Module 1: Flash Unit

In this module, we will familiarize ourselves with the IDAES framework by creating and working with a flowsheet that contains a single flash tank. The flash tank will be used to perform separation of Benzene and Toluene. The inlet specifications for this flash tank are:

Inlet Specifications:

- Mole fraction (Benzene) = 0.5
- Mole fraction (Toluene) = 0.5
- Pressure = 101325 Pa
- Temperature = 368 K

We will complete the following tasks:

- Create the model and the IDAES Flowsheet object
- Import the appropriate property packages
- Create the flash unit and set the operating conditions
- Initialize the model and simulate the system
- Demonstrate analyses on this model through some examples and exercises

Create the Model and the IDAES Flowsheet

In the next cell, we will perform the necessary imports to get us started. From `pyomo.environ` (a standard import for the Pyomo package), we are importing `ConcreteModel` (to create the Pyomo model that will contain the IDAES flowsheet) and `SolverFactory` (to create the object we will use to solve the equations). We will also import `Constraint` as we will be adding a constraint to the model later in the module. Lastly, we also import `value` from Pyomo. This is a function that can be used to return the current numerical value for variables and parameters in the model. These are all part of Pyomo.

We will also import the main `FlowsheetBlock` from IDAES. The flowsheet block will contain our unit model.

Inline Exercise: Execute the cell below to perform the imports. Let a workshop organizer know if you see any errors.

In [1]:
```
from pyomo.environ import ConcreteModel, SolverFactory, Constraint, valu
from idaes.core import FlowsheetBlock
```

In the next cell, we will create the `ConcreteModel` and the `FlowsheetBlock` , and attach the flowsheet block to the Pyomo model.

Inline Exercise: Execute the cell below to create the objects

In [2]:
```
m = ConcreteModel()
m.fs = FlowsheetBlock(default={"dynamic": False})
```

At this point, we have a single Pyomo model that contains an (almost) empty flowsheet block.

Inline Exercise: Use the pprint method on the model object to see what is currently contained in the model.

In [3]:
```
# Todo: call pprint on the model
m.pprint()
```

```
1 Block Declarations
    1 Set Declarations
        time : Dim=0, Dimen=1, Size=1, Domain=None, Ordered=Insertion,
Bounds=(0.0, 0.0)
            [0.0]

    1 Declarations: time

1 Declarations: fs
```

Define Properties

We need to define the property package for our flowsheet. In this example, we have created a property package based on ideal VLE that contains the necessary components.

IDAES supports creation of your own property packages that allow for specification of the fluid using any set of valid state variables (e.g., component molar flows vs overall flow and mole fractions). This flexibility is designed to support advanced modeling needs that may rely on specific formulations. As well, the IDAES team has completed some general property packages (and is currently working on more). For this workshop, we will import the BTX_idea_VLE property package and create a properties block for the flowsheet.

This properties block will be passed to our unit model to define the appropriate state variables and equations for performing thermodynamic calculations.

Inline Exercise: Execute the following two cells to import and create the properties block.

In [4]:
```
import idaes.property_models.ideal.BTX_ideal_VLE as ideal_props
```

In [5]:
```
m.fs.properties = ideal_props.BTXParameterBlock()
```

Adding Flash Unit

Now that we have the flowsheet and the properties defined, we can create the flash unit and add it to the flowsheet. IDAES fully supports the development of customized unit models (which we will see in a later module). However, there are a number of unit models already pre-written for you in the IDAES framework.

Some of the IDAES pre-written unit models:

- Mixer / Splitter
- Heater / Cooler
- Heat Exchangers (simple and 1D discretized)
- Flash
- Reactors (kinetic, equilibrium, gibbs, stoichiometric conversion)
- Pressure changing equipment (compressors, expanders, pumps)
- Feed and Product (source / sink) components

In this module, we will import the `Flash` unit model from `idaes.unit_models` and create an instance of the flash unit, attaching it to the flowsheet. Each IDAES unit model has several configurable options to customize the model behavior, but also includes defaults for these options. In this example, we will specify that the property package to be used with the Flash is the one we created earlier.

