf_init() (idaes.dmf.magics.DmfMagics method), 204
dmf_list() (idaes.dmf.magics.DmfMagics method), 204
dmf_workspaces() (idaes.dmf.magics.DmfMagics method), 204
DMFBadWorkspaceError, 202
DMFConfig
  idaes.dmf.dmf, 188
DMFConfig (class in idaes.dmf.dmf), 201
DMFError, 202
DmfError, 202
DmfMagics (class in idaes.dmf.magics), 204
DMFWorkspaceNotFoundError, 202
do_copy (idaes.dmf.resource.FilePath attribute), 209
DOC_HTML_PATH (idaes.dmf.workspace.Fields attribute), 224
doi (idaes.dmf.resource.Source attribute), 214
doNothing() (in module idaes.core.util.misc), 97, 155
DTYPE (idaes.dmf.tabular.Fields attribute), 218
dump() (idaes.dmf.tabular.Table method), 219
dumps() (idaes.dmf.tabular.Table method), 219
DuplicateResourceError, 202

E

email (idaes.dmf.resource.Contact attribute), 209
embedded_units (idaes.dmf.propdata.PropertyData attribute), 206
embedded_units (idaes.dmf.tabular.TabularData attribute), 220
EquilibriumReactorData (class in idaes.models.equilibrium_reactor), 130
errors_dataframe() (idaes.dmf.propdata.PropertyData method), 206
errors_dataframe() (idaes.dmf.tabular.TabularData method), 220
Experiment
idaes.dmf.experiment, 239
Experiment (class in idaes.dmf.experiment), 203
experiment (idaes.dmf.resource.ResourceTypes attribute), 212
expr (idaes.dmf.resource.Identifier attribute), 210

F

FeedData (class in idaes.models.feed), 100
fetch_many() (idaes.dmf.dmf.DMF method), 199
fetch_one() (idaes.dmf.dmf.DMF method), 200
Fields (class in idaes.dmf.propdata), 205
Fields (class in idaes.dmf.tabular), 217
Fields (class in idaes.dmf.workspace), 224
FileError, 202
filename (idaes.dmf.dmf.DMFConfig attribute), 202
FilePath (class in idaes.dmf.resource), 209
find() (idaes.dmf.dmf.DMF method), 200
find() (idaes.dmf.resourcedb.ResourceDB method), 215
find_html_docs() (in module idaes.dmf.help), 204
find_process_byname() (in module idaes.dmf.util), 222
find_related() (idaes.dmf.dmf.DMF method), 200
find_related() (idaes.dmf.resourcedb.ResourceDB method), 215
find_workspaces() (in module idaes.dmf.workspace), 226
fix() (idaes.core.stream.VarDict method), 60, 179
fix() (idaes.models.feed.FeedData method), 101
fix_initial_conditions() (idaes.core.process_base.ProcessBlockData method), 94, 175
fix_port() (in module idaes.core.util.misc), 97, 155
FlashData (class in idaes.models.flash), 123
flowsheet, **262**
FlowsheetBlock (class in idaes.core.flowsheet_model), 53
FlowsheetBlockData
idaes.core.flowsheet_model, 51
FlowsheetBlockData (class in idaes.core.flowsheet_model), 53
FlowsheetResource
idaes.dmf.resource, 235
FlowsheetResource (class in idaes.dmf.resource), 210
from_csv() (idaes.dmf.propdata.PropertyData static method), 206
from_csv() (idaes.dmf.tabular.Metadata static method), 218
from_csv() (idaes.dmf.tabular.TabularData static method), 220
from_dict() (idaes.dmf.resource.TraitContainer class method), 214
from_flowsheet() (idaes.dmf.resource.FlowsheetResource class method), 210
from_json() (in module idaes.core.util.model_serializer), 159
fs (idaes.dmf.resource.ResourceTypes attribute), 212
fullpath (idaes.dmf.resource.FilePath attribute), 209

