System Requirements

These Python files were written for Python 3.7.0 and will not work with Python 2 installations. The files also require several add-on packages that I downloaded with the Anaconda3 installation, but there are other options to acquire these packages. Presently, this code only works on Windows, because of how it interfaces with NREL's compiled C++ code.

Getting Started

Download or clone NREL's System Advisor Model (SAM) [2] repository from GitHub (https://github.com/NREL/SAM). The code described in this document is in the subfolder *samples/CSP/sco2_analysis_python*.

Summary

This code interfaces with NREL's supercritical carbon dioxide (sCO_2) power cycle design, cost, and off-design performance models. The sCO_2 models are part of NREL's SAM Simulation Core (SSC) that contains the technology models for SAM. This Python code allows the user to explore the sCO_2 cycle in more detail than possible when using the sCO_2 cycle option in the Molten Salt Power Tower model. In particular, this code allows the user to:

- set more design variables than the SAM User Interface does.
- easily run, save, and plot parametric analyses on design variables.
- use component cost correlations from Carlson et al. [1] to calculate component and total cycle cost.

This document describes high-level modeling assumptions and provides examples for common cycle design analyses.

Code Organization

The *core* folder contains Python methods that streamline configurating a simulation, handling output data, and plotting. This folder also must contain a copy *ssc.dll* as explained above.

The *examples* folder contains *examples_main.py*, a Python file that performs common sCO_2 analyses by calling methods contained in the files in the *core* folder. The following section describes the examples in the Python file. You should be able to run *examples_main.py* without modifying it, and it should create output files and plots in the *examples* folder.

Modeling Overview

This code allows the user to select between the simple, recompression, and partial cooling cycle configurations. It uses simple isentropic efficiency equations for turbomachinery performance. The recuperator model assumes a counterflow configuration, and it discretizes each heat exchanger to solve its performance.

The function **get_sco2_design_parameters()** returns a Python *dictionary* containing default values for all cycle model parameters[1]. The parameters include options to either fix or let the model optimize pressure levels and the recompression fraction. The default parameters fix the cycle upper pressure and allow the model to optimize the pressure ratio and recompression fraction. We've found that for typical values for the upper pressure limit (~20-35 MPa), the model finds the optimal design pressure at the upper pressure limit. In cases where the optimizer chooses a lower pressure as the optimal value, usually the efficiency improvement is negligible while the power density significantly decreases.

The design model uses the design parameter **design_method** to provide several different options to define the recuperator performance. The default value of **design_method** is **2**, and in this case the model uses the total recuperator conductance set in design parameter **UA_recup_tot_des** and optimizes how much is allocated to each the low temperature and high temperature recuperator. If **design_method** is **3**, then the model uses design parameters to set the performance of each recuperator. For example, the design parameter **LTR_design_code** let's the user specify either the LTR conductance (1), minimum temperature difference (2) or the design effectiveness (3). If **design_method** is **1**, then the model finds the total recuperator conductance required to achieve the cycle efficiency set in design parameter **eta_thermal_des**. The optimizer will determine how much of the total conductance to allocate to each the low temperature and high temperature recuperator.

Examples

Cycle design simulation with default parameters

The simplest analysis is to solve the cycle design for pre-defined "default" design parameters. First, create an instance of the **C_sco2_sim** class. Next, pass a copy of the design parameters from **get_sco2_design_parameters** using the class member **overwrite_default_design_parameters**. Then, run the design simulation through the class member **solve_sco2_case()**. Simulation results are stored as a *dictionary* in the class member **m_solve_dict**, and member **m_solve_success** reports whether the performance code solved successfully. View these values by printing them. Finally, use the method **save_m_solve_dict** to save results to a file. **C_sco2_sim** initializes with instructions to only save the results in *json* format, but you can also save in *csv* format by changing member variable **m_also_save_csv** to *True*.