Inline Exercise: Execute the following two cells to import the Flash and create an instance of the unit model, attaching it to the flowsheet object.

In [6]:
```
from idaes.unit_models import Flash
```

In [7]:
```
m.fs.flash = Flash(default={"property_package": m.fs.properties})
```

At this point, we have created a flowsheet and a properties block. We have also created a flash unit and added it to the flowsheet. Under the hood, IDAES has created the required state variables and model equations. Everything is open. You can see these variables and equations by calling the Pyomo method `pprint` on the model, flowsheet, or flash tank objects. Note that this output is very exhaustive, and is not intended to provide any summary information about the model, but rather a complete picture of all of the variables and equations in the model.

Set Operating Conditions

Now that we have created our unit model, we can specify the necessary operating conditions. It is often very useful to determine the degrees of freedom before we specify any conditions.

The `idaes.ui.report` package has a function `degrees_of_freedom` . To see how to use this function, we can make use of the Python function `help(func)` . This function prints the appropriate documentation string for the function.

Inline Exercise: Import the degrees_of_freedom function and print the help for the function by calling the Python help function.

In [8]:
```
# Todo: import the degrees_of_freedom function from the idaes.ui.report
from idaes.ui.report import degrees_of_freedom

# Todo: Call the python help on the degrees_of_freedom function
help(degrees_of_freedom)
```

```
Help on function degrees_of_freedom in module idaes.ui.report:

degrees_of_freedom(blk)
    Return the degrees of freedom.
```

Inline Exercise: Now print the degrees of freedom for your model. The result should be 7.

In [9]:
```
# Todo: print the degrees of freedom for your model
print("Degrees of Freedom =", degrees_of_freedom(m))
```

```
Degrees of Freedom = 7
```

To satisfy our degrees of freedom, we will first specify the inlet conditions. We can specify these values through the `inlet` port of the flash unit.

As an example, to fix the molar flow of the inlet to be 1.0, you can use the following notation:

```
m.fs.flash.inlet.flow_mol.fix(1.0)
```

To specify variables that are indexed by components, you can use the following notation:

```
m.fs.flash.inlet.mole_frac[0, "benzene"].fix(0.5)
```

Note: The "0" in the indexing of the component mole fraction is present because IDAES models support both dynamic and steady state simulation, and the "0" refers to a timestep. Dynamic modeling is beyond the scope of this workshop. Since we are performing steady state modeling, there is only a single timestep in the model.

For this module, we will use the following specifications for the inlet:

- inlet overall molar flow = 1.0 (`flow_mol`)
- inlet temperature = 368 K (`temperature`)
- inlet pressure = 101325 Pa (`pressure`)
- inlet mole fraction (benzene) = 0.5 (`mole_frac[0, "benzene"]`)
- inlet mole fraction (toluene) = 0.5 (`mole_frac[0, "toluene"]`)

Inline Exercise: Write the code below to specify the inlet conditions described above

In [10]:
```
# Todo: add code for the 5 specifications given above
m.fs.flash.inlet.flow_mol.fix(1)
m.fs.flash.inlet.temperature.fix(368)
m.fs.flash.inlet.pressure.fix(101325)
m.fs.flash.inlet.mole_frac[0, "benzene"].fix(0.5)
m.fs.flash.inlet.mole_frac[0, "toluene"].fix(0.5)
```

If everything is working correctly, the degrees of freedom should now be 2.

Inline Exercise: Check the remaining degrees of freedom for the model.