G

gather_global_data() (idaes.core.util.convergence.mpi_utils.ParallelTaskMa method), 150
get() (idaes.dmf.resourcedb.ResourceDB method), 215
get_class_attr_list() (idaes.core.util.model_serializer.StoreSpec method), 158
get_color_dictionary() (in module idaes.vis.plot_utils), 140
get_column() (idaes.dmf.tabular.TabularData method), 220
get_column_index() (idaes.dmf.tabular.TabularData method), 220
get_data_class_attr_list() (idaes.core.util.model_serializer.StoreSpec method), 158
get_doc_paths() (idaes.dmf.workspace.Workspace method), 226
get_fields() (idaes.dmf.workspace.WorkspaceConfig method), 226
get_file() (in module idaes.dmf.util), 222
get_html_docs() (in module idaes.dmf.help), 204
get_initialized_model() (idaes.core.util.convergence.convergence_base.Con method), 147
get_logger() (in module idaes.dmf.util), 222
get_module_author() (in module idaes.dmf.util), 222
get_module_version() (in module idaes.dmf.util), 222
get_package_units() (idaes.core.property_base.PropertyParameterBase method), 91, 177
get_port_value() (in module idaes.core.util.initialization), 153
get_property_package() (idaes.core.holdup.HoldupData method), 64, 169
get_propertydb_table() (in module idaes.dmf.dmf), 202
get_pyomo_tmp_files() (in module idaes.core.util.misc), 97, 155
get_resource_structure() (in module idaes.dmf.resource), 214
get_schema() (idaes.dmf.validate.JsonSchemaValidator method), 223
get_script() (idaes.dmf.validate.InstanceGenerator method), 223

get_solver() (idaes.core.util.convergence.convergence_base.ConvergenceEvaluation method), 148
get_specification() (idaes.core.util.convergence.convergence_base.ConvergenceEvaluation method), 148
get_stream_y_values() (in module idaes.vis.plot_utils), 140
get_sum_sq_diff() (in module idaes.core.util.cut_gen), 152
get_supported_properties() (idaes.core.property_base.PropertyParameterBase method), 91, 178
get_template() (idaes.dmf.validate.InstanceGenerator method), 223
get_time() (in module idaes.core.util.misc), 97, 155
get_variables() (idaes.dmf.validate.InstanceGenerator method), 223
GibbsReactorData (class in idaes.models.gibbs_reactor), 132
global_to_local_data() (idaes.core.util.convergence.mpi_utils.ParallelTaskManager method), 150
goodness_of_fit() (idaes.vis.plot.Plot class method), 135

H

have_mpi (idaes.core.util.convergence.mpi_utils.MPIInterface attribute), 150
heat_exchanger_network() (idaes.vis.plot.Plot class method), 135
HeatExchanger1dData (class in idaes.models.heat_exchanger_1D), 120
HeatExchangerData (class in idaes.models.heat_exchanger), 117
Help
    dmf.help, 249
    idaes.help, 249
help() (idaes.dmf.resource.Resource method), 211
HENStreamType (class in idaes.vis.plot_utils), 139
hhmmss() (in module idaes.core.util.misc), 97, 155
Holdup (class in idaes.core.holdup), 162
Holdup0D (class in idaes.core.holdup), 163
Holdup0dData (class in idaes.core.holdup), 69, 165
Holdup1D (class in idaes.core.holdup), 166
Holdup1dData (class in idaes.core.holdup), 77, 167
HoldupData (class in idaes.core.holdup), 64, 169
HoldupStatic (class in idaes.core.holdup), 169
HoldupStaticData (class in idaes.core.holdup), 83, 171

