Experimental Testing of sCO2 Switched Bed Regenerators for Power Applications

by

Logan M. Rapp

A thesis submitted in partial fulfillment of

the requirements for the degree of:

MASTER OF SCIENCE

(MECHANICAL ENGINEERING)

At the

UNIVERSITY OF WISCONSIN – MADISON

2017

Approved by:

_____ Date: _____

Professor Gregory Nellis

Contents

List of Figures

1 INTRODUCTION

1.1 BACKGROUND

Electricity is the world's fastest growing form of end use energy and global net generation is forecast to increase 69% by 2040 from the levels of 2012 [1]. The majority of currently installed electricity generation capacity is based on thermal power cycles that couple a heat source to a thermodynamic power cycle. The thermal power cycles used in most commercial power generation facilities today are either air breathing direct-fired open Brayton cycles or indirect-fired closed Rankine cycles with water as the working fluid (coal and nuclear) [2]. While installation of renewable generation capacity is expected to continue grow, the U.S. Energy Information Administration forecasts that traditionally fired power sources (coal, natural gas, and nuclear) will provide 71% of global net generated electricity in 2040. Because electricity generation from traditionally fired thermal power cycles will continue to be a dominant source of global electricity generation for the foreseeable future, there is an economic incentive to minimize the construction costs and maximize the operating efficiency (i.e., minimize fuel consumption) of these electricity generation facilities. Additionally, with the current global movement towards implementing policies and regulations to reduce the emission of greenhouse gasses and other pollutants from electric power generation, the development of technologies to increase the efficiency (and thus reduce consumption of fossil fuels) is made even more attractive.

While coal and natural gas are expected to remain important to the global energy generation mix, the adoption of renewable energy sources is viewed as a crucial step to

reduce the emission of greenhouse gases and other pollutants to the environment. Renewables are the fastest growing source of electricity generation, and their growth is expected to continue at an average of 5.7%[1] per year through 2040 [1]. One challenge and restriction to implementing renewables such as photovoltaic solar and wind is the intermittent nature of their operation. For the electric grid to operate effectively it must be balanced; that is, electricity generation must match the load at all times. This matching has traditionally been accomplished with large base load plants, such as nuclear and coal, running near their name plate capacities while having "peaking plants" or dispatchable plants, most commonly natural gas, ramp up or down generation to match the load. Wind and photovoltaic solar without accompanying energy storage are inherently non-dispatchable because power generation only occurs when the wind is blowing or the sun is shining. Integration of these variable energy resources (VERs) sources into the grid is therefore challenging and may limit the penetration of VERs into the electric grid [3]. One renewable source that can be operated as either a dispatchable or base load source is concentrated solar power (CSP) with thermal energy storage (TES) (or CSP-TES). CSP collects solar energy and converts it to thermal energy, which can either be inexpensively stored[2] and used on demand or used immediately to generate electricity in a thermodynamic power cycle. This discussion focuses only on central receiver CSP as opposed to parabolic trough or dish/Stirling configurations because the central receiver type power plant has the potential to achieve the highest efficiencies due to the intense

[1] Non-hydropower renewables – wind, solar, geothermal, biomass, tidal
[2] Cost of thermal storage is currently $72-$240/kWh versus >$300 kWh for electrochemical battery storage [6]

solar flux and resulting high temperatures at the receiver [4]. An illustration of a central receiver (power tower) CSP plant is shown in Figure 1.1.

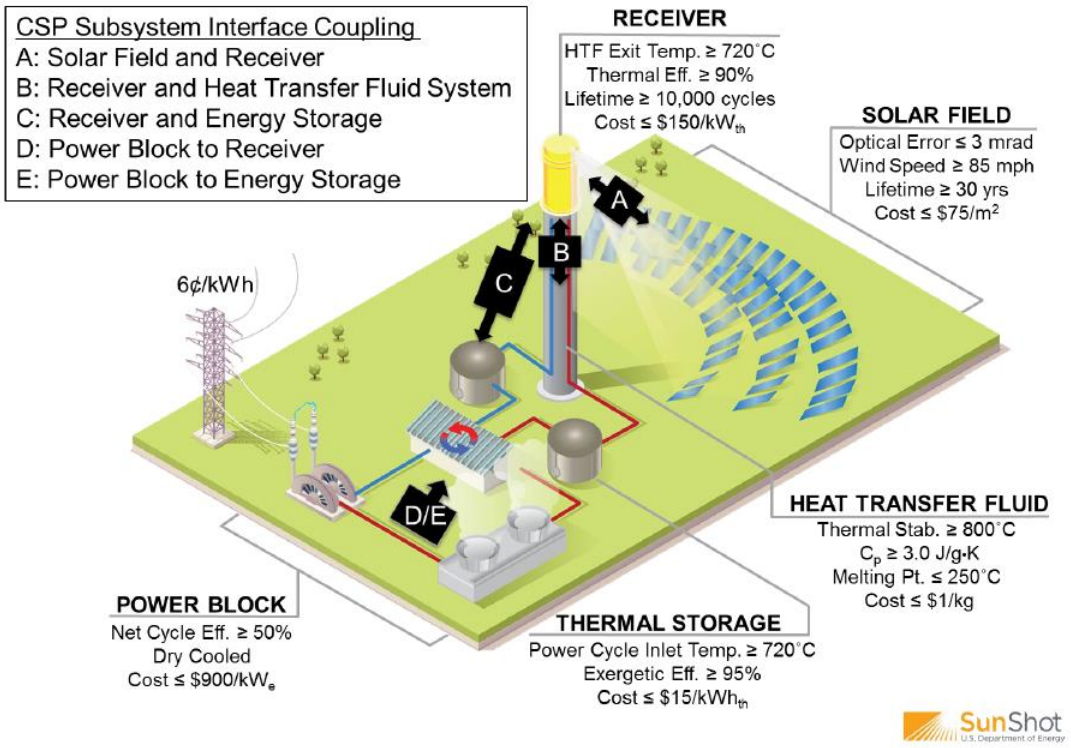


Figure 1.1 - Illustration of a Power Tower CSP plant with the subsystems of interst and SunShot goals [5]

Since the energy can be stored and used on demand, the value of dispatchable CSP-TES to the grid is increased by 5-6 cents[3] per kilowatt-hour compared with variable generation sources. Additionally, as the penetration of variable generation sources increases, the relative value of dispatchable sources (such as CSP-TES) is likely to increase [6].

[3] Estimates based on utility-scale solar energy analysis in California under 2020 33% and 40% renewables portfolio standards [6]

One technology that has the potential to improve the economics and efficiencies of both fossil fuel fired power plants as well as CSP power plants is the supercritical carbon dioxide (sCO2) Brayton cycle. As shown in Figure 1.2, the sCO2 Brayton cycle can achieve higher cycle efficiencies compared with steam Rankine cycles for turbine inlet temperatures of approximately 450 °C or greater and can achieve cycle efficiencies greater than 50% for turbine inlet temperature greater than approximately 650°C [2], [7].

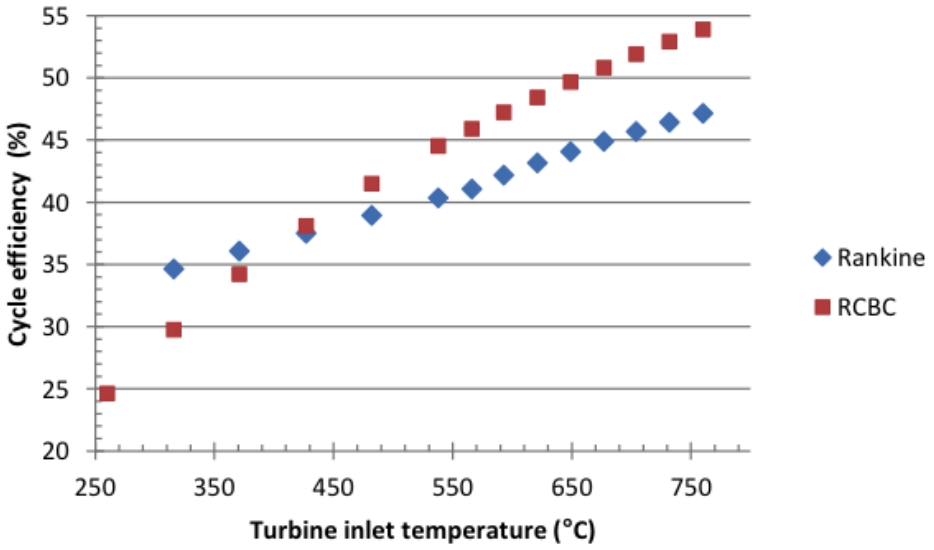


Figure 1.2 - Comparison of Steam Rankine cycle with Recompression Brayton Cycle (RCBC) [8]. Analysis performed by NETL and figure from [2].

The sCO2 Brayton cycle has also been shown to achieve comparable efficiencies with a helium Brayton cycle but at significantly lower temperatures (550 °C for CO2 vs 850 °C for helium) [7]. This efficiency gain is primarily due to the reduction of compression work achieved by the relatively high working fluid density of CO2 operating in the supercritical region as well as a high degree of thermal recuperation in the cycle. The relatively high

density of sCO2 leads to significantly smaller turbomachinery sizes compared with steam or other ideal gas turbomachinery for the same power output. Smaller turbomachinery reduces the volume of material required for construction and thus also reduces the capital cost and plant footprint. Additionally, the sCO2 Brayton cycle may be more practically suited for air cooling making it more attractive in regions where water cooling is not available or where water scarcity is a concern. However, additional research and development is needed to demonstrate the practicality of implementing air cooling with the sCO2 Brayton cycle [9]. CO2 is also non-toxic, non-explosive, non-flammable and abundant further making its use as a working fluid in a power cycle advantageous [10]. The sCO2 Brayton cycle could theoretically be implemented in any application currently operating a Rankine cycle, thus the potential applications are quite large. In general, the sCO2 Brayton cycle offers the benefit of reduced fuel consumption for plants with significant fuel costs such as coal and natural gas fired plants. For power plants with significant upfront capital investments such as nuclear and CSP, the reduced capital cost of the power block components as well as the increased electricity production for the same thermal input are the primary benefits.

1.2 SunShot Initiative and CSP: APOLLO

In order to drive the development and reduce the costs of solar energy, the United States Department of Energy (DOE) launched the SunShot Initiative in 2011. The SunShot initiative provided funding for research and development of solar technologies through collaborations between private and public entities with the goal of making solar electricity

cost competitive with conventionally generated electricity by 2020 [11]. In 2011, this meant a reduction in costs of PV and CSP by about 75%[4]. After four years and a number of successful outcomes, the CSP: Advanced Projects Offering Low LCOE Opportunities (APOLLO) program was launched in September of 2015 to specifically target CSP systems and develop technologies necessary to meet the $0.06/kWh levelized cost of energy (LCOE) cost target specified in the SunShot Initiative for CSP (shown in Figure 1.3).

