# Table of Contents

[Table of Contents 1](#_Toc147910907)

[List of Figures 1](#_Toc147910908)

[List of Tables 2](#_Toc147910909)

[Introduction 2](#_Toc147910910)

[Getting Started 3](#_Toc147910911)

[Installing Software 3](#_Toc147910912)

[Installing Libraries 3](#_Toc147910913)

[Formatting 4](#_Toc147910914)

[Importing and Calling Libraries 4](#_Toc147910915)

[Variable Definitions 5](#_Toc147910916)

[Print Statements 6](#_Toc147910917)

[Model Structure 6](#_Toc147910918)

[Component Layer 8](#_Toc147910919)

[Sub-system Layer 10](#_Toc147910920)

[System Layer 12](#_Toc147910921)

[Data Transfer 12](#_Toc147910922)

[Model Logic 13](#_Toc147910923)

[Model Logic for Scenarios 1-2 14](#_Toc147910924)

[Model Logic for Scenarios 3-4 16](#_Toc147910925)

[Using the Model 18](#_Toc147910926)

[Developing Component Models 18](#_Toc147910927)

[Integrating Component Models into Sub-System Models 20](#_Toc147910928)

[Defining Inputs 20](#_Toc147910929)

[FAQ 21](#_Toc147910930)

[Appendix 21](#_Toc147910931)

[Stream Tag Information 22](#_Toc147910932)

[Components 22](#_Toc147910933)

# List of Figures

[Figure 1: Printed sample list of installed libraries from Spyder’s console 4](#_Toc145336792)

[Figure 2: Model structure 7](#_Toc145336793)

[Figure 3: Example module directory tree for P2P system 8](#_Toc145336794)

[Figure 4: Example control volume diagram of AEZ stack in AEZ sub-system 8](#_Toc145336795)

[Figure 5: Example control volume of AEZ sub-system within the P2P system 10](#_Toc145336796)

[Figure 6: Example control volume diagram of P2P system showing the modeled sub-systems 11](#_Toc145336797)

[Figure 7: Stream class definition in sub-system input module 12](#_Toc145336798)

[Figure 8: Relevant inputs to be defined in input module for relevant subsystem 13](#_Toc145336799)

[Figure 9: Definition of components to solve and absolute paths of module locations 15](#_Toc145336800)

# List of Tables

[Table 1: Stream tag table 16](#_Toc146700641)

[Table 2: Component development summary 16](#_Toc146700642)

# Introduction

The following document details the codes and structure used in developing physical models of a power-to-gas-to-power (P2P) system in Python 3.9.12 within Spyder IDE 5.1.5. The modeled components can be extended to other physical systems. The developed models are to be tuned to the D2E P2P experimental equipment, but also designed to be easily tuned to other future equipment.

The model developments are focused on two different aspects: steady state and dynamic operation. Steady state models are useful for sizing components, understanding general system interactions at design and off-design points, and understanding design point performance characteristics. Dynamic modeling is useful for understanding how a system will react to transient inputs (i.e., changing solar power input), output demand, and/or how system control strategies can be improved to yield improved performance. Since steady state inputs are typically used as the initial conditions for a dynamic model, current efforts are focused on developing steady state models. The steady state models are based on 0-D mass and energy balances. The steady state models are to consist of a series of equations that can be solved using fsolve from Python’s scipy.optimize library.

The current modeling approach is to develop a series of physical and black box models. Physical modeling efforts will focus on critical components where such an effort can lead to improved understanding of how physical phenomena affect system performance (i.e., electrolyzer and fuel cell stacks). Black box models will be used in instances where “off-the-shelf” components are used, the governing physics are already understood, and program work product is unlikely to yield suggestions in how component operation should change (i.e., chiller systems used to provide cooling to the electrolyzer and fuel cell systems). The in-progress P2P system models are available at <https://github.com/rjflores2/D2E>.

# Getting Started

The models are developed using the Python programming language. Python is an open-source programming language with many integrated development environments (IDEs). This section details getting started in the Spyder IDE which is referred to as the scientific development environment designed for scientists, engineers, and data scientists. The provided downloads and instructions are all based on a Windows operating system (OS). No details are provided for any other OS.

## Installing Software

The Spyder IDE can be accessed via the Anaconda Repository which contains other IDEs but also included built-in libraries which eliminates installation steps. The download for the Anaconda Repository can be accessed via this [link](https://www.anaconda.com/products/distribution). Once installed, the Spyder IDE can be opened via the main window that opens from running Anaconda.

The Spyder IDE can also be downloaded directly without downloading the Anaconda Repository, but built-in libraries will likely need to be installed such as NumPy, SciPy, and matplotlib. The download for the direct Spyder IDE can be accessed via this [link](https://www.spyder-ide.org/).