I

id_ (idaes.dmf.resource.Resource attribute), 211
ID_FIELD (idaes.dmf.resource.Resource attribute), 210
ID_FIELD (idaes.dmf.workspace.Workspace attribute), 225
Idaes
    idaes.help, 1
idaes() (idaes.dmf.magics.DmfMagics method), 204
idaes.core.util.convergence (module), 146
idaes.core.flowsheet_model
    FlowsheetBlockData, 51
idaes.core.flowsheet_model (module), 53, 162
idaes.core.holdup (module), 64, 69, 77, 83, 162
idaes.core.plugins (module), 146
idaes.core.plugins.pyosyn (module), 146
idaes.core.ports (module), 85, 172
idaes.core.process_base (module), 93, 175
idaes.core.process_block (module), 94, 176
idaes.core.property_base (module), 91, 177
idaes.core.stream (module), 59, 178
idaes.core.unit_model (module), 55, 180
idaes.core.util (module), 146
idaes.core.util.compare (module), 150
idaes.core.util.concave (module), 151
idaes.core.util.config (module), 95, 151
idaes.core.util.convergence (module), 146
idaes.core.util.convergence.convergence (module), 146
idaes.core.util.convergence.convergence_base (module), 147
idaes.core.util.convergence.mpi_utils (module), 150
idaes.core.util.convex (module), 151
idaes.core.util.cut_gen (module), 152
idaes.core.util.debug (module), 152
idaes.core.util.expr (module), 153
idaes.core.util.initialization (module), 153
idaes.core.util.mccormick (module), 153
idaes.core.util.misc (module), 96, 154
idaes.core.util.model_serializer (module), 157
idaes.core.util.mpdisagg (module), 159
idaes.core.util.stream (module), 160
idaes.core.util.var (module), 160
idaes.core.util.var_test (module), 161
idaes.dmf
    DMF, 185
idaes.dmf (module), 198
idaes.dmf.commands (module), 198
idaes.dmf.dmf
    DMF, 185, 194
    DMFConfig, 188
idaes.dmf.dmf (module), 199
idaes.dmf.errors (module), 202
idaes.dmf.experiment
    Experiment, 239
idaes.dmf.experiment (module), 203
idaes.dmf.help (module), 204
idaes.dmf.magics (module), 204
idaes.dmf.propdata (module), 205
idaes.dmf.resource
    FlowsheetResource, 235
    PropertyDataResource, 234
    Resource, 230
idaes.dmf.resource (module), 198, 208

idaes.dmf.resourcedb (module), 215
idaes.dmf.schemas (module), 198
idaes.dmf.surrmod (module), 216
idaes.dmf.tabular (module), 217
idaes.dmf.util (module), 221
idaes.dmf.validate (module), 223
idaes.dmf.workspace
    Workspace, 188
idaes.dmf.workspace (module), 224
idaes.help
    Help, 249
    Idaes, 1
idaes.models.cstr (module), 126
idaes.models.equilibrium_reactor (module), 130
idaes.models.feed (module), 100
idaes.models.flash (module), 123
idaes.models.gibbs_reactor (module), 132
idaes.models.heat_exchanger (module), 117
idaes.models.heat_exchanger_1D (module), 120
idaes.models.mixer (module), 104
idaes.models.pfr (module), 128
idaes.models.pressure_changer (module), 111
idaes.models.product (module), 102
idaes.models.separator (module), 108
idaes.models.splitter (module), 106
idaes.models.stoichiometric_reactor (module), 124
idaes.models.temperature_changer (module), 114
idaes.models.translator (module), 133
idaes.vis.plot_utils (module), 139
idaes_dmf.dmf
    DMF, 185
idaes_dmf.propdata
    PropertyData, 240
    PropertyMetadata, 240
    PropertyTable, 240
idaes_dmf.tabular
    Metadata, 240
    Table, 240
    TabularData, 240
idaes_help() (idaes.dmf.magics.DmfMagics method), 204
idaes_models.core.flowsheet_model (module), 183
idaes_models.core.process_base (module), 184
idaes_models.core.unit_model (module), 183
idaes_models.core.util.model_serializer (module), 95
idaes_models.process.conceptd.water.flowsheet (module), 183, 184
idaes_models.process.conceptd.water.main (module), 184
idaes_models.unit.water_net.feed (module), 184
idaes_models.unit.water_net.reactor (module), 184
idaes_models.unit.water_net.sink (module), 184
Identifier (class in idaes.dmf.resource), 210
idhash (idaes.dmf.resource.Code attribute), 208
import_module() (in module idaes.dmf.util), 222
indent (idaes.dmf.validate.InstanceGenerator attribute), 223
INFO (idaes.dmf.tabular.Fields attribute), 218
info (idaes.dmf.tabular.Metadata attribute), 218
info_text (idaes.dmf.resource.DateTime attribute), 209
info_text (idaes.dmf.resource.Identifier attribute), 210
info_text (idaes.dmf.resource.RelationType attribute), 210
info_text (idaes.dmf.resource.SemanticVersion attribute), 213
init_conf() (in module idaes.dmf.commands), 198
init_isentropic() (idaes.models.pressure_changer.PressureChangerData method), 111
initialize() (idaes.core.holdup.Holdup0dData method), 70, 165
initialize() (idaes.core.holdup.Holdup1dData method), 78, 168
initialize() (idaes.core.holdup.HoldupStaticData method), 84, 171
initialize() (idaes.core.ports.InletMixerData method), 87, 172
initialize() (idaes.core.ports.OutletSplitterData method), 89, 174
initialize() (idaes.core.property_base.PropertyBlockBase method), 93
initialize() (idaes.core.unit_model.UnitBlockData method), 57, 181
initialize() (idaes.models.heat_exchanger.HeatExchangerData method), 117
initialize() (idaes.models.heat_exchanger_1D.HeatExchanger1dData method), 120
initialize() (idaes.models.pressure_changer.PressureChangerData method), 112
initialize() (idaes.models.translator.TranslatorData method), 133
InletMixer (class in idaes.core.ports), 172
InletMixerData (class in idaes.core.ports), 87, 172
inputs (idaes.core.util.convergence.convergence_base.ConvergenceEvaluati attribute), 148
InstanceGenerator (class in idaes.dmf.validate), 223
instances() (idaes.dmf.validate.JsonSchemaValidator method), 224
InvalidRelationError, 202
is_fixed_by_bounds() (in module idaes.core.util.var), 161
is_hot_or_cold_utility() (in module idaes.vis.plot_utils), 141
is_jupyter_notebook() (in module idaes.dmf.util), 222
is_linear() (in module idaes.core.util.expr), 153
is_parameter_block() (in module idaes.core.util.config), 95, 151
is_port() (in module idaes.core.util.config), 96, 151
is_process_unit() (idaes.core.unit_model.UnitBlockData method), 57, 182
is_property_column() (idaes.dmf.propdata.PropertyData