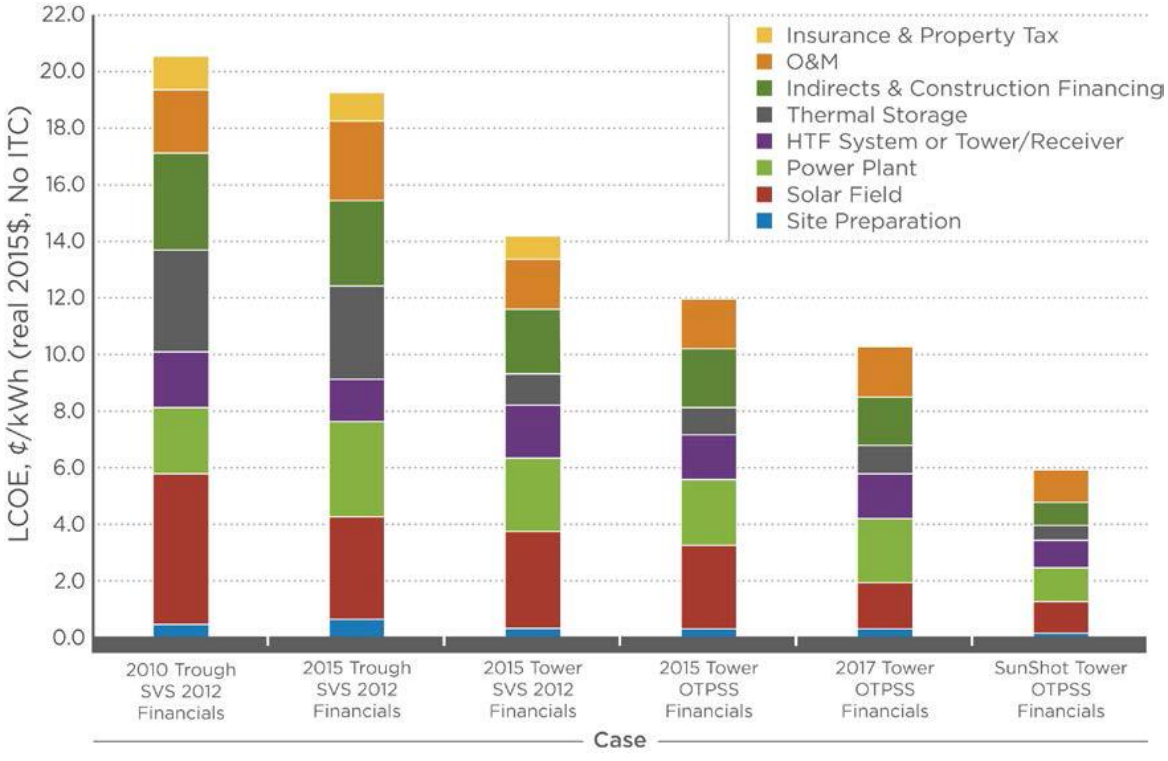


Figure 1.3 - Cost reductions for parabolic trough and tower technologies since the SunShot Vision Study [11]. SVS=SunShot Vision Study (DOE 2012). OTPSS = On the Path to SunShot.

In order to achieve the cost targets, it is estimated that the power cycle will need to achieve at least 50% thermal-to-electric power conversion and because the optimal solar resources

[4] Based on 2010 costs

are often located in arid regions and concerns over water scarcity are growing, the cooling must be accomplished without utilizing water (known as dry cooling) [11]. These requirements must be met with a total power block cost less than $900/kW. The sCO2 Brayton cycle has the potential to meet both the required thermal efficiency as well as the dry-cooling requirement and thus it appears to be a challenger to the Rankine cycle for CSP. The Rankine cycle has over a hundred years of industry operating experience and therefore a large amount of expertise exists relative to this cycle. In contrast, the sCO2 Brayton cycle is a relatively young and immature technology. Components including the sCO2 turbo machinery and heat exchangers have limited development or operational experience in the laboratory or in industry [2]. Before this technology can be commercialized, the technological risk must be reduced by demonstrating its performance. As a part of the CSP: APOLLO program, the University of Wisconsin – Madison has been awarded funding to design and test regenerative type heat exchangers for use in the sCO2 Brayton cycle. The regenerative type heat exchangers potentially offer improvements in the economics and thermodynamic performance of the sCO2 Brayton cycle compared with the recuperative style heat exchangers currently proposed, and thus may help realize the cost targets of the SunShot Initiative.

2 EXPERIMENTAL SETUP

2.1 DATA ACQUISITION EQUIPMENT

Data acquisition, display and equipment control was accomplished with Labview. A National Instruments cDAQ-9178 (Compact Data Acquisition) was used for all I/O. The cDAQ-9178 is an 8-slot, USB chassis which is capable of handling a mix of analog and digital I/O. The modules used are listed in Table 2.1.

Slot	Module Number	Type
1	NI 9265	4 Channel, 0-20mA, Analog Out
2	NI 9205	16 AI Diff/32 AI Single Inputs, programmable input range
3	NI 9476	32-Channel,24V, Digital Out
4	NI 9213	16 TC,±78mV
5	NI 9213	16 TC,±78mV
6	NI 9214	16 TC,±78mV,Isothermal Terminal Block
7	Empty	
8	Empty	

Table 2.1 - Data Acqusition Module Information

The experiment includes 48 thermocouples[5], 7 absolute pressure sensors, 4 differential pressure sensors, and 1 Coriolis flow meter. For the current experiment, pressure and flow measurements were sampled at 4 Hz and temperatures were sampled at 2 Hz. Information on the specific instrumentation is listed in Table 2.2 and the layout of the equipment and instrumentation is shown in the piping and instrumentation diagram (P&ID) in Figure 2.2.

[5] Thermocouples 33-48 are used in a separate experiment involving the PCHE recuperator and thus are not shown on the P&ID

Type	Manufacture	Span	Accuracy	Serial Number
All Thermocouples	Omega		±2.2 °C	Type K
PT01	Rosemount	0-2000 psi		3051S1CG5A2B11A1A
PT02	Omega	0-5000 psi		
PT03	Siemens	0-3000 psi	≤ 0.1%	7MF4033-1GA10-1AC8-2
PT04	Omgea	0-5000 psi		
PT05				
PT06	Siemens	0-1500 psi	≤ 0.1%	7MF4032-1GA10-1NC1-Z
PT07	Siemens	0-3000 psi	≤ 0.1%	7MF4032-1GA10-1NC1-Z
DP01	Siemens	-7 to 7 psi	≤ 0.1%	7MF4532-1GA30-1NC1-Z
DP02	Siemens	0-20 psi	≤ 0.1%	7MF4432-1HA62-1NC1-Z
DP03	Siemens	-15 to 7 psi	≤ 0.1%	7MF4553-1GA32-1AC8-Z
DP04	Siemens	-15 to 7 psi	≤ 0.1%	7MF4532-1GB30-1NC1-Z
FI01	Endress-Hauser		±0.5% o.r.	EC137D020000

Table 2.2- Instrument Information

2.2 LABVIEW

A screenshot of the Labview control screen is shown in Figure 2.1. The Labview program records all temperature, pressure, and flow data and also provides real time graphing. All equipment is also controlled from the program including the pump on/off, heater power levels, and valve switching. Some safety interlocks were also included to turn off heaters or the pump if temperature or pressure levels became unsafe.