## Installing Libraries

Python has a wide range of libraries that can be used for different purposes. The standard ones included with Anaconda that are used in the model include NumPy, SciPy, and matplotlib. These are libraries that do not need to be installed when using Spyder within Anaconda. Details on how to use these and other libraries will be described and exemplified in the following section, [Formatting](#_Formatting).

Non-standard libraries that are used are [CoolProp](http://www.coolprop.org/coolprop/HighLevelAPI.html) and [Cantera](https://cantera.org/documentation/index.html). These are libraries to access thermodynamic properties to be incorporated into the developed models. Both can do similar things, but Cantera has been primarily used to access thermodynamic constants. CoolProp is used to access specific parameters at a given condition such as the enthalpy of a gas at a known temperature and pressure. Since these are non-standard libraries in Anaconda, these must be installed in the Console of Spyder. The following formatting has been successful in installing the respective library.

conda install conda-forge::coolprop

conda install --channel cantera cantera

All installed libraries can be viewed and verified via a short command in the Console of Spyder. If you have successfully installed the libraries, they will appear in the printed list. Once the libraries are installed, there is no need to do any future installations unless Anaconda is uninstalled from the operating computer.

conda list

A screenshot of a computer screen

Description automatically generated with medium confidence

Figure : Printed sample list of installed libraries from Spyder’s console

Long term development of the models will require switching from CoolProp to REFPROP to include all encompassing thermodynamic properties that may not be normally included in CoolProp.

# Formatting

To facilitate ease of use for other users, a consistent formatting method is implemented. This section applies to individual lines of code and specific segments within a modeled component. The details of the following sections are for both user clarity and code organization (minimizing line count). Images with examples for each section are shown below.

In general, similar variables can be grouped together by using a semi-colon to start a new line. This helps organize similar variables to track their location in the code.

## Importing and Calling Libraries

In the Python programming language, it is a requirement to import directories, functions, and toolboxes from their respective libraries. All items should be imported at the beginning of the code, not imported as needed throughout the code. Items only need to be imported once. This import should be done in the input file of a modeled subsystem or system.

Referencing these imported libraries is also required when calling the functions/properties within the library. The general format for importing a library is shown below where the name of the imported library can be rewritten as a shorter name to make it easier when referencing.



The imported library then needs to be referenced when calling a function/property from that library.

A computer screen with white text

Description automatically generatedAlternatively to calling a function/property while referencing the library, users can import a function/property directly from the library. This is useful when you want to rename a specific function/property and won’t be using many others from that same library.

A screen shot of a computer

Description automatically generated

## Variable Definitions

Variables should be named with clarity and briefness. Variables with the same units should be grouped on the same line if definitions are short. The pound sign “#” is used for short comments. This is useful to describe the defined variables, for example in including the variables units.

A computer screen with green text

Description automatically generated

## Print Statements

In the Python programming language, the command “print” must be used for each item you desire to see in the. In Spyder, you see the print statements in the Console window. The semi-colon is useful to start a new line and maintain the print statement within the same line of the item you desire to print. The print statement can be commented out with the pound sign to prevent many statements from printing at once. This strategy is useful when debugging because the print statements can always be locally located but commented out until needed for debugging.

A computer screen with white text

Description automatically generated

# Model Structure

The model is developed in layers where the inner-most layer (Component Layer) contains fundamental models and governing equations that are to be solved for each component such as a cell stack, heat exchanger, pump, etc. The next layer (Subsystem Layer) integrates all the modeled components and allows for inputs and outputs to flow between the components of one physical piece of equipment (subsystem) such as an electrolyzer or fuel cell. The outer-most layer (System Layer) integrates all the physical subsystems into the overall energy system and is solved using the inputs and outputs from the smaller systems such as a P2P system, gas turbine power plant, fueling station, etc.

Each layer is designed to run independently from the other to solve specific components and specific subsystems. To accomplish this, it is required to call and execute specific modules for a given layer. Figure 2 shows that some components will require an individual module for the parameters, electrochemical model, and mass/energy balance. Each block represents one executable python module, a module ending in “.py”. Other components may not have as many parameters or governing equations and thus individual modules are not necessary.

The model is built around the current working directory ([CWD](https://sites.pitt.edu/~naraehan/python3/file_path_cwd.html#:~:text=The%20concept%20of%20Current%20Working,top%20(e.g.%2C%20'alice.)) concept to minimize the need for users to specify the absolute path of a .py module. The CWD is the folder that Python is operating inside. Thus, when modules are in a different folder, they are in a different directory than the CWD. The CWD must be continuously updated to match the module organization. Based on the structure shown in Figure 2, this means that the associated modules detailed in Sub-system Layer section would all be in the “subsystem” folder. An example module directory tree is shown in Figure 3 which shows how the CWD must be changed to open and execute modules with only a relative path, not specifying absolute paths. For example, a relative path is just the module name if it is in the same CWD. It is important to note that even if a folder is located within the CWD, the CWD must be changed to that folder to access modules in that folder. For example, the “AEZ” directory shown in Figure 3 is the CWD when the model is first ran via the “AEZ.py” module, but the CWD must be changed to “AEZ\_stack” and “AEZ\_BoP” to execute the modules in each respective directory.