method), 206
is_python() (in module idaes.dmf.util), 222
is_resource_json() (in module idaes.dmf.util), 222
is_root() (idaes.core.util.convergence.mpi_utils.ParallelTaskManager method), 150
is_state_column() (idaes.dmf.propdata.PropertyData method), 206
is_tmp (idaes.dmf.resource.FilePath attribute), 209
isbn (idaes.dmf.resource.Source attribute), 214
isfixed() (idaes.core.util.model_serializer.StoreSpec class method), 158
isobar() (idaes.vis.plot.Plot class method), 136
isoformat() (idaes.dmf.resource.DateTime class method), 209

J

JsonSchemaValidator (class in idaes.dmf.validate), 223
jupyter (idaes.dmf.resource.ResourceTypes attribute), 213
jupyter_nb (idaes.dmf.resource.ResourceTypes attribute), 213

K

keywords (idaes.dmf.validate.InstanceGenerator attribute), 223

L

language (idaes.dmf.resource.Code attribute), 208
language (idaes.dmf.resource.Source attribute), 214
lb() (in module idaes.core.util.var), 161
line_expr (idaes.dmf.tabular.Metadata attribute), 218
link() (idaes.dmf.experiment.Experiment method), 203
list_of_floats() (in module idaes.core.util.config), 96, 151
list_of_strings() (in module idaes.core.util.config), 96, 151
list_resources() (in module idaes.dmf.commands), 198
list_workspaces() (in module idaes.dmf.commands), 198
load() (idaes.dmf.propdata.PropertyTable static method), 207
load() (idaes.dmf.tabular.Table class method), 219
load_json() (in module idaes.core.util.model_serializer), 159
location (idaes.dmf.resource.Code attribute), 208
log_active_nonlinear_constraints() (in module idaes.core.util.expr), 153
LOG_CONF (idaes.dmf.workspace.Fields attribute), 224
log_disjunct_values() (in module idaes.core.util.debug), 153