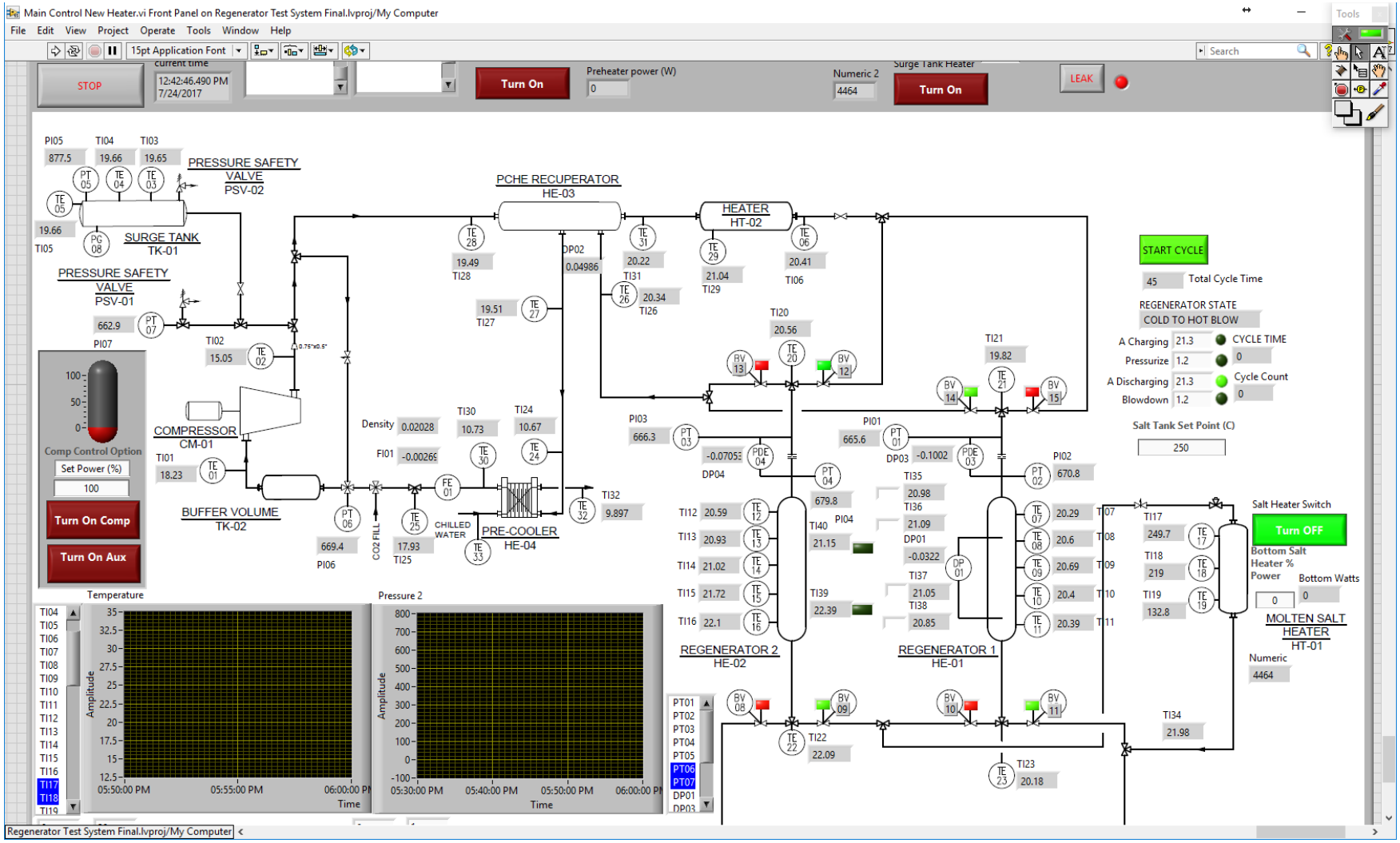


Figure 2.1 - Labview Control Screen

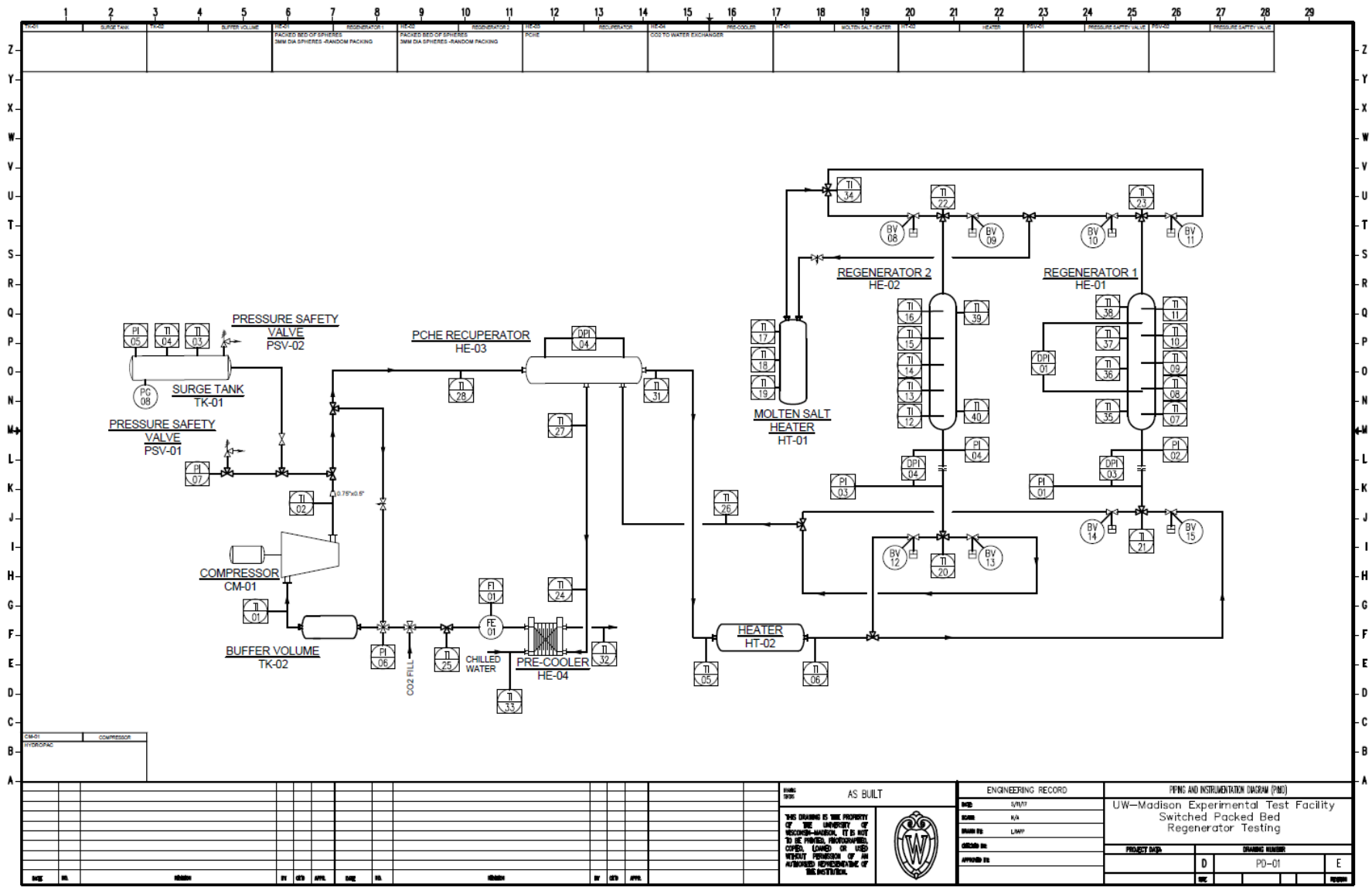


Figure 2.2 - P&ID of test facility

2.3 EXPERIMENTAL EQUIPMENT

2.3.1 Regenerators

A model of the regenerator used in the experiment is shown in Figure 2.3a and the 3mm diameter spheres used for packing are shown in Figure 2.3b.

Figure 2.3 - a) 3D model of regenerator showing temperature and pressure instrument locations (spheres in model are larger than actual spheres in order to show detail . b) Image of spheres used to fill packed bed.

As shown in figure 2-1a, there are three thermocouples placed within the packed bed and a thermocouple in each end cap of the regenerator. The three thermocouples in the packed bed are inserted such that the tip of the thermocouple is in the middle of the packed bed. This was done to capture the fluid temperature as a function of position in the packed bed. A drawing with dimensions of the regenerator is shown in Figure 2.4.

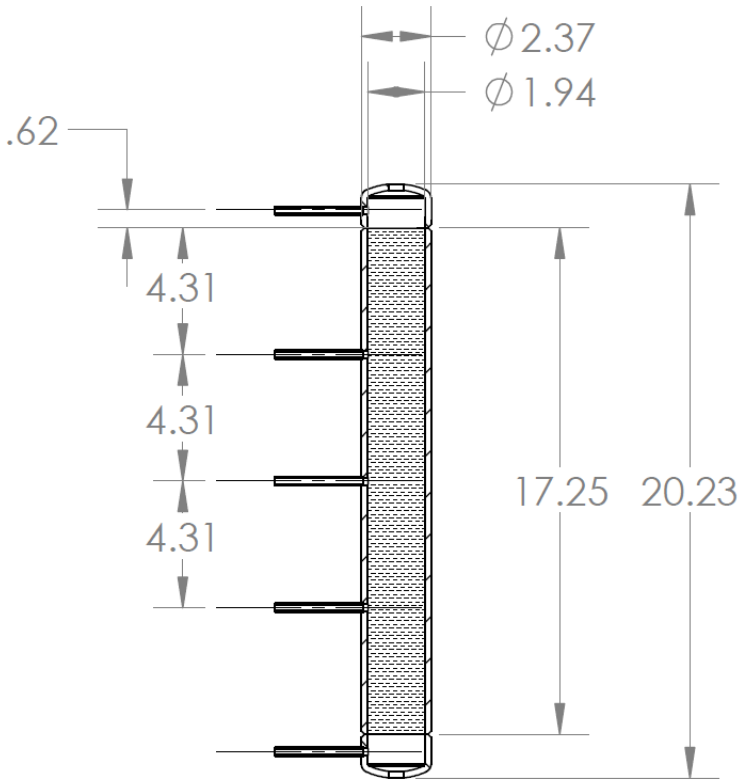


Figure 2.4 - Regenerator with dimensions (in inches). Dashed area represents packed bed of spheres

2.3.2 Binary Valves

The binary valves used for this experiment are HIP[6] air operated diaphragm valves. The valve body is 316SS with Grafoil packing. They have a maximum operating pressure of 10,000 psi with 9/16" medium pressure[7] female connections. The actuation type is air to open and air to close controlled with a solenoid valve manifold. The actuation time of the valve is ≈0.4 seconds. An image of two of the low-temperature binary valves is shown in Figure 2.5.

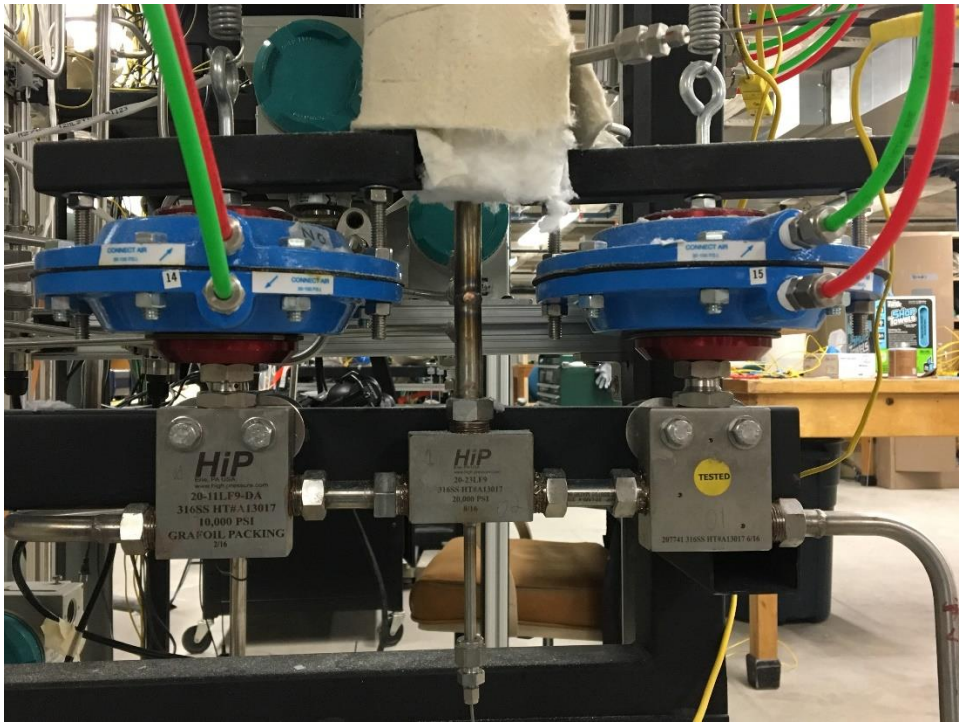


Figure 2.5 - Image of low temperature binary valves used for regenerator switching

[6] HIP-High Pressure Equipment Company
[7] medium pressure connection type is rated to 20,000 psi

There are 8 valves in total, and the arrangement as well as the open/close sequence is shown in Figure 2.6. In the figure, flow direction and temperature are indicated with colored arrows[8] and valve position is indicated with either a green (open) or red (closed) colored block.

During State 1, Regenerator 1 is experiencing the HTCB (hot to cold blow) and Regenerator 2 is experiencing the CTHB (cold to hot blow). That is, the hot fluid is transferring heat to the matrix during the HTCB and the matrix is transferring heat to the cold fluid during the CTHB. In State 2a, all the valves close; this is approximately 0.6 s. Then in State 2b, two valves open to allow the high-pressure regenerator to depressurize, and the low pressure regenerator to pressurize this is also approximately 0.6 s. During State 3, each regenerator experiences the opposite process it underwent in State 1.

The original design for the connection between the valves and the regenerators called for an HIP fitting to be used between the regenerator and all valve connections. This is shown in greater detail in Figure 2.7 and is also shown Figure 2.5.

[8] Red arrow=Hottest temperature, Blue arrow = cold (inlet temperature), Orange arrow = intermediate temperature

