# Table of Contents

[Table of Contents 1](#_Toc129186379)

[List of Figures 1](#_Toc129186380)

[List of Tables 1](#_Toc129186381)

[Introduction 2](#_Toc129186382)

[Getting Started 3](#_Toc129186383)

[Installing Software 3](#_Toc129186384)

[Installing Libraries 3](#_Toc129186385)

[Formatting 5](#_Toc129186386)

[Importing and Calling Libraries 5](#_Toc129186387)

[Variable Definitions 5](#_Toc129186388)

[Print Statements 6](#_Toc129186389)

[Model Structure 7](#_Toc129186390)

[Component Layer 8](#_Toc129186391)

[Sub-system Layer 9](#_Toc129186392)

[System Layer 11](#_Toc129186393)

[Data Transfer 11](#_Toc129186394)

[Using the Model 12](#_Toc129186395)

[FAQ 12](#_Toc129186396)

# List of Figures

[Figure 1: Printed sample list of installed libraries from Spyder’s console 4](#_Toc129190808)

[Figure 2: Model structure 7](#_Toc129190809)

[Figure 3: Example control volume diagram of AEZ stack in AEZ sub-system 8](#_Toc129190810)

[Figure 4: Code image from AEZ\_stack model calling the required sections 8](#_Toc129190811)

[Figure 5: Example control volume of AEZ sub-system within the P2P system 10](#_Toc129190812)

[Figure 6: Example control volume diagram of P2P system showing the modeled sub-systems 11](#_Toc129190813)

[Figure 7: Stream class definition 12](#_Toc129190814)

# List of Tables

# 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 (the version used in Anaconda’s Spyder IDE). 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

# 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). Generic formatting examples are shown in red and examples from actual code are shown in orange.

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.

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.

import name\_of\_libaray as LIB

import CoolProp.CoolProp as CP; import cantera as CT

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

new\_variable = LIB.property

O2\_mm = CP.PropsSI('M', 'O2'); N2\_mm = CP.PropsSI('M', 'N2')

Alternatively 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.

from name\_of\_library import function as fctn

from numpy import log as ln

## 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.

variable = x+2 # [units] variable description

T\_op\_C = T\_out - 273.15 # [deg C]

## 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.

print(‘Hello world’)

i\_cell = i\*Area\_c; print('i\_cell = ', i\_cell) # [A] current input for stack

# 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 (Sub-system Layer) integrates all the modeled components and allows for inputs and outputs to flow between the components of one physical system such as an electrolyzer or fuel cell. The outer-most layer (System Layer) integrates all the physical sub-systems 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 sub-systems. To accomplish this, it may be required to call and execute specific scripts for a given layer. Figure 2 shows that some components will require an individual script for the parameters, electrochemical model, and mass/energy balance. Each block represents one executable python script, a file ending in “.py”. Other components may not have as many parameters or governing equations and thus individual scripts are not necessary.

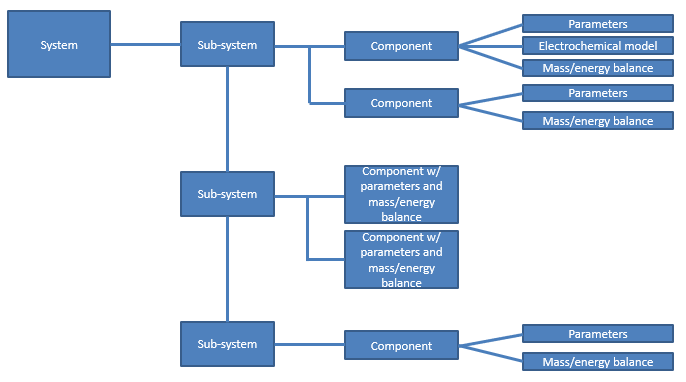


Figure : Model structure

## 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 scripts can be used for just 1 component.

Figure 3 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.