In [11]:
```
# Todo: print the degrees of freedom for your model
print("Degrees of Freedom =", degrees_of_freedom(m))
```

```
Degrees of Freedom = 2
```

To satisfy the remaining degrees of freedom, we will make two additional specifications on the flash tank itself. To specify the value of a variable on the unit itself, use the following notation.

```
m.fs.flash.heat_duty.fix(0)
```

We will add the following specifications to the flash tank:

- The heat duty on the flash set to zero (`heat_duty`)
- The pressure drop across the flash tank set to 0 (`deltaP`)

Inline Exercise: Write the code below to fix variables to the specifications described above.

In [12]:
```
# Todo: add code for the 2 specifications given above
m.fs.flash.heat_duty.fix(0)
m.fs.flash.deltaP.fix(0)
```

Inline Exercise: Check the degrees of freedom again to ensure that the system is now square. You should see that the degrees of freedom is now 0.

In [13]:
```
# Todo: print the degrees of freedom for your model
print("Degrees of Freedom =", degrees_of_freedom(m))
```

```
Degrees of Freedom = 0
```

Initializing the Model

IDAES includes pre-written initialization routines for all unit models. You can call this initialize method on the units. In the next module, we will demonstrate the use of a sequential modular solve cycle to initialize flowsheets.

Inline Exercise: Call the initialize method on the flash unit to initialize the model.

In [14]:
```
# Todo: initialize the flash unit
m.fs.flash.initialize()
```

```
2019-05-13 14:38:53 - INFO - idaes.property_models.ideal.ideal_prop_pa
ck_VLE - Starting fs.flash.control_volume.properties_in initialisation
2019-05-13 14:38:53 - INFO - idaes.property_models.ideal.ideal_prop_pa
ck_VLE - Starting fs.flash.control_volume.properties_out initialisatio
n
```

Now that the model has been defined and intialized, we can solve the model.

Inline Exercise: Using the notation described in the previous model, create an instance of the "ipopt" solver and use it to solve the model. Set the tee option to True to see the log output.

In [15]:
```
# Todo: create the ipopt solver
solver = SolverFactory('ipopt')

# Todo: solve the model
status = solver.solve(m, tee=True)
```

```
Ipopt 3.12.12:

**********************************************************************
********
This program contains Ipopt, a library for large-scale nonlinear optim
ization.
 Ipopt is released as open source code under the Eclipse Public Licens
e (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
(http://projects.coin-or.org/Ipopt)
**********************************************************************
********

This is Ipopt version 3.12.12, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt document
ation).

Number of nonzeros in equality constraint Jacobian...:      139
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:       53

Total number of variables............................:       43
                     variables with only lower bounds:        3
                variables with lower and upper bounds:       10
                     variables with only upper bounds:        0
Total number of equality constraints.................:       43
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du al
pha_pr  ls
   0  0.0000000e+00 4.37e-11 1.00e+00  -1.0 0.00e+00    -  0.00e+00 0.
00e+00   0

Number of Iterations....: 0
```

```
                                   (scaled)                 (unscaled)
Objective...............:   0.0000000000000000e+00    0.00000000000000
00e+00
Dual infeasibility......:   0.0000000000000000e+00    0.00000000000000
00e+00
Constraint violation....:   3.5261611248706071e-12    4.36557456851005
55e-11
Complementarity.........:   0.0000000000000000e+00    0.00000000000000
00e+00
Overall NLP error.......:   3.5261611248706071e-12    4.36557456851005
55e-11


Number of objective function evaluations             = 1
Number of objective gradient evaluations             = 1
Number of equality constraint evaluations            = 1
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 1
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 0
Total CPU secs in IPOPT (w/o function evaluations)   =      0.020
Total CPU secs in NLP function evaluations           =      0.000

EXIT: Optimal Solution Found.
```

Viewing the Results

Once a model is solved, the values returned by the solver are loaded into the model object itself. We can access the value of any variable in the model with the `value` function. For example:

```
print('Vap. Outlet Temperature = ', print(value(m.fs.flash.vap_o
utlet.temperature[0])))
```