M

magics (idaes.dmf.magics.DmfMagics attribute), 205
main() (in module idaes.core.util.convergence.convergence), 147
make_stream_table() (in module idaes.core.util.stream), 160
max_ub() (in module idaes.core.util.var), 161
Max_ubb() (in module idaes.core.util.var), 161
META (idaes.dmf.tabular.Fields attribute), 218
meta (idaes.dmf.workspace.Workspace attribute), 226
Metadata
    idaes_dmf.tabular, 240
Metadata (class in idaes.dmf.tabular), 218
metadata (idaes.dmf.resource.FilePath attribute), 209
metadata (idaes.dmf.tabular.Table attribute), 219
mimetype (idaes.dmf.resource.FilePath attribute), 209
min_lb() (in module idaes.core.util.var), 161
min_lbb() (in module idaes.core.util.var), 161
MixerData (class in idaes.models.mixer), 104
model_check() (idaes.core.flowsheet_model.FlowsheetBlockData method), 53
model_check() (idaes.core.holdup.Holdup0dData method), 70, 165
model_check() (idaes.core.holdup.Holdup1dData method), 78, 168
model_check() (idaes.core.holdup.HoldupStaticData method), 84, 171
model_check() (idaes.core.ports.InletMixerData method), 87, 173
model_check() (idaes.core.ports.OutletSplitterData method), 90, 174
model_check() (idaes.core.unit_model.UnitBlockData method), 58, 182
model_check() (idaes.models.heat_exchanger.HeatExchangerData method), 117
model_check() (idaes.models.heat_exchanger_1D.HeatExchanger1dData method), 121
model_check() (idaes.models.pressure_changer.PressureChangerData method), 112
model_check() (idaes.models.temperature_changer.TemperatureChangerDa method), 114
model_check() (idaes.models.translator.TranslatorData method), 134
modified (idaes.dmf.resource.Resource attribute), 211
ModuleFormatError, 202
MPIInterface (class in idaes.core.util.convergence.mpi_utils), 150

N

name (idaes.dmf.resource.Code attribute), 208
name (idaes.dmf.resource.Contact attribute), 209
name (idaes.dmf.resource.Resource attribute), 212
name (idaes.dmf.resource.Version attribute), 214
name (idaes.dmf.workspace.Workspace attribute), 226
names() (idaes.dmf.propdata.PropertyData method), 207
names() (idaes.dmf.tabular.TabularData method), 221
nb (idaes.dmf.resource.ResourceTypes attribute), 213

NEED_INIT_CMD (idaes.dmf.magics.DmfMagics attribute), 204
none_if_empty() (in module idaes.core.util.var), 161
NoSuchResourceError, 202
num_columns (idaes.dmf.tabular.TabularData attribute), 221
num_rows (idaes.dmf.tabular.TabularData attribute), 221

O

object (idaes.dmf.resource.Triple attribute), 214
open() (idaes.dmf.resource.FilePath method), 209
OutletSplitter (class in idaes.core.ports), 173
OutletSplitterData (class in idaes.core.ports), 89, 174