Changing the CWD is accomplished via the “OS” module in Python. The “OS” module must be imported and the “chdir” function can be used to change the CWD. At the end of each component model, the CWD must be changed back to the respective subsystem folder.

A green and black background with white text

Description automatically generated

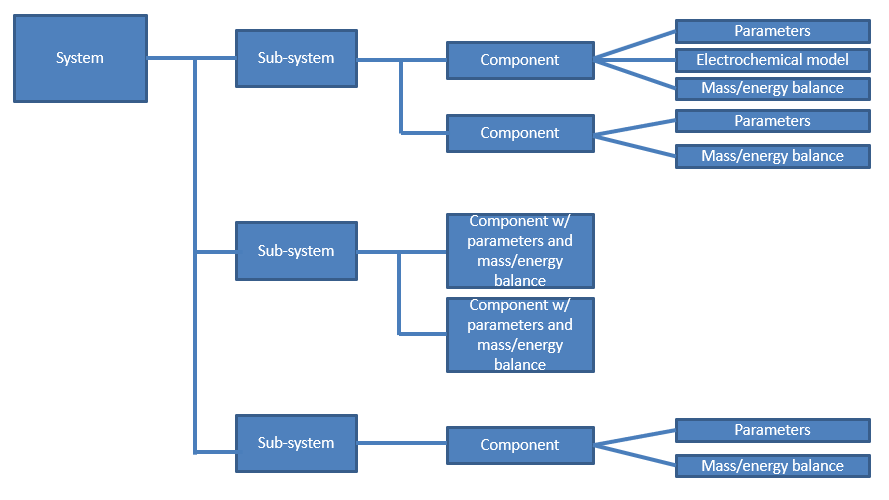


Figure : Model structure

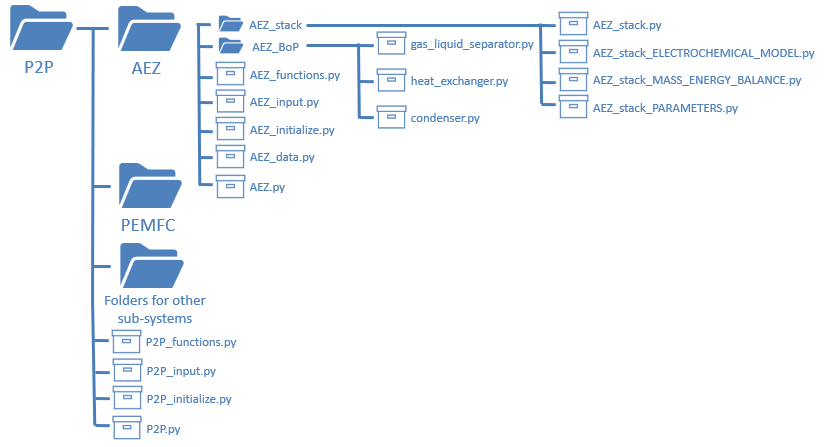


Figure : Example module directory tree for P2P system

## Component Layer

The component layer is the basis for the entire model. The solutions from each component are the same solutions that will be used in the outer layers. All components will have a minimum of 2 sections, parameters and mass and energy balances. Some components will have additional sections such as electrochemical models for cell stacks. As previously mentioned, depending on the amount of parameters, governing equations, calculations, etc, one or multiple modules can be used for just 1 component.

Figure 4 shows an example control volume that should be created for each modeled component. The mass flows into and out of the AEZ stack are shown in green while the input and output thermal energy are shown in red.