You can also find more information about a variable or an entire port using the `display` method from Pyomo:

```
m.fs.flash.vap_outlet.temperature.display()
m.fs.flash.vap_outlet.display()
```

Inline Exercise: Use the value function with print to show the current value of the flash vapor outlet pressure. Also use the display function to see the values of the variables in the vap_outlet and the liq_outlet

```
In [16]: # Todo: print the pressure of the flash vapor outlet
         print('Pressure =', value(m.fs.flash.vap_outlet.pressure[0]))

         print()
         print('Output from display:')
         # Todo: Call display on vap_outlet and liq_outlet of the flash
         m.fs.flash.vap_outlet.display()
         m.fs.flash.liq_outlet.display()
```

```
Pressure = 101325.0

Output from display:
vap_outlet : Size=1
    Key  : Name        : Value
    None :    flow_mol : {0.0: 0.3546244301390874}
         :   mole_frac : {(0.0, 'benzene'): 0.6429364285519159, (0.0,
'toluene'): 0.35706357144808415}
         :    pressure : {0.0: 101325.0}
         : temperature : {0.0: 368.0}
liq_outlet : Size=1
    Key  : Name        : Value
    None :    flow_mol : {0.0: 0.6453755698609125}
         :   mole_frac : {(0.0, 'benzene'): 0.42145852448015086, (0.0,
'toluene'): 0.5785414755198491}
         :    pressure : {0.0: 101325.0}
         : temperature : {0.0: 368.0}
```

The output from `display` is quite exhaustive and not really intended to provide quick summary information. Because Pyomo is built on Python, there are opportunities to format the output any way we like. The IDAES team is working on several potential solutions for effective viewing of model results.

To help with this workshop, we created a short module called `workshoptools` that contains a few helpful functions. Here, we will import the `print_ports_summary` function from the `workshoptools` module and use it to view our results.

Inline Exercise: Execute the cell below which uses the function above to print a summary of the key variables in the inlet, the vapor, and the liquid ports of the flash.

```
In [17]: from workshoptools import print_ports_summary
         print_ports_summary([m.fs.flash.inlet, m.fs.flash.vap_outlet, m.fs.flash
```

```
                 Variable      fs.flash.inlet   fs.flash.vap_outlet
fs.flash.liq_outlet
---------------------------    --------------   -------------------
-------------------
             flow_mol[0.0]           1.000000              0.354624
0.645376
mole_frac[(0.0, 'benzene')]          0.500000              0.642936
0.421459
mole_frac[(0.0, 'toluene')]          0.500000              0.357064
0.578541
          temperature[0.0]         368.000000            368.000000
368.000000
             pressure[0.0]      101325.000000         101325.000000
101325.000000
```

Studying Purity as a Function of Heat Duty

Since the entire modeling framework is built upon Python, it includes a complete programming environment for whatever analysis we may want to perform. In this next exercise, we will make use of what we learned in this and the previous module to generate a figure showing some output variables as a function of the heat duty in the flash tank.

First, let's import the matplotlib package for plotting as we did in the previous module.

Inline Exercise: Execute the cell below to import matplotlib appropriately.

```
In [18]: import matplotlib.pyplot as plt
```

Exercise specifications:

- Generate a figure showing the flash tank heat duty (`m.fs.flash.heat_duty[0]`) vs. the vapor flowrate (`m.fs.flash.vap_outlet.mol_flow[0]`)
- Specify the heat duty from -17000 to 25000 over 20 steps

Inline Exercise: Using what you have learned so far, fill in the missing code below to generate the figure specified above. (Hint: import numpy and use the linspace function from the previous module)

```
In [19]: # import the solve_successful checking function from workshop tools
         from workshoptools import solve_successful

         # Todo: import numpy
```