P

ParallelTaskManager (class in idaes.core.util.convergence.mpi_utils), 150
PARAM_DATA_KEY (idaes.dmf.surrmod.SurrogateModel attribute), 216
ParseError, 202
path (idaes.dmf.resource.FilePath attribute), 209
Plot (class in idaes.vis.plot), 135
plot_line_segment() (in module idaes.vis.plot_utils), 141
plot_stream_arrow() (in module idaes.vis.plot_utils), 142
Port (class in idaes.core.ports), 85, 175
post_transform_build() (idaes.core.flowsheet_model.FlowsheetBlockData method), 54
post_transform_build() (idaes.models.cstr.CSTRData method), 126
post_transform_build() (idaes.models.equilibrium_reactor.EquilibriumReactorData method), 130
post_transform_build() (idaes.models.feed.FeedData method), 101
post_transform_build() (idaes.models.flash.FlashData method), 123
post_transform_build() (idaes.models.gibbs_reactor.GibbsReactorData method), 132
post_transform_build() (idaes.models.heat_exchanger.HeatExchangerData method), 117
post_transform_build() (idaes.models.heat_exchanger_1D.HeatExchanger1dData method), 121
post_transform_build() (idaes.models.mixer.MixerData method), 104
post_transform_build() (idaes.models.pressure_changer.PressureChangerData method), 112
post_transform_build() (idaes.models.product.ProductData method), 103
post_transform_build() (idaes.models.separator.SeparatorData method), 108
post_transform_build() (idaes.models.splitter.SplitterData method), 106
post_transform_build() (idaes.models.stoichiometric_reactor.StoichiometricReactorData method), 124
post_transform_build() (idaes.models.temperature_changer.TemperatureCha method), 114
post_transform_build() (idaes.models.translator.TranslatorData method), 134
predicate (idaes.dmf.resource.Triple attribute), 214
Predicates (idaes.dmf.resource.RelationType attribute), 210
PressureChangerData (class in idaes.models.pressure_changer), 111
pretty() (idaes.dmf.resource.SemanticVersion class method), 213
print_active_units() (idaes.core.flowsheet_model.FlowsheetBlockData method), 54
print_all_units() (idaes.core.flowsheet_model.FlowsheetBlockData method), 54
print_convergence_statistics() (in module idaes.core.util.convergence.convergence_base), 148
println() (idaes.dmf.util.CPrint method), 221
ProcessBase (class in idaes_models.core.process_base), 184
ProcessBlock (class in idaes.core.process_block), 94, 176
ProcessBlockData (class in idaes.core.process_base), 93, 175
ProductData (class in idaes.models.product), 102
profile() (idaes.vis.plot.Plot class method), 136
properties (idaes.dmf.propdata.PropertyData attribute), 207
property data, **262**
property model, **263**
property_data (idaes.dmf.resource.ResourceTypes attribute), 213
property_model() (idaes.vis.plot.Plot class method), 137
property_table (idaes.dmf.resource.Resource attribute), 212
PropertyBlockBase (class in idaes.core.property_base), 93
PropertyBlockDataBase (class in idaes.core.property_base), 92, 177
PropertyColumn (class in idaes.dmf.propdata), 205
PropertyData
    idaes_dmf.propdata, 240
PropertyData (class in idaes.dmf.propdata), 205
PropertyDataResource
    idaes.dmf.resource, 234
PropertyDataResource (class in idaes.dmf.resource), 210
PropertyMetadata
    idaes_dmf.propdata, 240
PropertyMetadata (class in idaes.dmf.propdata), 207
PropertyParameterBase (class in idaes.core.property_base), 91, 177
PropertyTable
    idaes_dmf.propdata, 240
PropertyTable (class in idaes.dmf.propdata), 207

put() (idaes.dmf.resourcedb.ResourceDB method), 215
Pyomo, **263**
Pyosyn (class in idaes.core.plugins.pyosyn), 146
python (idaes.dmf.resource.ResourceTypes attribute), 213

R

R_DERIVED (in module idaes.dmf.resource), 210
rank (idaes.core.util.convergence.mpi_utils.MPIInterface attribute), 150
read() (idaes.dmf.resource.FilePath method), 210
register() (in module idaes.dmf.magics), 205
registered (idaes.dmf.magics.DmfMagics attribute), 205
relation, **263**
relations (idaes.dmf.resource.Resource attribute), 212
RelationType (class in idaes.dmf.resource), 210
release (idaes.dmf.resource.Code attribute), 208
release_state() (idaes.core.holdup.Holdup0dData method), 70, 165
release_state() (idaes.core.holdup.Holdup1dData method), 78, 168
release_state() (idaes.core.holdup.HoldupStaticData method), 84, 171
release_state() (idaes.core.ports.InletMixerData method), 88, 173
release_state() (idaes.core.ports.OutletSplitterData method), 90, 175
remove() (idaes.dmf.dmf.DMF method), 201
remove() (idaes.dmf.experiment.Experiment method), 204
requires_solver() (in module idaes.core.util.misc), 97, 155
reset() (idaes.dmf.validate.JsonSchemaValidator method), 224
residual() (idaes.vis.plot.Plot class method), 137
resize() (idaes.vis.plot.Plot method), 138
Resource
    idaes.dmf.resource, 230
resource, **263**
Resource (class in idaes.dmf.resource), 210
ResourceDB (class in idaes.dmf.resourcedb), 215
ResourceError, 202
ResourceTypes (class in idaes.dmf.resource), 212
revision (idaes.dmf.resource.Version attribute), 214
root (idaes.dmf.resource.FilePath attribute), 210
root (idaes.dmf.workspace.Workspace attribute), 226
root_var (idaes.dmf.validate.InstanceGenerator attribute), 223
round_() (in module idaes.core.util.misc), 97, 155
ROWS (idaes.dmf.tabular.Fields attribute), 218
run() (idaes.dmf.surrmod.SurrogateModel method), 216
run_convergence_evaluation() (in module idaes.core.util.convergence.convergence_base), 149
run_convergence_evaluation_from_sample_file() (in module idaes.core.util.convergence.convergence_base), 149