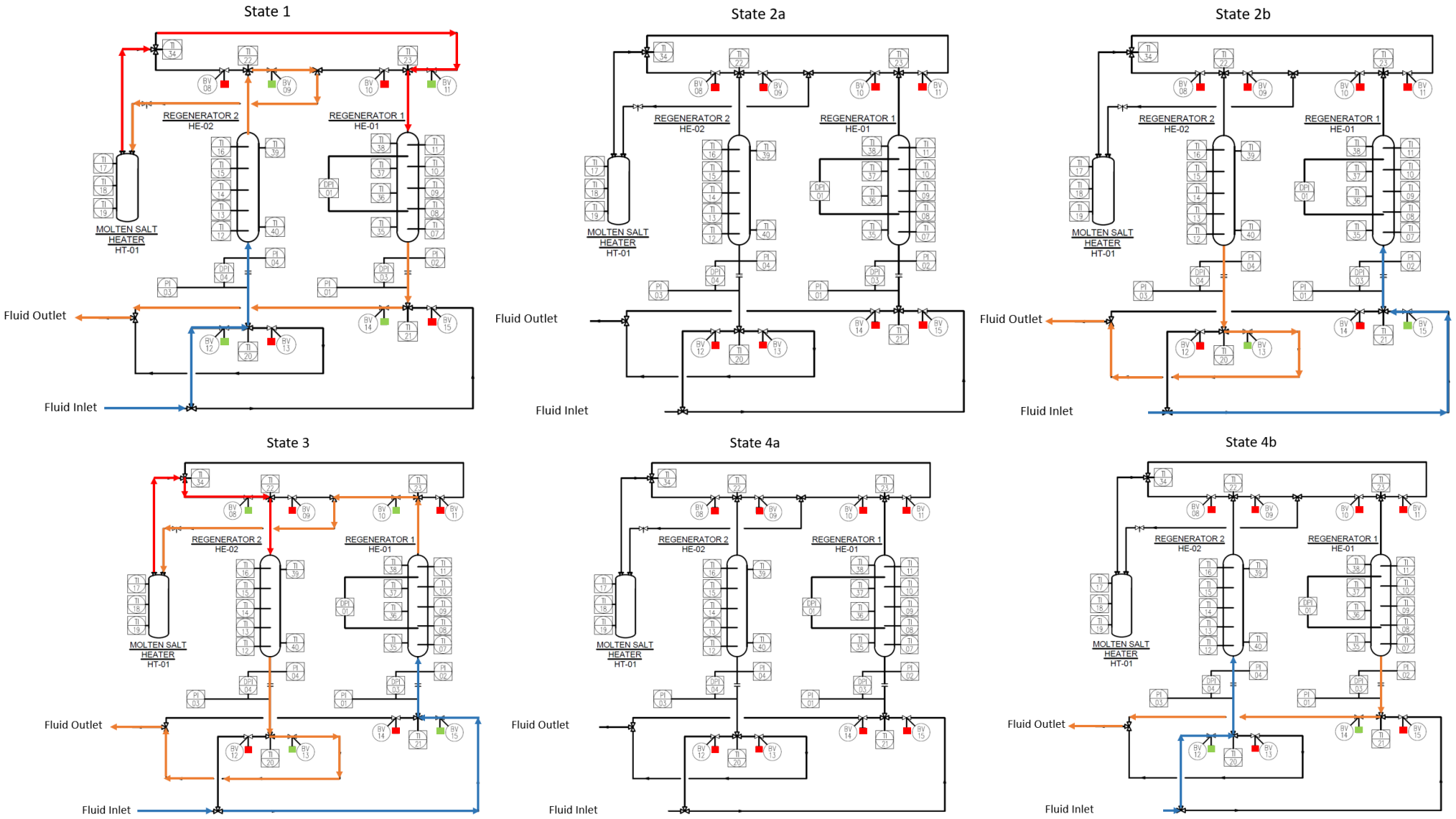


Figure 2.6-Valve Switching Sequence

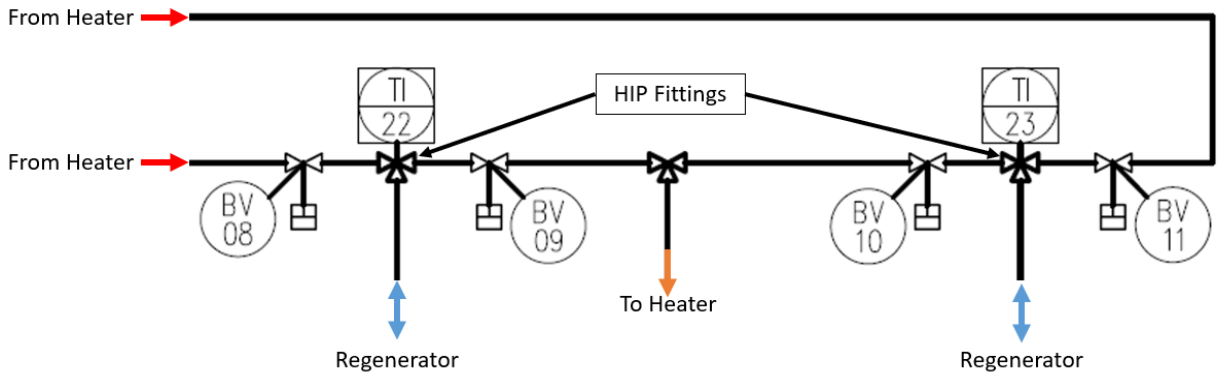


Figure 2.7 - Tee connection between Regenerators and Valves

The HIP fittings performed well for the first several months of initial testing. However, when high temperature-high pressure tests (500 C and 15 Mpa) began, a leak developed in the high temperature fittings. During the high temperature experiments, the temperature leaving the regenerator can vary up to 250 °C over 10-20 seconds. This large temperature transient seems to be creating a temperature difference between the tubing and the fitting, which is shown in Figure 2.8

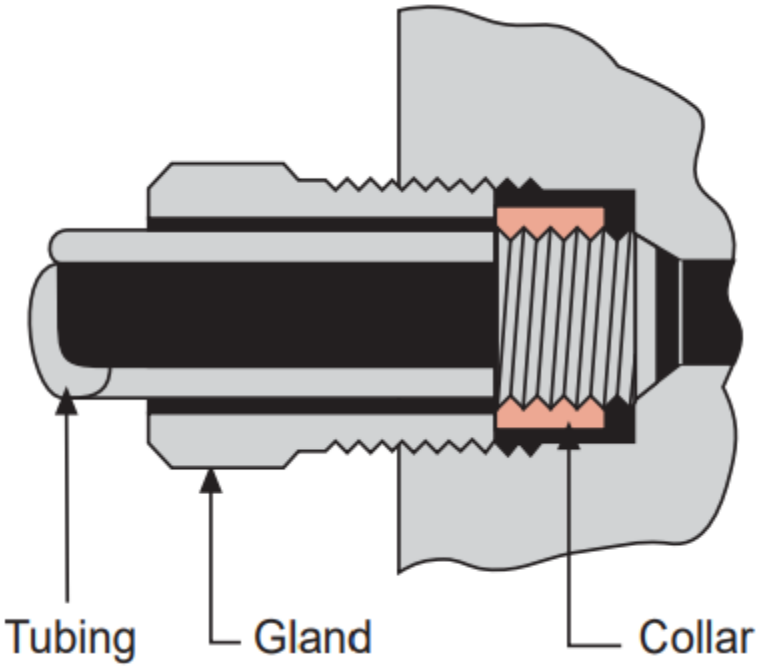
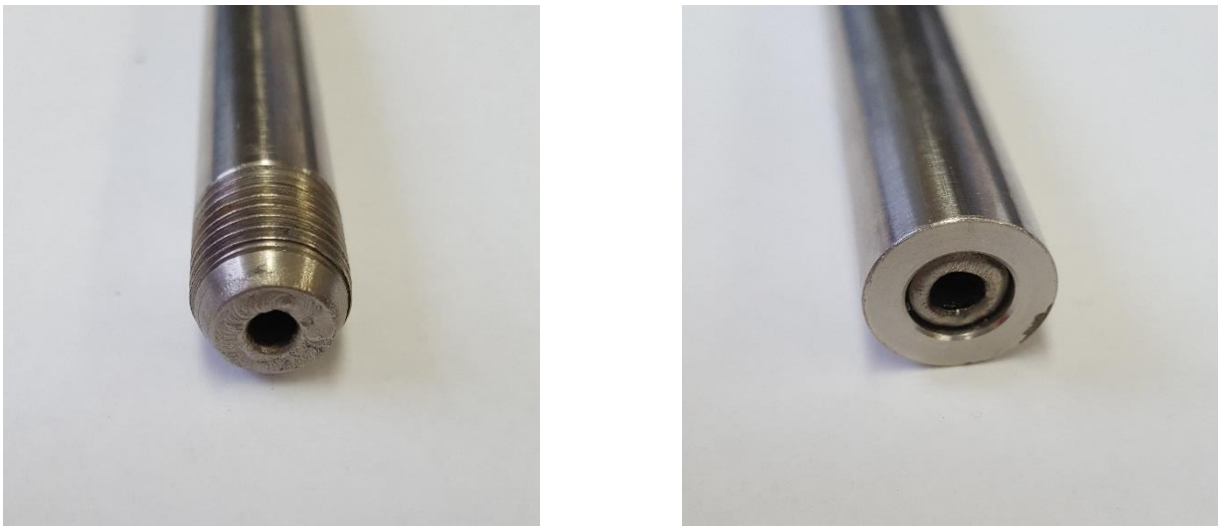


Figure 2.8 - Coned and threaded fittings

The tubing has a much smaller mass compared with the fitting and there is also significantly more surface area exposed to the fluid in the tubing than the fitting. Thus, the tubing will respond more quickly to a temperature change than the fitting. The leaks were observed when the fluid flowing through the fitting was initially at a high temperature and was cooling with time. This likely resulted in the tubing contracting faster than the fitting, causing the fitting to leak. After the fitting had adequate time to equilibrate in temperature with the fluid, the fitting contracted and the seal was re-established. To try and isolate the tubing and fittings from these large temperature fluctuations, the tubing was sleeved with ¼" x 0.035" tubing resulting in an annular gap of 1/32". An image of a sleeved tube is shown in Figure 2.9.



(a) (b)

Figure 2.9 - Sleeved HIP tubing (a) welded end (b) open end

The annular gap should insulate the outer tube from the temperature changes of the fluid, and intial calculations show the resistance to heat transfer from the fluid to the wall should increase by approximately 30x. The system was tested at maximum temperature and pressure and the sleeved fittings did not fail. This design seems to perform well, and will be recommended for use in future, higher power test facilities.

2.3.3 Compressor

A Hydro-Pac LX compressor was used to circulate CO2 for the experiment. It is a double ended hydraulically driven intensifier single stage type compressor. An image of the compressor is shown in Figure 2.10.

Figure 2.10 - Compressor used for regenerator experiments

Flow control for the experiment was accomplished using a bypass valve which could be manipulated to achieve the desired flow rate through the experiment. The maximum mass flow rate achievable is 1.6 kg/s.

2.3.4 Molten Salt Heater

A molten salt heater was used as the primary heat source for this experiment. The design of the heater is a tank of molten salt with submerged spiral tubing for the fluid to flow. An image of the tank and spiral tubing is shown in Figure 2.11. The large thermal mass of the molten salt helped to dampen out effects of unsteady flow due to valve switching. The maximum power of the salt heater is approximately 12kW provided by two band heaters wrapped around the outside of the tank.

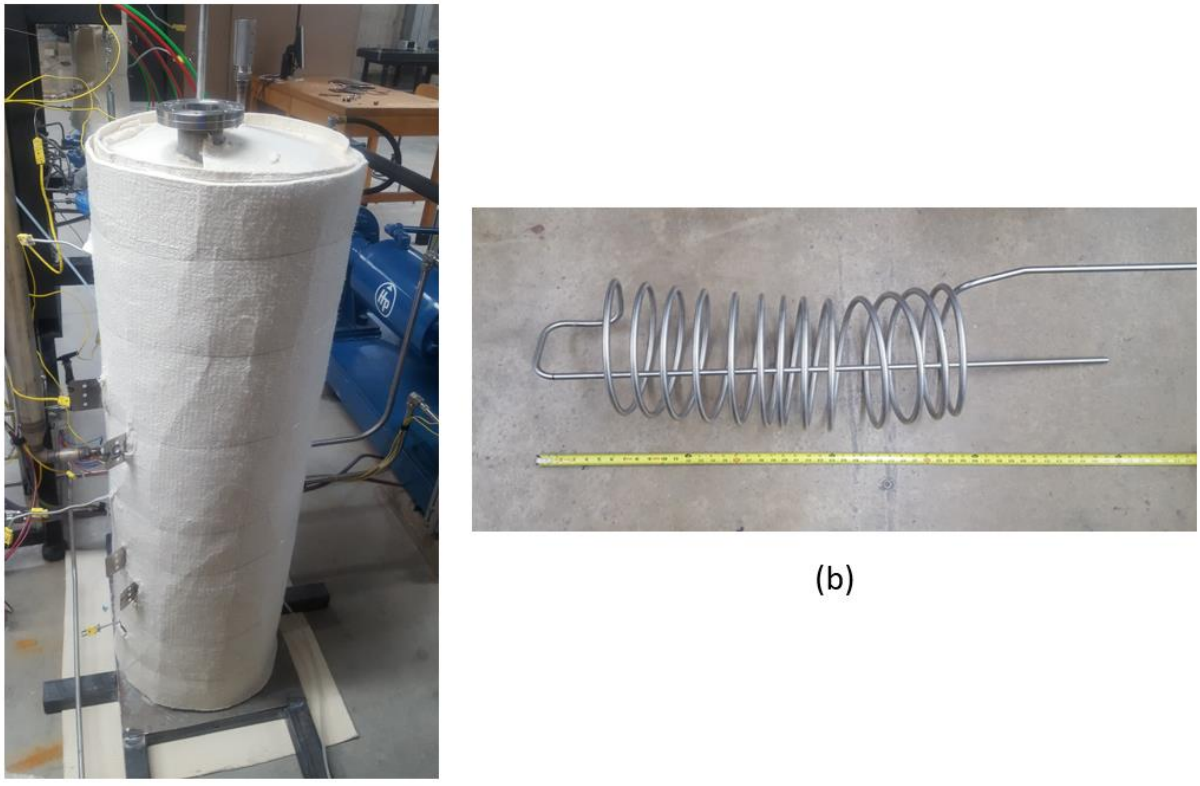


Figure 2.11 - Image of (a) molten salt tank and (b) spiral tubing

3 DATA ANALYSIS

3.1 DATA STORAGE AND ACCESS

The recorded data were exported from LabView in three tab separated text files corresponding to temperature, pressure/flow, and valve position. The files include the time of data capture and the value of the measured variable(s). A small section of a temperature data file is shown in Figure 3.1.