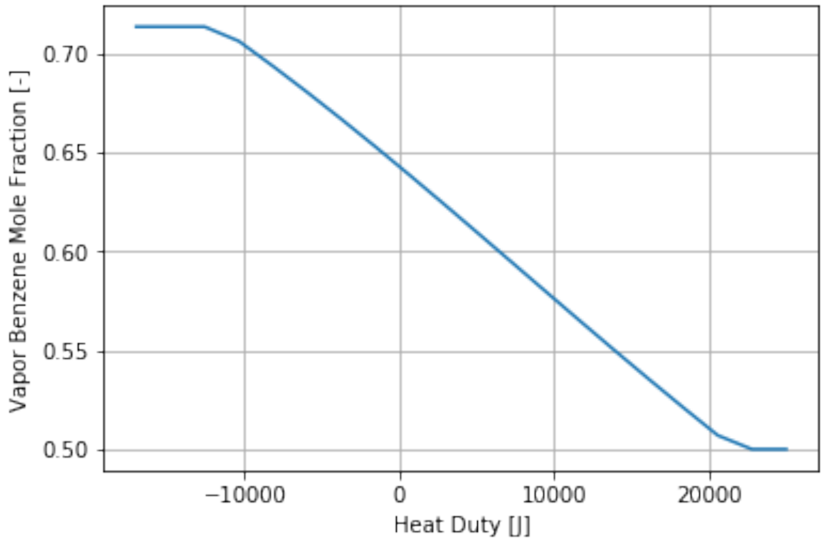
```python
import numpy as np

# create the empty lists to store the results that will be plotted
Q = []
V = []

# create the solver
solver = SolverFactory('ipopt')

# Todo: Write the for loop specification using numpy's linspace
for duty in np.linspace(-17000, 25000, 20):
    # fix the heat duty
    m.fs.flash.heat_duty.fix(duty)

    # append the value of the duty to the Q list
    Q.append(duty)

    # print the current simulation
    print("Simulating with Q = ", value(m.fs.flash.heat_duty[0]))

    # Todo: solve the model
    status = solver.solve(m)

    # append the value for vapor fraction if the solve was successful
    if solve_successful(status):
        V.append(value(m.fs.flash.vap_outlet.flow_mol[0]))
        print('... solve successful.')
    else:
        V.append(0.0)
        print('... solve failed.')

# Todo: Create and show the figure
plt.figure("Vapor Fraction")
plt.plot(Q, V)
plt.grid()
plt.xlabel("Heat Duty [J]")
plt.ylabel("Vapor Fraction [-]")
plt.show()
```

```
Simulating with Q =  -17000.0
... solve successful.
Simulating with Q =  -14789.473684210527
... solve successful.
Simulating with Q =  -12578.947368421053
... solve successful.
Simulating with Q =  -10368.421052631578
... solve successful.
Simulating with Q =  -8157.894736842105
... solve successful.
Simulating with Q =  -5947.368421052632
... solve successful.
Simulating with Q =  -3736.8421052631566
... solve successful.
Simulating with Q =  -1526.3157894736833
```

```
... solve successful.
Simulating with Q =  684.21052631579
... solve successful.
Simulating with Q =  2894.7368421052633
```

Inline Exercise: Repeate the exercise above, but create a figure showing the heat duty vs. the mole fraction of Benzene in the vapor outlet. Remove any unnecessary printing to create cleaner results.

```
In [20]: # Todo: generate a figure of heat duty vs. mole fraction of Benzene in t
         Q = []
         V = []

         for duty in np.linspace(-17000, 25000, 20):
             # fix the heat duty
             m.fs.flash.heat_duty.fix(duty)

             # append the value of the duty to the Q list
             Q.append(duty)

             # solve the model
             status = solver.solve(m)

             # append the value for vapor fraction if the solve was successful
             if solve_successful(status):
                 V.append(value(m.fs.flash.vap_outlet.mole_frac[0, "benzene"]))
             else:
                 V.append(0.0)
                 print('... solve failed.')

         plt.figure("Purity")
         plt.plot(Q, V)
         plt.grid()
         plt.xlabel("Heat Duty [J]")
         plt.ylabel("Vapor Benzene Mole Fraction [-]")
         plt.show()
```



Recall that the IDAES framework is an equation-oriented modeling environment. This means that we can specify "design" problems natively. That is, there is no need to have our specifications on the inlet alone. We can put specifications on the outlet as long as we retain a well-posed, square system of equations.