S

save() (idaes.dmf.dmf.DMFConfig method), 202
save() (idaes.vis.plot.Plot method), 138
save_convergence_statistics() (in module idaes.core.util.convergence.convergence_base), 149
save_json() (in module idaes.core.util.model_serializer), 159
save_results_to_dmf() (in module idaes.core.util.convergence.convergence_base), 149
SearchError, 202
self_proj_var_rule() (in module idaes.core.util.cut_gen), 152
SemanticVersion (class in idaes.dmf.resource), 213
sensitivity() (idaes.vis.plot.Plot class method), 138
SeparatorData (class in idaes.models.separator), 108
set_input_data() (idaes.dmf.surrmod.SurrogateModel method), 216
set_input_data_np() (idaes.dmf.surrmod.SurrogateModel method), 216
set_meta() (idaes.dmf.workspace.Workspace method), 226
set_read_callback() (idaes.core.util.model_serializer.StoreSpec method), 158
set_validation_data() (idaes.dmf.surrmod.SurrogateModel method), 217
set_validation_data_np() (idaes.dmf.surrmod.SurrogateModel method), 217
set_write_callback() (idaes.core.util.model_serializer.StoreSpec method), 158
setup_mccormick_cuts() (in module idaes.core.util.mccormick), 154
setup_multiparametric_disagg() (in module idaes.core.util.mpdisagg), 160
show() (idaes.vis.plot.Plot method), 138
size (idaes.core.util.convergence.mpi_utils.MPIInterface attribute), 150
SliceVar (class in idaes.core.util.var), 160
smooth_abs() (in module idaes.core.util.misc), 97, 156
smooth_max() (in module idaes.core.util.misc), 98, 156
smooth_min() (in module idaes.core.util.misc), 98, 156
smooth_minmax() (in module idaes.core.util.misc), 98, 156
solve_indexed_blocks() (in module idaes.core.util.misc), 98, 156
Source (class in idaes.dmf.resource), 213
source (idaes.dmf.resource.Source attribute), 214
source (idaes.dmf.tabular.Metadata attribute), 218
source_expr (idaes.dmf.tabular.Metadata attribute), 218
sources (idaes.dmf.resource.Resource attribute), 212
SplitterData (class in idaes.models.splitter), 106

squish() (in module idaes.core.util.mccormick), 154
squish_concat() (in module idaes.core.util.mccormick), 154
StateColumn (class in idaes.dmf.propdata), 208
states (idaes.dmf.propdata.PropertyData attribute), 207
Stats (class in idaes.core.util.convergence.convergence_base), 148
StoichiometricReactorData (class in idaes.models.stoichiometric_reactor), 124
StoreSpec (class in idaes.core.util.model_serializer), 157
Stream (class in idaes.core.stream), 178
stream_table() (idaes.vis.plot.Plot class method), 138
StreamData (class in idaes.core.stream), 59, 178
strlist() (in module idaes.dmf.util), 222
subdir (idaes.dmf.resource.FilePath attribute), 210
subject (idaes.dmf.resource.Triple attribute), 214
suffix() (idaes.core.util.model_serializer.StoreSpec class method), 158
surrmod (idaes.dmf.resource.ResourceTypes attribute), 213
SurrogateModel (class in idaes.dmf.surrmod), 216