Plotting a cycle design

First, create an instance of the **C_sco2_TS_PH_plot(result_dict)** class and use a saved design solution *dictionary* as the argument. To save automatically save the plot when you create it, set the class member **is_save_plot** to *True* and assign a file name to class member **file_name**. Use class method **plot_new_figure()** to create the plot. You can use classes **C_sco2_cycle_TS_plot** and **C_sco2_cycle_PH_plot** to create individual TS and PH plots, respectively. If you change design

[1] These parameters do not define any specific commercial or proposed system but rather a set of reasonable target values based on the literature. Our intent is to have a default model and simulation results to which we can compare when modifying the underlying performance code.

parameters significantly from the default values, you may have to adjust the axis limits or annotation formatting.

Modifying the cycle design parameters to fully constrain the cycle design

After you create an instance of the **C_sco2_sim** class, you can overwrite its baseline design parameters with a *dictionary* of parameters you want to change. Design parameters that are not included in your *dictionary* keep their previously defined default values. For example, you may want to fix the pressures, recompression fraction, and recuperator parameters that are typically set by the optimizer to values from previous simulations or another reference. Use a partial dictionary to fix the recompression fraction, fix the pressure ratio, change the design method, and design recuperators to hit a target effectiveness. Next, use the **C_sco2_sim** method **overwrite_des_par_base(dictionary_new_parameters)** to modify the baseline design parameters. Then, you can run the simulation and view, save, and plot your results. Note that rerunning **solve_sco2_case()** will overwrite calculated class member like **m_solve_dict**.

Comparing two cycle designs

You can compare two cycle designs by processing the data in the saved *json* or *csv* files. You also can create a plot with that overlays both cycles on TS and PH diagrams. Create an instance of the **C_sco2_TS_PH_overlay_plot(result_dict1, result_dict2)** class and use the save design solution *dictionaries* as the arguments. Set the class member **is_save_plot** to True to save the plot. This class names the plot file based on the values of important cycle design parameters. Use class method **plot_new_figure()** to create the plot. You may need to adjust the axis limits or annotation formatting.

Running a parametric study

The **C_sco2_sim** class method **solve_sco2_parametric(list_of_partial_dictionaries)** runs a parametric using the modified cases in the input argument that is a *list* where each element is a *dictionary* that modifies some default design parameters. *Dictionary* elements in the *list* do not need to change the same parameters or even the same number of parameters, although organizing the parametrics this way may makes the results easier to understand. The parametric simulation results are stored in the class member **m_par_solve_dict**. Each item in **m_par_solve_dict** *dictionary* is a *list* with a length equal to the number of parametric runs. Each element in the *list* corresponds to the item variable type, so variables that are themselves *lists* (e.g. ‘P_state_points’) are *lists* of *lists*. Use the method **save_m_par_solve_dict** to save results to a file. To save in *csv* format change member variable **m_also_save_csv** to True. Each column in the *csv* represents a single simulation in the parametric study.

Plotting a 1D parametric study

You can plot 1D parametric studies using the class **C_des_stacked_outputs_plot(list_of_parametric_dictionaries)**. The input argument requires the parametric solution *dictionary* is in *list* format, because this class can overlay multiple parameteric solutions. This class uses variable label and unit conventions defined in

get_des_od_labels_unit_info__calc_metrics() in *sco2_cycle_ssc.py*. Note that the class member **x_var** uses “recup_tot_UA” to define the HTF temperature, which is different than the string used to set the corresponding design parameter (“UA_recup_tot_des”). The class member **y_var** sets the dependent variables, and each will have its own subplot. The class member **max_rows** determines how the subplots are configured on the figure. There are several other class members in *sco2_plot.py* that allow you to adjust formatting. Set **is_save** to *True* and specify **file_name** to save the plot, and use **create_plot()** to generate the plot.

References

[1] M. D. Carlson, B. M. Middleton, and C. K. Ho, “Techno-Economic Comparison of Solar-Driven sCO2 Brayton Cycles Using Component Cost Models Baselined with Vendor Data and Estimates,” in *Proceedings of the ASME 2017 11th International Conference on Energy Sustainability*, 2017, pp. 1–7.

[2] National Renewable Energy Laboratory, “System Advisor Model Version 2018.11.11,” 2018. [Online]. Available: https://sam.nrel.gov/download.