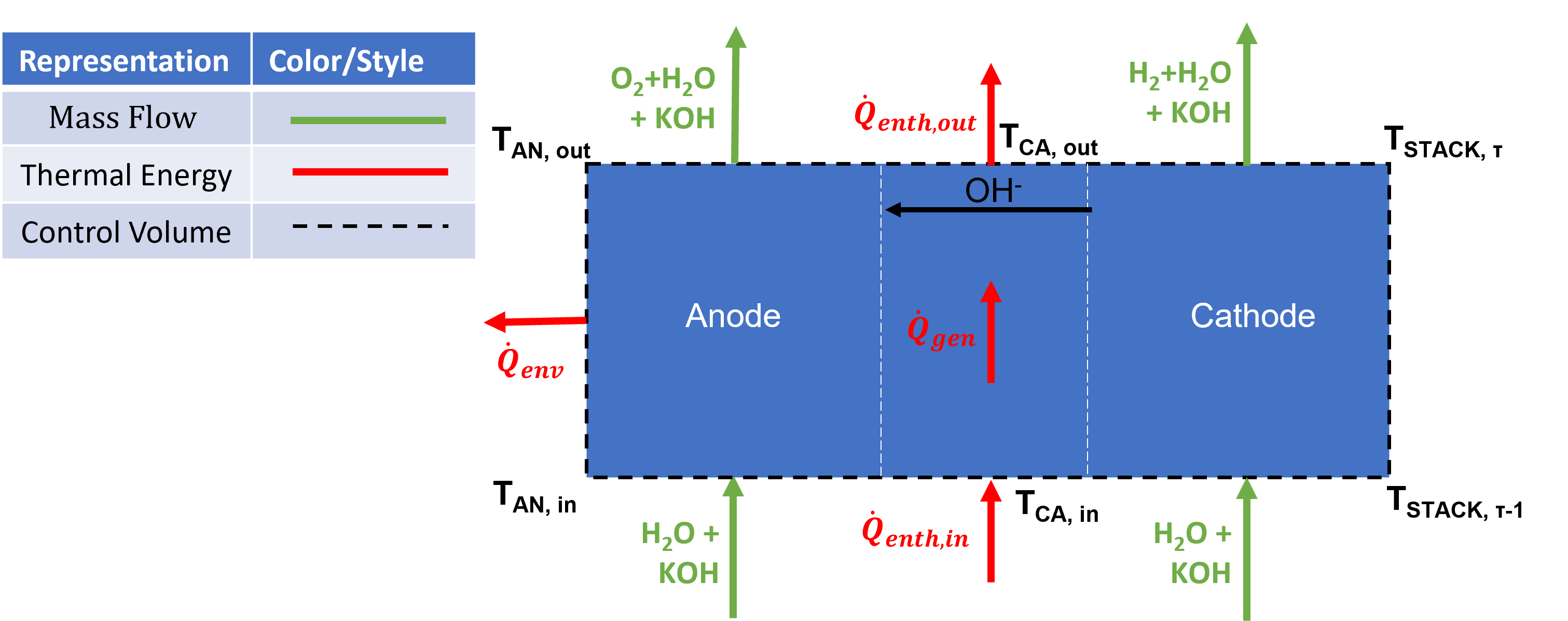


Figure : Example control volume diagram of AEZ stack in AEZ sub-system

For the sake of organization, cell stacks are one of the components to have a module for each the parameters, electrochemical model, and mass/energy balance. These individual modules are then opened and executed in a module named after the respective component. The template for naming modules for components is the following: “sub-system-name\_component-name\_SECTION”. The name for the part “SECTION” would either be “PARAMETERS”, “ELECTROCHEMICAL\_MODEL”, or “MASS\_ENERGY\_BALANCE” only if the component is complex and would benefit from individual modules for each section. The naming template for the component itself is the following: “sub-system-name\_component-name” where the required sections are included in the module and may or may not need to be called.

While components can run independently from other components, the required sections are dependent on each other. For example, the values that are first defined in the “PARAMETERS” section are then required in the “ELECTROCHEMICAL\_MODEL” section and then those values are required in the “MASS\_ENERGY\_BALANCE” section. Thus, if separate modules are used for each section of a component, they must be called and executed within a separate module for that component. These modules are shown in the example directory for “AEZ\_stack” in Figure 3.

A screen shot of a computer code

Description automatically generated

## Sub-system Layer

The sub-system layer compiles all the components to be used in that physical sub-system. This is where an initial component is chosen for a range of chosen components within a specific sub-system such as an electrolyzer or fuel cell. Figure 5 shows all the components within the AEZ sub-system to be modeled including gas-liquid separators, condensers, heat exchangers, pumps, drying system, and de-oxidation (DE-OX) system.

A diagram of a supply system

Description automatically generated

Figure : Example control volume of AEZ sub-system within the P2P system

Each sub-system layer will have 4 associated modules, 1 input module, 1 initialize module, 1 data module, and 1 execution module. The input module is required to run the initialize module, thus it is called prior to the initialize module. The input module is where the input conditions of the initial component are defined. The required input parameters and other information will be explained in Data Transfer and Using the Model sections. The initialize module is the written code containing the model logic to facilitate 1 component’s output to be the input into another component. No edits should be made to the initialize module, all edits/changes are completed in the input module. The data module is where plots and Excel sheets are generated to view and access the data. The data module uses results from the initialize module and thus must be executed after the initialize module. The execution module is a file entitled as the specific subsystem/system where the 3 previously mentioned modules are all called and executed. The naming formats for the input, initialize, data and execution modules are “subsystem\_input” and “subsystem\_initialize”, “subsystem\_data” and “subsystem” respectively. This is exemplified in Figure 3 where the AEZ subsystem and P2P system each have associated input, initialize, and data modules.

## System Layer

The system layer will function similarly to the sub-system layer except the inputs and outputs of sub-systems will be integrated for the entire system.