Year	Month	Day	Hour	Min	Sec	ms	TI01	TI02	TI03
2017	4	26	10	24	44	0.1261	-0.78662	46.05057	49.9224
2017	4	26	10	24	44	0.6264	0.710408	45.18439	49.06635
2017	4	26	10	24	45	0.1267	1.34425	44.54455	49.7091
2017	4	26	10	24	45	0.6264	2.21088	44.0454	51.27842

Figure 3.1 – Section of Temperature Data File

The tab separated files were read into a Python program and stored as a Pandas DataFrame object. The DataFrame object is a 2-dimensional data structure that stores data along with index (row labels) and column (column labels) arguments. An example of a DataFrame constructed from the first 4 rows and first 3 columns of the temperature data shown in Figure 3.1 is shown in Figure 3.2.

```
                                TI01       TI02       TI03
2017-04-26 10:24:44.126099 -0.786622  46.050568  49.922403
2017-04-26 10:24:44.626400  0.710408  45.184385  49.066351
2017-04-26 10:24:45.126700  1.344250  44.544554  49.709098
2017-04-26 10:24:45.626400  2.210880  44.045398  51.278415
```

Figure 3.2 - Example of DataFrame object structure

The data can then be selected based on the index (date and time-left most column) or the column name (TI01,TI02,or TI03). The DataFrame object can also be read and written from various file formats including CSV, text, Excel, and HDF5. This enables the DataFrame to be constructed one time from the raw data, and then loaded from a saved DataFrame at any time in the future for a variety of post-processing purposes. This significantly reduces computational time, especially when mathematical operations must be performed on the data.

To calculate thermodynamic properties of the fluid from the experimental data, a single temperature and pressure must be used for each time of interest. In order to make the temperature and pressure data sets the same length (pressure is sampled twice as fast as temperature), the pressure data is averaged. This is accomplished by selecting the first temperature point index and averaging all of the pressure data points with an index that is less than or equal to the temperature point index and greater than the previous temperature index. The averaged pressure is assigned the same index as the temperature point, and this procedure is repeated for all temperature indexes. The function used for this process is shown in Appendix F – Create_averaged_vars.

3.2 Thermodynamic Properties

The Fluid Property Interpolation Tables (FIT) program developed for the Solar Energy Lab by Northland Numerics was used to calculate the thermodynamic properties of carbon dioxide from the experimental temperature and pressure. FIT uses a piecewise interpolation of Helmholtz free energy and all other thermodynamic properties are derived from its analytical derivatives. To facilitate calculation of properties within the Python program used for data

reduction, the FIT Fortran modules were 'wrapped' using Python packages 'f2py' and 'f90wrap'. These Python packages allow Python extension modules to be built which call Fortran modules and subroutines from Python. The development of the Python/Fortran modules are described further in Appendix B – CO2 Properties.

3.3 Cycle Analysis

In order to analyze the experimental data on a 'per-cycle' basis, the data was divided into various "states" based on the valve switching times. Each state corresponds to a unique condition of the valves. An example of this is shown in Figure 3.3.

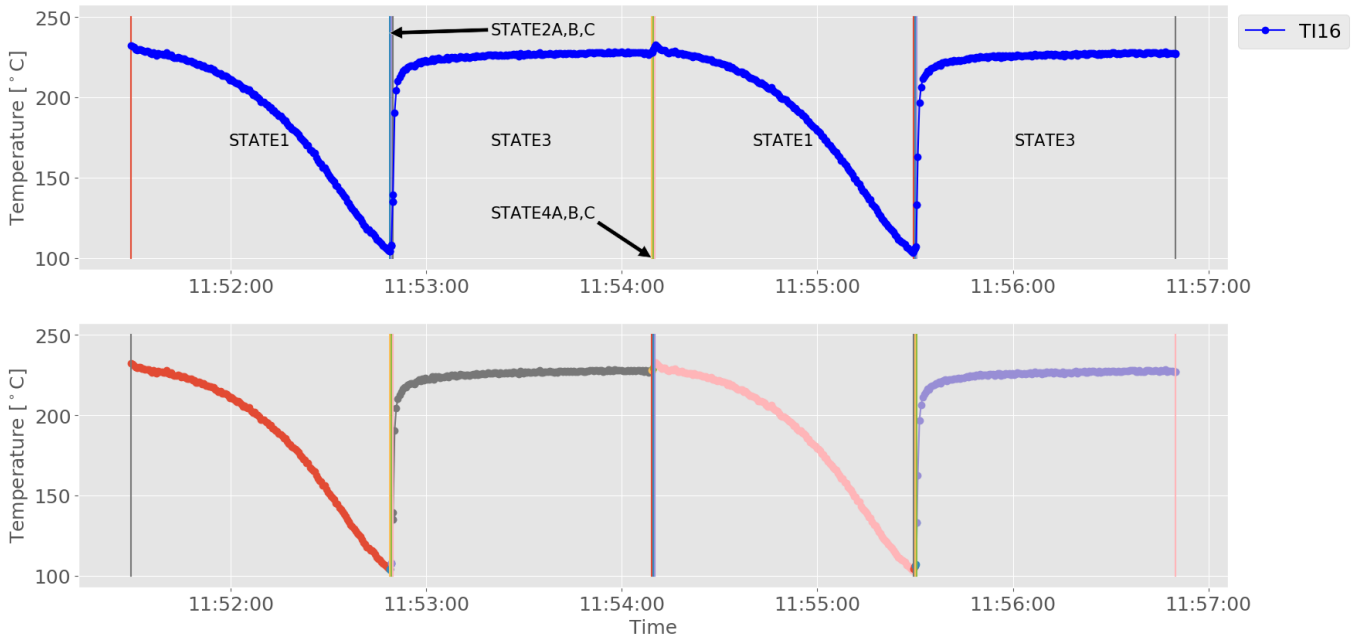


Figure 3.3 - Example of experimental data (top) split into various cycles (bottom – indicated by color)

In order to ensure each "state" had data representing the entire switching time, points are added to the beginning and end of each state so that they lie exactly on the switching time;

these points are added using a linear interpolation routine named “lin_int_cycle2” and contained in Appendix G – Linear Interpolation.

In Figure 3.3, the solid vertical lines indicate the times when the valves switch position. In this example, states 1 and 3 have switching times of 79.4 seconds and all other switching times are 0.5 seconds. The top plot shows the data before they are split into the different states and the bottom plot shows the states in different colors. With the data split into the various states as shown in Figure 3.3, the states can be averaged over a steady state operating period to get cycle-averaged values. See Python function “fast_ave” in Appendix C – Fast Average for additional information.

3.4 Heat Transfer Calculations

The total heat transfer from the hot fluid during one cycle is represented by Q_{HTCB} and the total heat transfer to the cold fluid is represented by Q_{CTHB}. The number included in the subscript for the heat transfer indicates for which regenerator the heat transfer is calculated. Equations 1 and 2 describe the heat transfer calculations for regenerator 1 and Equations 3 and 4 describe heat transfer for regenerator 2.

Regenerator 1:

$$Q_{HTBC1} = \int_0^{t_{HTCB}} \dot{m}(h_{11} - h_{07})dt \tag{1}$$

$$Q_{CTHB1} = \int_{t_{HTCB}}^{t_{CTHB}+t_{HTCB}} \dot{m}(h_{11} - h_{07})dt \tag{2}$$

Regenerator 2:

$$Q_{CTHB2} = \int_0^{t_{HTCB}} \dot{m}(h_{16} - h_{12})dt \quad (3)$$

$$Q_{HTCB2} = \int_{t_{HTCB}}^{t_{CTHB}+t_{HTCB}} \dot{m}(h_{16} - h_{12})dt \quad (4)$$

Where *h* is the specific enthalpy of the fluid and the subscripts indicate what temperature was used to calculate the enthalpy. The locations of the thermocouples in the packed bed are depicted in Figure 3.4.

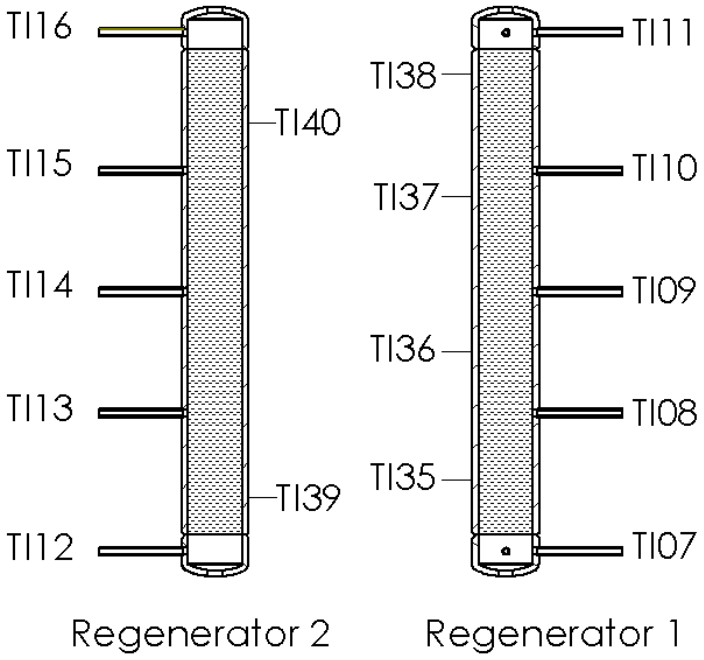


Figure 3.4 - Thermocouple locations for regenerators

The effectiveness of the regenerator is calculated using Equation 5:

$$\varepsilon = \frac{Q}{Q_{max}} = \frac{Q}{\dot{Q}_{max}(t_{HTCB}+t_{valves})} \quad (5)$$

Where *Q* is the experimental heat transfer calculated in either equation 1, 2, 3, or 4. Q_{max} is calculated by using the averaged properties from a steady state operating portion of the

experimental data to calculate the total heat transferred in an equivalent counterflow recuperator with an effectiveness of 1 (which corresponds to a maximum heat transfer rate multiplied by an operating time). The sub-heater exchanger approach as described in [1] is used to calculate the maximum heat transfer rate from prescribed temperature, pressure, and mass flow rate. In this approach the heat exchanger is divided into N segments that are each separately tasked with dealing with 1/Nth of the total duty. Each sub-heat exchanger is analyzed using the effectiveness-*NTU* technique. A diagram of this approach is shown in Figure 3.5.