For example, we can remove the specification on heat duty and instead specify that we want the mole fraction of Benzene in the vapor outlet to be equal to 0.6. The mole fraction is not a native variable in the property block, so we cannot use "fix". We can, however, add a constraint to the model.

Note that we have been executing a number of solves on the problem, and may not be sure of the current state. To help convergence, therefore, we will first call initialize, then add the new constraint and solve the problem. Note that the reference for the mole fraction of Benzene in the vapor outlet is `m.fs.flash.vap_outlet.mole_frac[0, "benzene"]`.

Inline Exercise: Fill in the missing code below and add a constraint on the mole fraction of Benzene (to a value of 0.6) to find the required heat duty.

In [21]:
```
# unfix the heat duty
m.fs.flash.heat_duty.unfix()

# re-initialize the model - this may or may not be required depending on
m.fs.flash.initialize()

# Todo: Add a new constraint (benzene mole fraction to 0.6)
m.benz_purity_con = Constraint(expr= m.fs.flash.vap_outlet.mole_frac[0,

# solve the problem
status = solver.solve(m, tee=True)

# print the value of the heat duty
print('Q =', value(m.fs.flash.heat_duty[0]))
```

```
2019-05-13 14:38:57 - INFO - idaes.property_models.ideal.ideal_prop_pa
ck_VLE - Starting fs.flash.control_volume.properties_in initialisation
2019-05-13 14:38:57 - INFO - idaes.property_models.ideal.ideal_prop_pa
ck_VLE - Starting fs.flash.control_volume.properties_out initialisatio
n

Ipopt 3.12.12:

**********************************************************************
********
This program contains Ipopt, a library for large-scale nonlinear optim
ization.
 Ipopt is released as open source code under the Eclipse Public Licens
e (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
(http://projects.coin-or.org/Ipopt)
```

```
**********************************************************************
********

This is Ipopt version 3.12.12, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt document
ation).

Number of nonzeros in equality constraint Jacobian...:      141
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:       53

Total number of variables............................:       44
                     variables with only lower bounds:        3
                variables with lower and upper bounds:       10
                     variables with only upper bounds:        0
Total number of equality constraints.................:       44
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du al
pha_pr  ls
   0  0.0000000e+00 6.88e-03 1.00e+00  -1.0 0.00e+00    -  0.00e+00 0.
00e+00   0
   1  0.0000000e+00 5.39e-02 1.03e-02  -1.0 1.02e+03    -  9.90e-01 1.
00e+00H  1
   2  0.0000000e+00 5.60e-10 2.30e-04  -1.0 2.01e-01    -  9.90e-01 1.
00e+00h  1

Number of Iterations....: 2

                                   (scaled)                 (unscaled)
Objective...............:   0.0000000000000000e+00    0.00000000000000
00e+00
Dual infeasibility......:   0.0000000000000000e+00    0.00000000000000
00e+00
Constraint violation....:   7.0305080713992107e-12    5.60248736292123
79e-10
Complementarity.........:   0.0000000000000000e+00    0.00000000000000
00e+00
Overall NLP error.......:   7.0305080713992107e-12    5.60248736292123
79e-10


Number of objective function evaluations             = 4
Number of objective gradient evaluations             = 3
Number of equality constraint evaluations            = 4
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 3
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 2
Total CPU secs in IPOPT (w/o function evaluations)   =      0.022
Total CPU secs in NLP function evaluations           =      0.000

EXIT: Optimal Solution Found.
Q = 6455.280946055221
```

```
In [ ]:
```