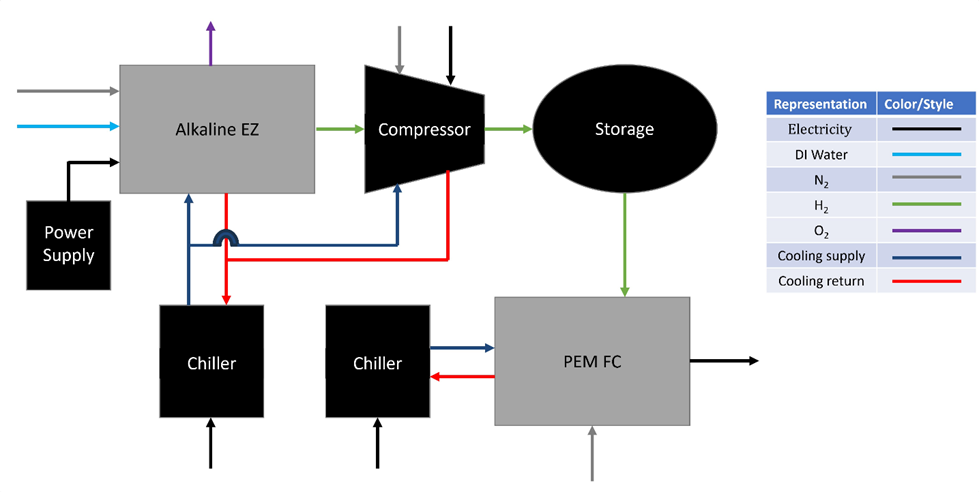


Figure : Example control volume diagram of P2P system showing the modeled sub-systems

# Data Transfer

Object oriented programming ([OOP](https://realpython.com/python3-object-oriented-programming/)) is used to pass inputs and outputs from one component or subsystem into another. The benefit of OOP is data is much more accessible because of the ability to search for a specific parameter within the created object. Also, OOP is not order dependent compared to accessing data within matrices where the order must be known to index and obtain a value. The goal of using OOP to facilitate the easy addition of new stream species into a specific system. For example, adding methane as a species would be useful for systems that integrate a solid oxide subsystem.

Although the primary application of OOP in the model is to facilitate the flow of inputs and outputs between components, it is also used for other applications. However, these other applications are not critical to the understanding of how the model works and subsequently will not be discussed.

In the Python programming language, a class can be defined as a data structure like lists and arrays of number and strings can be defined. Each class can have attributes, or parameters, that provide information about a certain subject. A defined class can have class attributes and instance attributes. Class attributes are used to define properties that should have the same value for every class. Instance attributes are used to define properties that vary from one instance to another. For example in Figure 7, the class entitled “Stream” is defined and its initial class attribute is defined as the first component to be solved and instance attributes are defined as the parameters that change from stream to stream. The parameters in the present Stream class definition are the stream tag number (s), current density (i), component index (c), temperature (T), pressure (P), molar flow rate (N), molar concentration of KOH (x\_KOH), molar concentration of liquid water (x\_H2O\_l), molar concentration of water vapor (x\_H2O\_v), molar concentration of gaseous hydrogen (x\_H2), molar concentration of gaseous oxygen (x\_O2), and concentration of gaseous nitrogen (x\_N2).