T

Table
    idaes_dmf.tabular, 240
Table (class in idaes.dmf.tabular), 218
table (idaes.dmf.resource.Resource attribute), 212
tabular_data (idaes.dmf.resource.ResourceTypes attribute), 213
TabularData
    idaes_dmf.tabular, 240
TabularData (class in idaes.dmf.tabular), 219
TabularDataResource (class in idaes.dmf.resource), 214
TabularObject (class in idaes.dmf.tabular), 221
tags (idaes.dmf.resource.Resource attribute), 212
TempDir (class in idaes.dmf.util), 222
TemperatureChangerData (class in idaes.models.temperature_changer), 114
terminate_pid() (in module idaes.dmf.util), 222
tighten_block_bound() (in module idaes.core.util.var), 161
tighten_mc_var() (in module idaes.core.util.var), 161
tighten_var_bound() (in module idaes.core.util.var), 161
TITLE (idaes.dmf.tabular.Fields attribute), 218
title (idaes.dmf.tabular.Metadata attribute), 218
to_json() (in module idaes.core.util.model_serializer), 159
tradeoff() (idaes.vis.plot.Plot class method), 138
TraitContainer (class in idaes.dmf.resource), 214
TranslatorData (class in idaes.models.translator), 133
Triple (class in idaes.dmf.resource), 214
try_eval() (in module idaes.core.util.concave), 151
try_eval() (in module idaes.core.util.convex), 152
turn_off_grid_and_axes_ticks() (in module idaes.vis.plot_utils), 142
type (idaes.dmf.resource.Code attribute), 208
type (idaes.dmf.resource.Resource attribute), 212
TYPE_FIELD (idaes.dmf.resource.Resource attribute), 210
type_name (idaes.dmf.propdata.PropertyColumn attribute), 205
type_name (idaes.dmf.propdata.StateColumn attribute), 208
type_name (idaes.dmf.tabular.Column attribute), 217

U

ub() (in module idaes.core.util.var), 161
unfix() (idaes.core.stream.VarDict method), 60, 180
unfix() (idaes.models.feed.FeedData method), 101
unfix_initial_conditions() (idaes.core.process_base.ProcessBlockData method), 94, 175
unfix_port() (in module idaes.core.util.misc), 98, 156
unit model, **263**
UnitBlock (class in idaes.core.unit_model), 55, 182
UnitBlockData (class in idaes.core.unit_model), 56, 180
unwrap() (in module idaes.core.util.var), 161
update() (idaes.dmf.dmf.DMF method), 201
update() (idaes.dmf.experiment.Experiment method), 204
update() (idaes.dmf.resourcedb.ResourceDB method), 215
uuid (idaes.dmf.resource.Resource attribute), 212

V

validate() (idaes.dmf.resource.DateTime method), 209
validate() (idaes.dmf.resource.Identifier method), 210
validate() (idaes.dmf.resource.RelationType method), 210
validate() (idaes.dmf.resource.SemanticVersion method), 213
validate() (idaes.dmf.validate.JsonSchemaValidator method), 224
validate() (in module idaes.vis.plot_utils), 142
validate_elements() (idaes.dmf.resource.ValidatingList method), 214
ValidatingList (class in idaes.dmf.resource), 214
VALS (idaes.dmf.tabular.Fields attribute), 218
value() (idaes.core.util.model_serializer.StoreSpec class method), 158
value_correct() (in module idaes.core.util.var_test), 161
value_isfixed() (idaes.core.util.model_serializer.StoreSpec class method), 158
value_isfixed_isactive() (idaes.core.util.model_serializer.StoreSpec class method), 158
values_dataframe() (idaes.dmf.propdata.PropertyData method), 207
values_dataframe() (idaes.dmf.tabular.TabularData method), 221

VarDict (class in idaes.core.stream), 60, 179
Version (class in idaes.dmf.resource), 214
version (idaes.dmf.resource.Resource attribute), 212

W

Workspace
    idaes.dmf.workspace, 188
workspace, **263**
Workspace (class in idaes.dmf.workspace), 224
WORKSPACE (idaes.dmf.dmf.DMFConfig attribute), 201
workspace (idaes.dmf.dmf.DMFConfig attribute), 202
WORKSPACE_CONFIG
    (idaes.dmf.workspace.Workspace attribute), 225
workspace_import() (in module idaes.dmf.commands), 199
workspace_info() (in module idaes.dmf.commands), 199
workspace_init() (in module idaes.dmf.commands), 199
WorkspaceConfig (class in idaes.dmf.workspace), 226
WorkspaceConfMissingField, 203
WorkspaceConfNotFoundError, 203
WorkspaceError, 203
WorkspaceNotFoundError, 203
wrap_var() (in module idaes.core.util.var), 161
Wrapper (class in idaes.core.util.var), 160
write() (idaes.dmf.util.CPrint method), 221
write_sample_file() (in module idaes.core.util.convergence.convergence_base), 149
wsid (idaes.dmf.workspace.Workspace attribute), 226

X

xp (idaes.dmf.resource.ResourceTypes attribute), 213