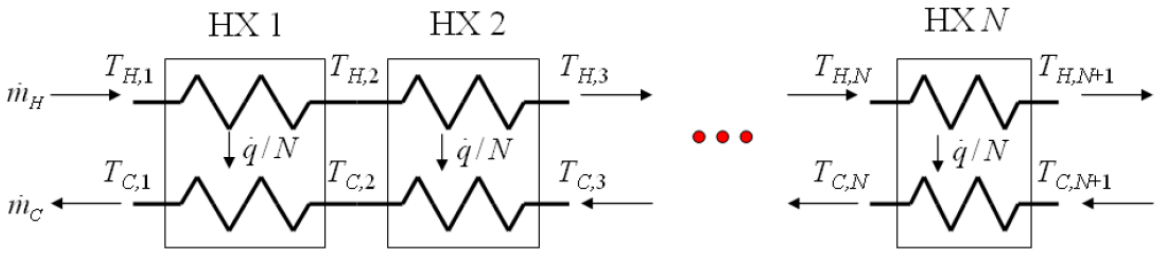


Figure 3.5 - Diagram of sub-heat exchanger model of a counter-flow heat exchanger.

The inlet temperatures and pressures used in the counterflow heat exchanger model are calculated by finding their cyclic average value across a steady state portion of the experimental data as described in Section 3.3. A 12 cycle average for TI11 and TI07 is shown in Figure 3.6[9].

[9] Data taken from 6-7-17 experiments

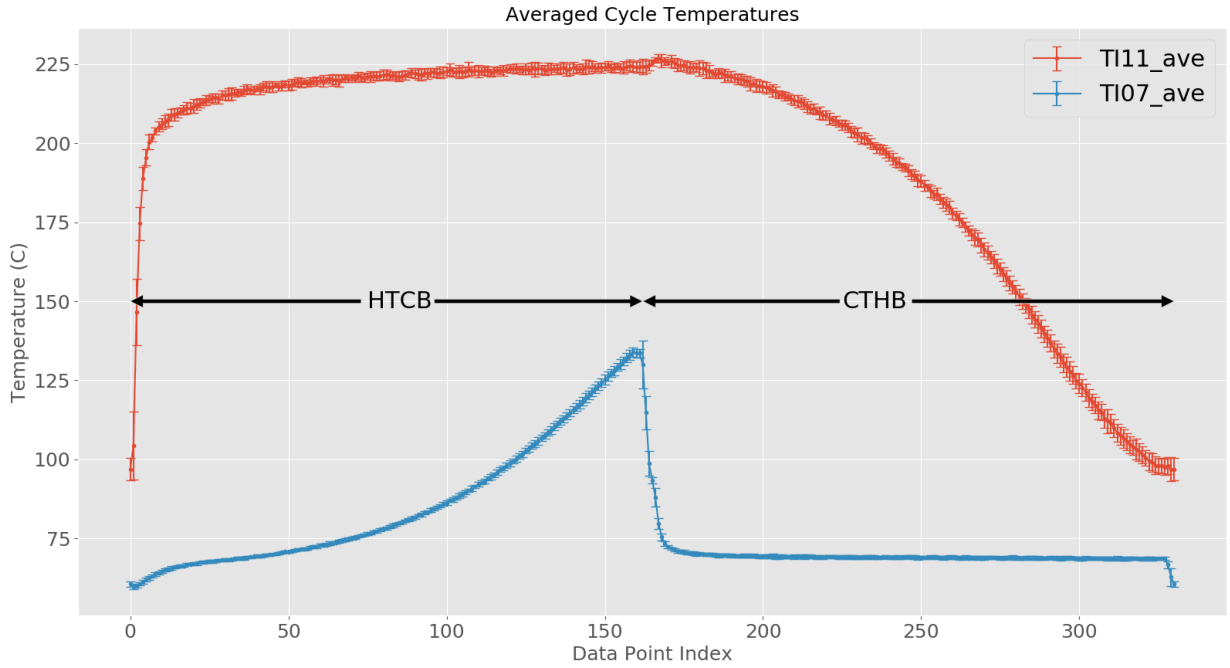


Figure 3.6 - Cyclic Average of TI16 with error bars

The average value of TI11 during the HTCB is used as hot inlet temperature to the recuperator and the average value of TI07 during the CTHB is used as the cold side inlet temperature. The same procedure is done for the cold and hot side pressure. To calculate an averaged mass flow rate that is comparable to a flow rate in an equivalent recuperator, Equation 6 was used.

$$\bar{\dot{m}} = \frac{1}{t_{HTCB}+t_{valves}} \int_0^{t_{HTCB}} \dot{m}\, dt \qquad (6)$$

Where t_{HTCB} is the HTCB switching time and t_{valves} is the time for states 2a and 2b to complete (typically 1.2 seconds). Note that t_{valves} is included in the denominator of Equation 6 even though the integration time is only t_{HTCB} in order to account for the valve switching time, during which the mass flow rate is zero. In this way, the averaged mass flow rate is more similar to what an equivalent recuperator would experience.

This process results in all of the inputs needed for the counterflow heat exchanger program:

$$T_{C,in}\ , T_{H,in}\ , P_{C,in}\ , P_{H,in}\ , \dot{m}_{C,in}, \dot{m}_{H,in}, \varepsilon$$

These inputs are supplied to a Fortran function developed by Northland Numerics that calculates the performance of a counter-flow heat exchanger. The function is described in more detail in Appendix E – FIT_HXR. The counterflow heat exchanger program returns a maximum heat transfer rate, $\dot{Q}_{max}$ and the effectiveness can then be calculated using Equation 5.

To get an average value for heat transfer and effectiveness for the steady state portion of the experiment, the procedure described above was done for each HTCB and CTHB in the steady state region. For example, the experimental temperature data from 10-4-17 is shown in Figure 3.7 with a steady state portion highlighted in red.

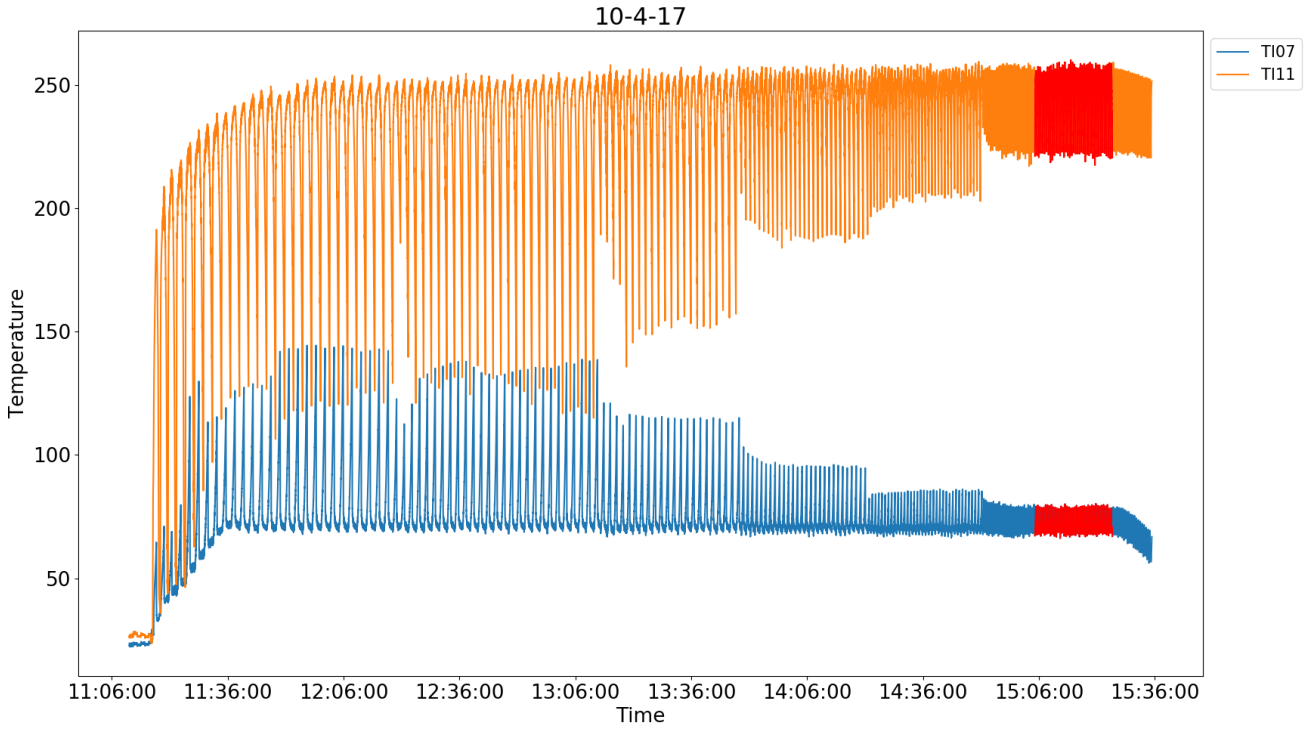


Figure 3.7 - Experimental Temperature Data from 10-4-17. Red Highlighted section is a steady state portion.

The highlighted portion from Figure 3.7 is shown in greater detail in Figure 3.8.

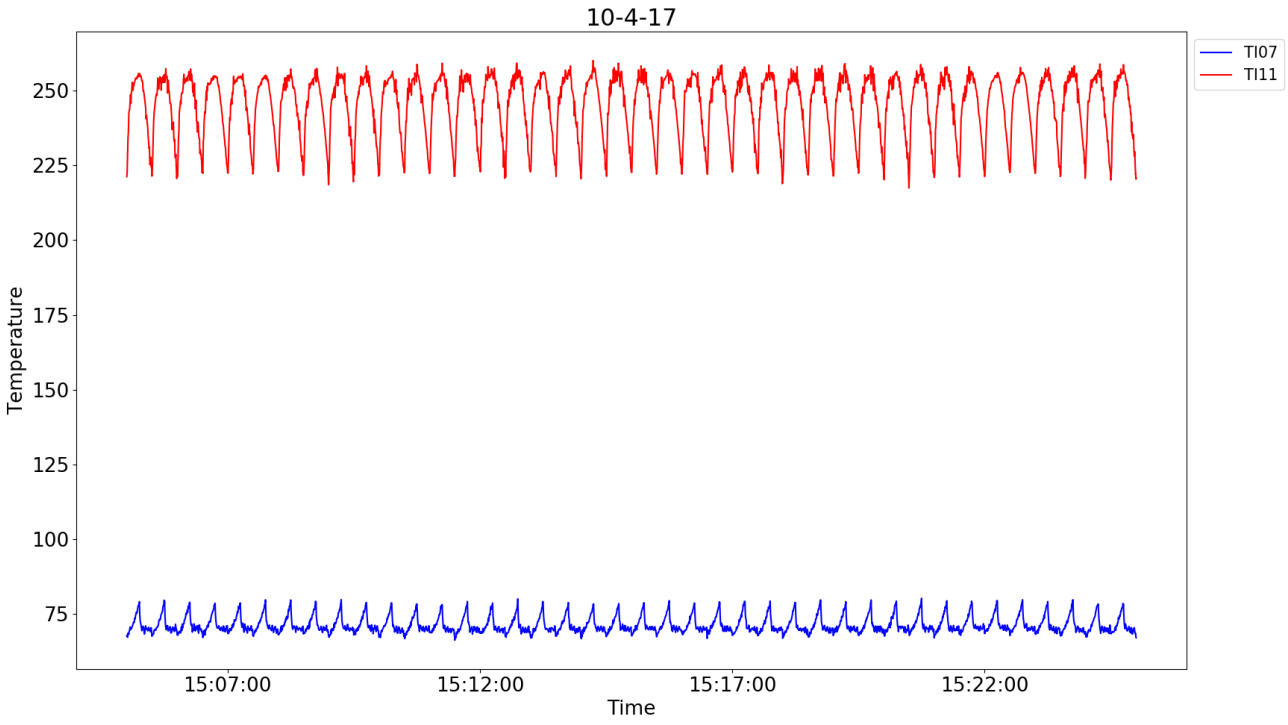


Figure 3.8 - A steady state portion of experimental data from 10-4-17

Each peak in Figure 3.8 represents a switch from HTCB to CTHB for Regenerator 1, which is shown in greater detail in Figure 3.6. Heat transfer, Q, maximum heat transfer, Q_{max}, and thus effectiveness can be calculated for each HTCB and CTHB. The average and standard deviation of these values can then be calculated and these are the results presented in Section 4.