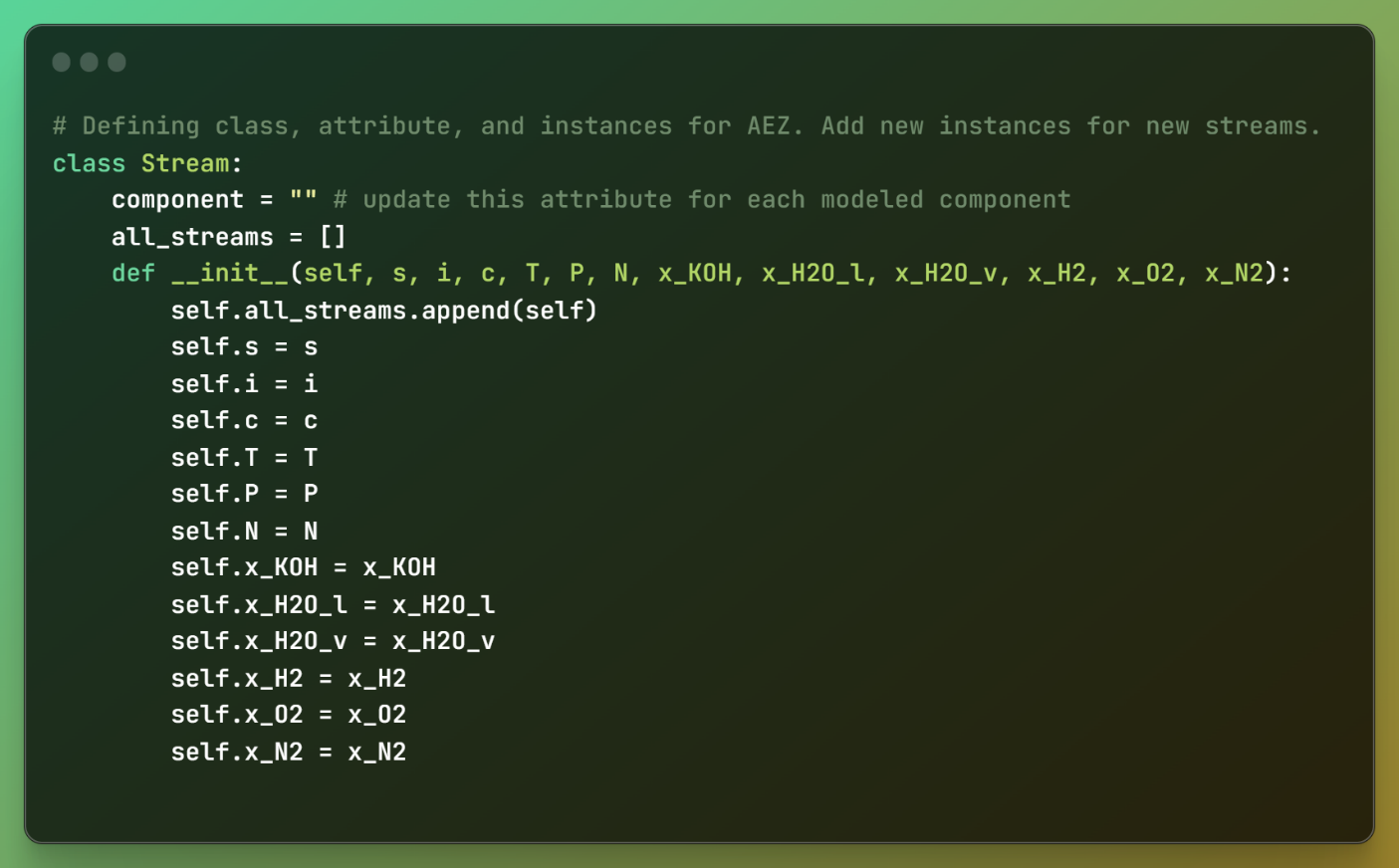


Figure : Stream class definition in sub-system input module

# Model Logic

This section is specifically for users to understand how the model solves systems and subsystems. A line by line explanation of the coded logic in the initialize module will be provided for users. The logic is currently only completed for solving multiple components of a desired subsystem, but will be similar for solving multiple subsystems of a desired system.

The logic can currently operate under 4 different scenarios: 1) 1 current density for 1 component, 2) 1 current density for range of components, 3) a range of current densities for 1 component, and 4) a range of current densities for a range of components. The user’s desired scenario is chosen in the relevant input file by changing the parameters *cs\_index\_start*, *cs\_index\_stop*, *i\_index\_start*, and *i\_index\_stop* which changes the amount of components and current densities to solve. More detail on these parameters is in the Defining Inputs section of this document. The logic for scenarios 1 and 2 is shown in Figure 8 and scenarios 3 and 4 is shown in Figure 9. It is important to recognize that current density is only a relevant input for electrochemical subsystems, but the same logic can be used for non-electrochemical subsystems by changing another input parameter such as inlet temperature, mass flow rate, etc rather than current density. This feature will be included in the future.

For all scenarios, the first 3 lines are used to define 2, 3Ds array to be used for storing the values of each instance of each inlet and outlet stream. These values are assigned to the parameters in the class definition like shown in Figure 7. A 3D array was chosen as the storage method because it most easily represents a .csv file with multiple sheets. The amount of sheets correspond to the amount of components of a subsystem, the amount of rows of inlet streams corresponds to the total amount of inlet streams of all components and similarly for the amount of outlet streams, and the amount of columns corresponds to the amount of parameters in an inlet/outlet stream, which is constant for 1 system. Any additional parameters that are not included in the users Stream class must be stored during post processing in the relevant data module. For example, polarization curves are a critical plot to obtain for electrochemical components, but cell voltage is not a parameter stored because it is only critical for those specific components. Cell voltage may be calculated for certain conditions during the solving of the component, but it is then required to calculate it again in the data module to store the value, plot it, etc. Thus, it is recommended to create functions for parameters that may be desired in post processing such as cell voltage, heat transfer rates, etc.

## Model Logic for Scenarios 1-2

After the 3D storage array has been defined, the proceeding 12 lines are to solve the first component of the array *components\_list* defined in the relevant input module. An empty array, *streams\_out*, is defined and populated with the only known parameters of *s* and *i* corresponding to the outlet stream tags and current density, respectively. The other known parameter is *c* the component index which is already defined in the input module and does not need to be updated. All other parameters are assigned a 0 value to later be replaced. Zeros are used to populate the *Stream* class because a class structure cannot be defined with empty instances. The 3D storage array for all inlet streams is then populated with the input streams. A directory change then occurs to reflect that the models to solve are stored in different folders relative to the initialize module. The initial component is then opened and executed and its output streams are stored. Scenario 1 then ends and scenario 2 can then begin if there are other components to solve in *components\_list*.