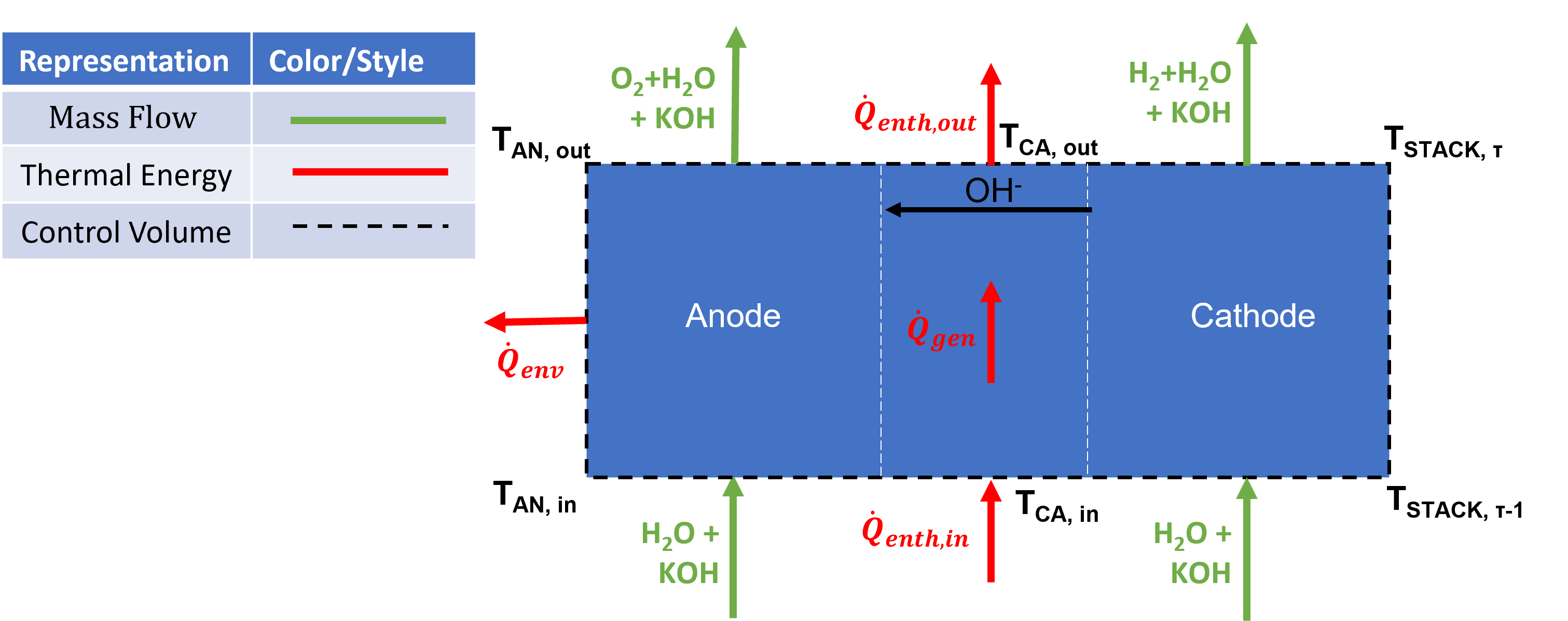


Figure : Example control volume diagram of AEZ stack in AEZ sub-system

For the sake of organization, cell stacks are one of the components to have a script for each the parameters, electrochemical model, and mass/energy balance. These individual scripts are then opened and executed in a script named after the respective component. The template for naming scripts 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 scripts 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 script and may or may not need to be called.



Figure : Code image from AEZ\_stack model calling the required sections

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 scripts are used for each section of a component, they must be called and executed within a separate script for that component.

## 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.

Each sub-system layer will have 3 associated files, 1 input file, 1 initialize file, and 1 data file. The input file is required to execute the initialize file, thus it is executed within the initialize file. The input file is where the input conditions into 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 file is the written code to allow 1 component’s output to be the input into another component. No edits need to be made to the initialize file, these are all completed in the input file. The naming formats for the input, initialize, and data files are “subsystem\_input” and “subsystem\_initialize”, and “subsystem\_data” respectively.

Diagram

Description automatically generated

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

## 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.

Diagram

Description automatically generated

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 sub-system 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 sub-system.

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), and molar concentration of gaseous oxygen (x\_O2).

Text

Description automatically generated

Figure : Stream class definition

# 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 files for 1 component or 1 python file 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, not ran via the sub-system model. 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 sub-system model.

A picture containing graphical user interface

Description automatically generated

Figure : Relevant inputs to be temporarily defined in component model. Image is from sub-system input file

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 sub-system 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 file of each sub-system. Figure 7 shows the definition for the AEZ sub-system of the P2P system. Note that the molar concentration of N2 will be added to this class definition because the PEMFC sub-system depends on N2 when operating. The Stream “class” data structure of each sub-system should be defined the same within a specific system.

# FAQ