3.5 Pressure Drop

Erdim et al. [2] collected 38 pressure drop correlations for flow through packed beds of spheres and from literature and developed a uniform notation to allow for comparison among the various correlations. All correlations are expressed in terms of the friction factor 'f', which is defined by:

$$f = \frac{1}{3}\frac{-\Delta P d_p}{\rho v_s^2 L}\frac{\phi^3}{(1-\phi)} \tag{7}$$

and is derived based on normalization by the kinetic force due to the flow of fluid on the solids of the packed bed.

The porosity of a packed bed is defined as the ratio of void volume to combined particle and void volume:

$$\phi = \frac{Vol_{void}}{Vol_{total}} \tag{8}$$

And v_s is the superficial velocity based on the empty cross-section of the bed:

$$v_s = \frac{\dot{m}}{\rho_f A_{cs}} \tag{9}$$

For ease of comparison among the various correlations, Erdim et al. adopted an alternative definition referred to as the particle friction factor:

$$f_p = \frac{\Delta P \, {d_p}^2}{\rho v_s^2 L} \tag{10}$$

Perhaps the most widely used correlation found in literature for pressure drop through a packed bed of spheres was proposed by Ergun and has the form:

$$f_p = \left(c_1 + c_2\left(\frac{Re}{(1-\varepsilon)}\right)\right)\frac{(1-\varepsilon)^2}{\varepsilon^3 Re} \tag{11}$$

While the Ergun correlation is the most widely used correlation, it has been shown to over predict pressure drop for flows above Re_m= 500 ([3], [2]) where Re_m is the modified Reynolds number, defined as:

$$Re_m = \frac{Re}{(1-\phi)} \tag{12}$$

with

$$Re = \frac{d_p \rho V}{\mu} \tag{13}$$

where d_p is the particle diameter, ρ is the fluid density, V is the fluid velocity, μ is kinematic viscosity, and φ is porosity [2]. The modified Reynolds numbers for the tests conducted here are $2700 < Re_m < 5300$, thus a correlation with better agreement in the range of modified Reynolds number of interest is needed. The Fahien and Schriver correlation has shown good agreement with experimental data and was chosen to calculate the pressure drop. The friction factor definition for the Fahien and Schriver correlation is given by [2] :

$$f_p = \left(q \frac{f_{1L}}{Re_m} + (1-q)(f_2 + \frac{f_{1T}}{Re_m})\right) \frac{(1-\phi)}{\phi^3} \tag{14}$$

$$q = \exp\left(-\frac{\phi^2(1-\phi)}{12.6} Re_m\right) \tag{15}$$

$$f_{1L} = \frac{136}{(1-\phi)^{0.38}} \tag{16}$$

$$f_{1T} = \frac{29}{(1-\phi)^{1.45}\phi^2} \tag{17}$$

$$f_2 = \frac{1.87\phi^{0.75}}{(1-\phi)^{0.26}} \tag{18}$$

The pressure drop is then calculated by:

$$\Delta P = \frac{f_p \rho V^2 L}{d_p} \tag{19}$$

A correction to the measured differential pressure across the bed is needed to account for the static pressure of the fluid in the tubing to the pressure sensor. A schematic of the differential pressure sensor is shown in Figure 3.9.

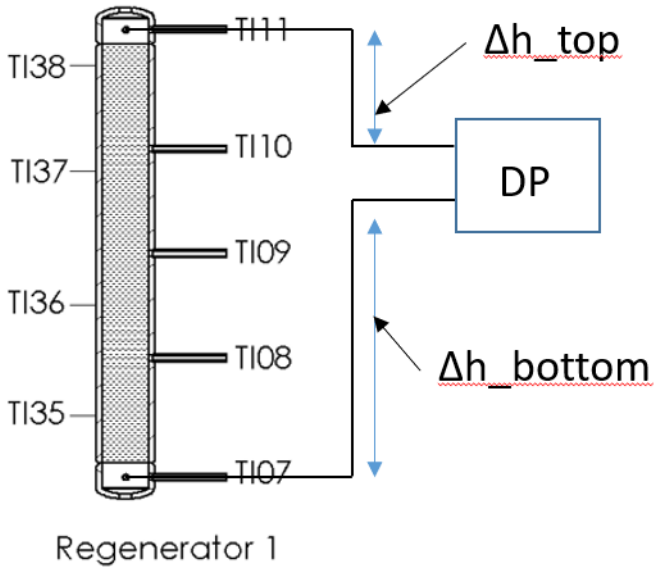


Figure 3.9 - Schematic of differential pressure sensor connections to regenerator

The differential pressure correction is calculated with:

$$\mathrm{d}P_{corrected} = \mathrm{d}P_{original} - \mathrm{d}P_{top} - \mathrm{d}P_{bottom} \tag{20}$$

where

$$\mathrm{d}P_{top} = \rho g \Delta h_{top} \tag{21}$$

$$\mathrm{d}P_{bottom} = \rho g \Delta h_{bottom} \tag{22}$$

The density was calculated using the pressure at the inlet of the bed and a temperature of 40 °C. This temperature was used to attempt to approximate the temperature of the stagnant fluid in the instrument tubing. In future experiments, a temperature measurement should be added at the tubing to calculate a more precise density.

3.6 HEAT TRANSFER COEFFICIENT

Presented here are some of the most commonly cited correlations for Nusselt number found in literature for flow through packed beds of spheres. Gnielinski introduced a correlation in which the Nusselt number for flow over a single sphere is adapted for flow through a packed bed by means of an empirical arrangement factor. The Nusselt number for flow around a sphere is given by a combination of the laminar and turbulent solutions [4]:

$$Nu_{sp} = 2 + (Nu_1^2 + Nu_t^2)^{0.5} \tag{23}$$

with

$$Nu_1 = 0.664 Pr^{1/3} \left(\frac{Re}{\varepsilon}\right)^{1/2} \tag{24}$$

And

$$Nu_t = \frac{0.037\left(\frac{Re}{\varepsilon}\right)^{0.8} Pr}{1+2.443\left(\frac{Re}{\varepsilon}\right)^{-0.1}\left(Pr^{2/3}-1\right)} \tag{25}$$

To apply this expression (equation 23) to a packed bed, an empirical arrangement factor is defined:

$$f(\varepsilon) = 1 + 1.5(1 - \varepsilon) \tag{26}$$

The overall Nusselt number is given by:

$$Nu_{Gnielinski} = f(\varepsilon)Nu_{sp} \tag{27}$$

Achenbach (1995), completed experiments using a bed with 0.983 m diameter and 0.84 m height filled with spheres of 0.06 m diameter and porosity 0.387 and compared his results with the Gnielinski correlation. The heat transfer experiments performed by Achenbach were carried out by using the method of an electrically heated single sphere in an unheated packing. His results matched the Gnielinksi correlation reasonably well for Re > 500 and he presented an empirical correlation (eq. 28) from his experiments.

$$Nu_{Achenbach} = \left[(1.18Re^{0.58})^4 + \left(0.23{Re_m}^{0.75}\right)^4\right]^{1/4} \tag{28}$$

Wakao and Kaguei collected a number of experimental results and compared them in [5]. The result of this comparison is given by equation 29:

$$Nu_{Wakao} = 2 + 1.1\ Pr^{1/3}\ Re^{0.6} \tag{29}$$

The heat transfer coefficient can be calculated from the Nusslet number using equation (30).

$$\bar{h} = \frac{Nu\ k_f}{d_p} \tag{30}$$

Where k_f is the conductivity of the fluid.

Kays and London presents a correlation based on the Colburn factor (j_h):

$$j_h = 0.23{Re_{KL}}^{-0.3} \tag{31}$$

With

$$Re_{KL} = \frac{4\ G\ r_{char}}{\mu_f} \tag{32}$$

Where $\mu_f$is the fluid viscosity, r_{char} is the characteristic radius:

$$r_{char} = \frac{\phi d_p}{6(1-\phi)} \tag{33}$$

And *G* is the mass flux:

$$G = \frac{\dot{m}}{\phi A_{fr}} \tag{34}$$

Where A_{fr} is the frontal area of the regenerator. The Colburn factor can then be related to the heat transfer coefficient with equation (35)

$$\bar{h} = \frac{j_h\ G\ c_f}{Pr^{\frac{2}{3}}} \tag{35}$$

Figure 3.10 shows the $\bar{h}$ vs Re using the various correlations with a Prandtl number of 0.71:

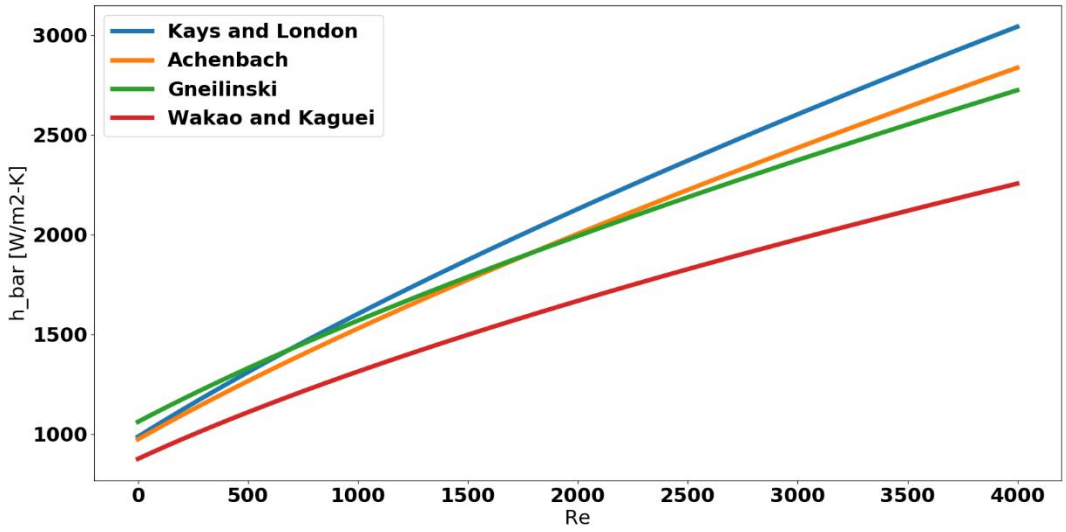


Figure 3.10 - heat transfer coefficient vs Reynolds number for various correlations, Pr=0.71

The range of Reynolds numbers for these experiments is typically between 1000-3500. The Kays and London correlation was chosen for this analysis, however the relatively large discrepancy among these correlations may be a source of uncertainty when calculating quantities such as NTU.