The initial for loop proceeding the conditional check for multiple components is to update next component to solve and redefine an empty array, *streams\_in,* to store the output streams of the previous component as the input streams into the next component. The next nested for loop iterates through each previous outlet stream and utilizes another conditional statement to check if any outlet stream tags of the previous component are the same as the inlet stream tags of the next component. If true, the array *streams\_in* is populated with the *Stream* class and the 3D storage array is updated. The array *streams\_out* is then redefined as empty and populated like for the first component. The next component is then solved and its outlet streams are stored and the loop proceeds to the next component.

A screen shot of a computer code

Description automatically generated

Figure : Logic for solving sceanrios 1-2 from initialize module

## Model Logic for Scenarios 3-4

Compared to the logic shown in Figure 8, the model logic shown in Figure 9 immediately enters a nested for loops to solve the users inputted array, *components\_list.* The order of the nested for loops is the following from outermost to innermost: *current\_range, components\_list, streams\_out.* This represents the physical situation of multiple components of an electrochemical subsystem requiring to be solved at 1 specific current density. For example, it is less realistic to solve an electrolyzer stack for all inputted current densities and then use the output of the from the last current density as the input into the next component. It is desirable to “travel” through the electrochemical subsystem (*components­\_list*) at 1 current density and then travel through again at another current density, and repeat this loop for the *current\_range*. The logic currently assumes the starting streams defined in the input module are always the same for each current density value.

The next for loop iterates over *components\_list* at 1 current density.

A screen shot of a computer code

Description automatically generated

Figure : Logic for solving scenarios 3-4 from initialize module

# Using the Model

The following sections are to provide general guidance on how to use the different layers of the model.

## Developing Component Models

Depending on the specific component that is being developed, 2-3 separate python modules for 1 component or 1 python module with internal sections for 1 component should be created and the naming format in the Component Layer section should be followed. When developing component models, it is necessary to run the model to verify calculations result in expected values. It is recommended to isolate the component model from other component models, in other words only the component model should be ran when being developed. To achieve this, the inputs into the component can be temporarily defined in the component model when in reality, these inputs will be defined in the subsystem model.

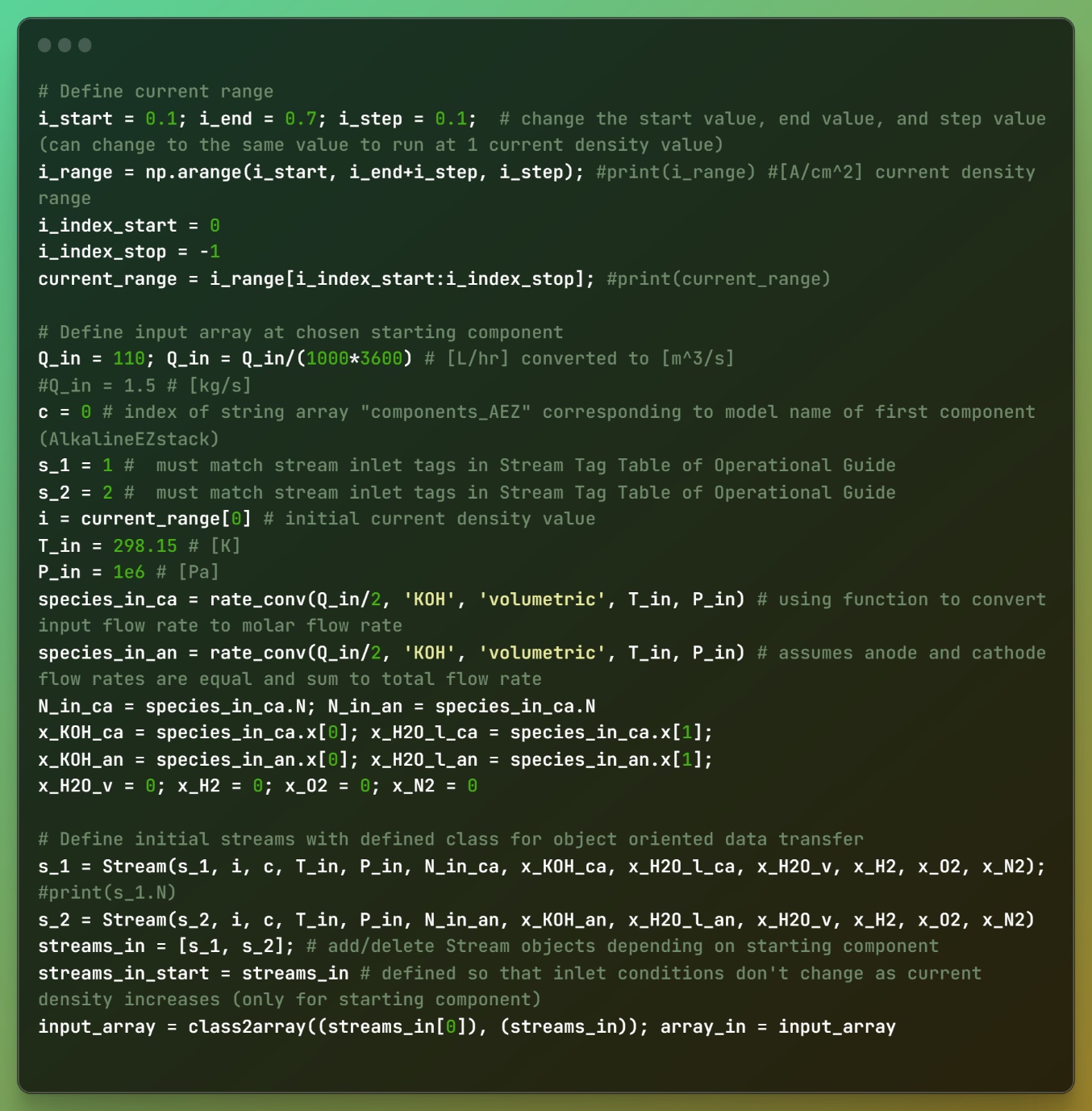


Figure : Relevant inputs to be defined in input module for relevant subsystem

For clarity and ease of use when developing component models, it is recommended to write equations in terms of the variables that are specific to the given component. For example, the inlet temperature for the cathode of a cell stack should be defined as “T\_in\_ca”. This variable name will eventually be redefined when the component model is to be integrated into a sub-system model.

Recommended order of operations when developing:

1. Define 1 section for each of the required component sections: parameters, electrochemical model (if required), mass-energy balances.

## Integrating Component Models into Sub-System Models

For every system, the species of each stream that flow through the components of every subsystem are known. These knowns species are used to define a class data structure to pass the relevant data about each stream from component to component. This class data structure is defined in the associated input module of each sub-system. Figure 7 shows the definition for the AEZ sub-system of the P2P system. The *Stream* class data structure of each subsystem should be defined the same within a specific system.

## Defining Inputs

The variables defined in the input module, whether it be “subsystem\_input.py” or “system\_input.py”, must be examined prior to running the model. Critical variables to change include the variables “path\_system”, “path\_subsystem”, “cs\_index\_start”, “cs\_index\_stop”, and the relevant physical parameters for the chosen starting component. An example of how these variables are defined in the “AEZ\_input.py” module is shown above in Figure 8 and below in Figure 11.

A screenshot of a computer program

Description automatically generated

Figure : Definition of components to solve and absolute paths of module locations

# FAQ

# Appendix

## Stream Tag Information

Table : Stream tag table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Subsystem | Component | Inlet Tags | Composition | Outlet Tags | Composition |
| AEZ.py | AEZ\_stack.py | 1: cathode | KOH | 3: cathode | H2+H2O(g)+KOH |
| 2: anode | KOH | 4: anode | O2+H2O(g)+KOH |
| gas\_liquid\_separator.py  (operated as H2-KOH separator) | 3: process | H2+H2O(g)  + KOH | 5: process | H2+H2O(g)) |
| 7: condenser | H2O(l) | 6: liquid | KOH |
| 8: feedwater | H2O(l) |
| gas\_liquid\_separator.py  (operated as O2-KOH separator) | 4: process | O2+H2O(g)  + KOH | 11: process | O2+H2O(g) |
| 9: condenser | H2O(l)) | 12: liquid | KOH |
| 10: feedwater | N/A |
| pos\_disp\_pump.py | 20: feedwater | H2O(l) | 21: feedwater | H2O(l) |
| heat\_exchanger.py  (H2 side) |  |  |  |  |
|  |  |  |  |
| Heat\_exchanger.py  (O2 side) |  |  |  |  |
|  |  |  |  |
| H2\_compressor.py | N/A | 40: process | H2 | 42: process | H2 |
| 41: CW supply | H­2O(l) | 43: CW return | H2O(l) |
| H2\_compressed\_storage. py | N/A |  |  |  |  |

## Components

Table : Component development summary

|  |  |  |  |
| --- | --- | --- | --- |
| Individual | Component | Progress?  Not Started (NS)/ In progress (IP)/ Complete (C) | Successfully Integrated?  Yes (Y)/ No (N) |
| Bobby | AEZ\_stack.py | C | Y |
| Jasper | PEMEZ\_stack.py | IP | N |
| Esteban | PEMFC\_stack.py | IP | N |
| Bobby | compressed\_storage.py | IP | N |
| Bobby | H2\_compressor.py | IP | N |
| Jasper | heat\_exchanger.py | IP | N |
| Bobby | gas\_liquid\_separator.py | IP | Y |
|  | heater.py | NS | N |
|  | centrifugal\_pump.py | NS | N |
|  | pos\_disp\_pump.py | NS | N |

Condenser – Bobby, Heat exchanger – Jasper, Air cooled chiller (operate as HX) -