3.7 Regenerator Design

An important dimensionless parameter in the design of regenerators is the matrix capacity ratio:

$$C_m = \frac{m_b c_b}{P_0 \dot{C}_{min}} \tag{36}$$

where m_b is the mass of the matrix material, c_b is the specific heat capacity of the matrix material, P_0 is the HTCB and CTHB switching time, and $\dot{C}_{min}$ is the minimum capacitance rate of the fluid flowing through the bed. The capacitance rate of each fluid stream is defined by:

$$\dot{C} = c_{p,ave}\dot{m} \tag{37}$$

where $c_{p,ave}$ is the average specific heat of the fluid and $\dot{m}$ is the mass flow rate of the fluid.

3.8 CARRYOVER

One characteristic inherent to regenerator operation is carryover. Carryover is fluid that remains trapped in the regenerator matrix material after the valves are switched. This affects the overall efficiency of the Brayton cycle because some fluid that was pressurized by the compressor is returned to the compressor without expanding through the turbine.

Experimental carryover was calculated by using the five temperature measurements in the regenerator bed and the pressure at the inlet of the bed to calculate a density profile of the fluid in the regenerator. The total void volume of the regenerator was divided into five equal parts and multiplied by the density at each respective location to get a fluid mass. The masses were then summed to get the total fluid mass. To calculate carryover, the mass was calculated at two different times and then subtracted. Since there are two regenerators in the system, this difference is multiplied by two:

$$carry\ over = 2(m_{end\ of\ CTHB} - m_{start\ of\ HTCB}) \tag{38}$$

4 RESULTS

4.1 EFFECTIVENESS RESULTS

The measured effectiveness as a function of the matrix capacity ratio are shown in Figure 4.1 and Figure 4.2 based on the HTCB and CTHB, respectively.

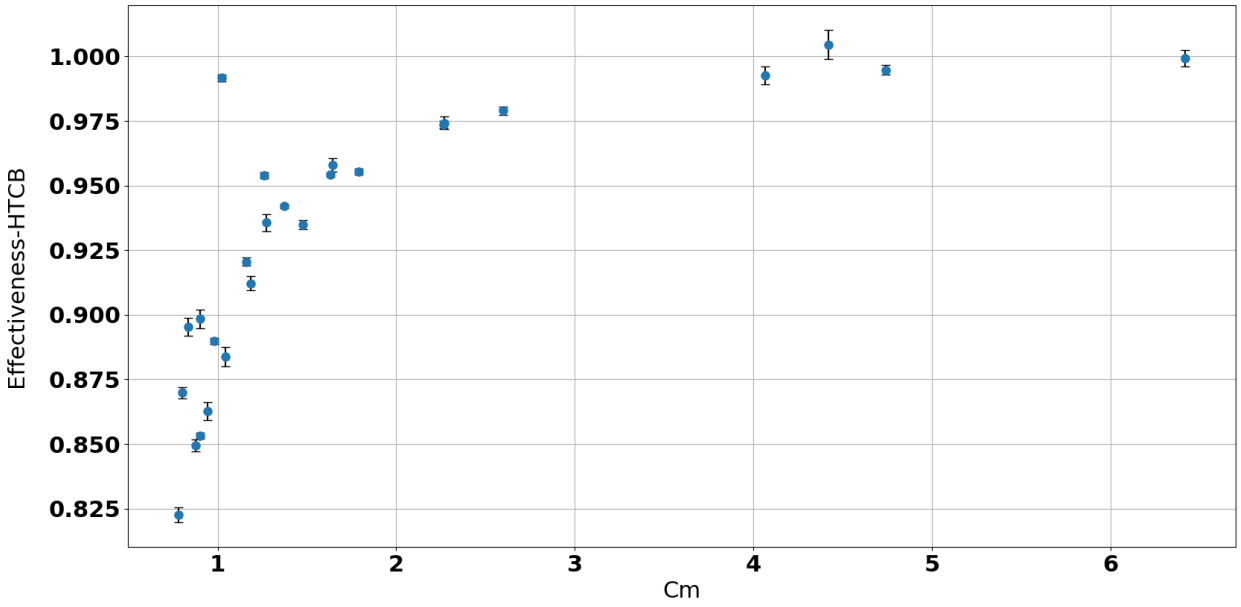


Figure 4.1 - Effectiveness vs Cm for the HTCB

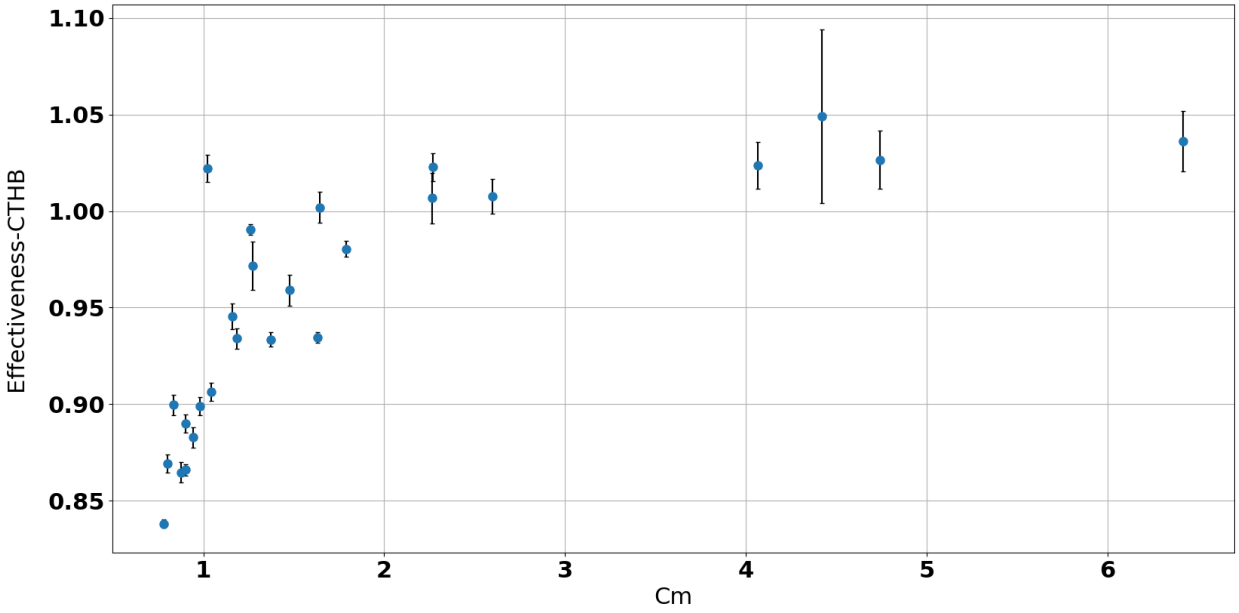


Figure 4.2 - Effectiveness vs Cm for the CTHB

Notice that there are some instances where the measured effectiveness is greater than one, which is not physically possible. One possible explanation could be the participation of the regenerator wall in the heat transfer process somehow confounding the measurement results. Figure 4.3 shows the temperature of the exterior regenerator wall as a function of time.

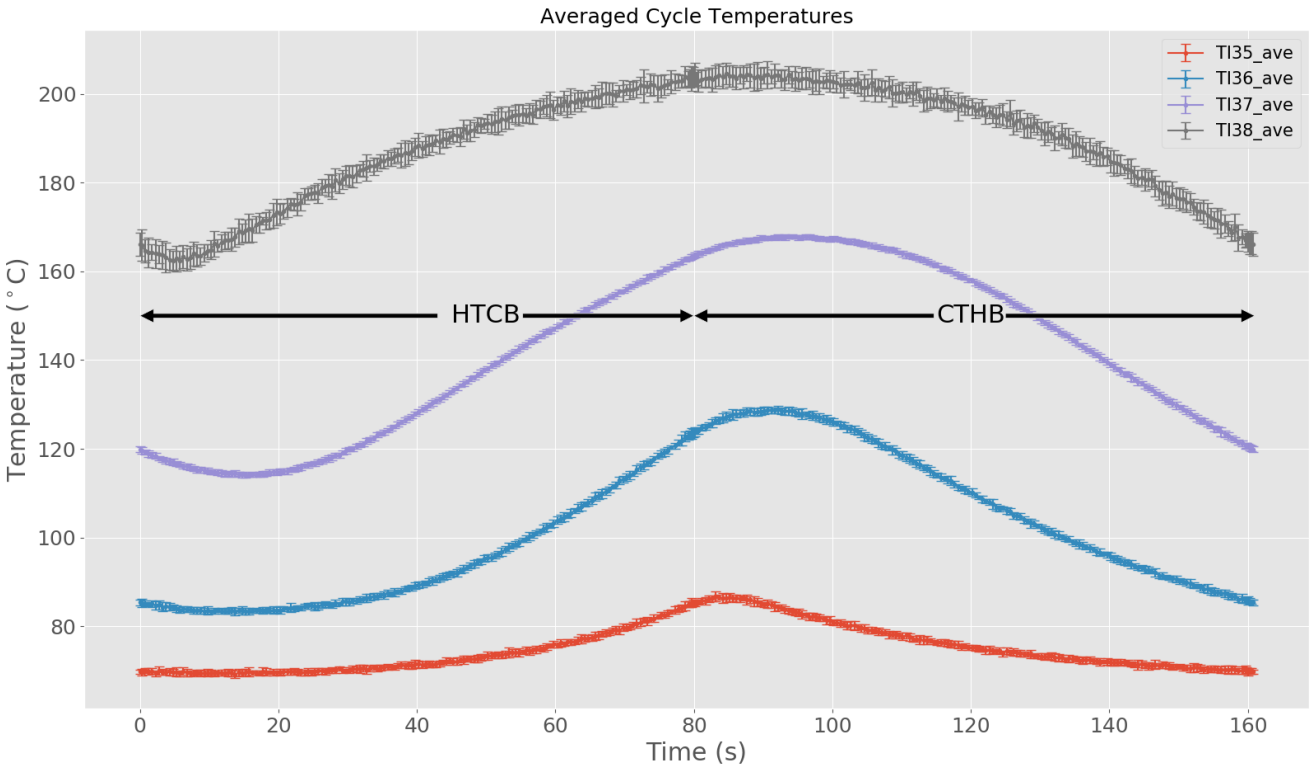


Figure 4.3 - Regenerator Wall Temperature

At the hot end of the regenerator, the temperature of the wall fluctuates by approximately 50°C during each cycle. If a lumped capacitance is used to model the wall material then the amount of energy being stored and released in each cycle can be approximated. For the temperature plot shown, this energy is about 100 kJ which corresponds to approximately 30% of the total energy being transferred to the fluid. As can be seen in Figure 4.3, the wall temperature is slightly out of phase with the flow in the bed, meaning that for some time the wall is extracting heat during the CTHB and adding heat during the HTCB. This makes correcting for wall participation difficult. In future experiments, the regenerator bed will be insulated so the ability of heat to enter the wall will be dramatically reduced. Additionally, as the size of the

regenerators increase for larger scale systems, the mass of the walls compared to the mass of the matrix will be much smaller, which will reduce this effect even more.

Another possible explanation could be a "shift" in the data due to the time constant of the thermocouple. A thermocouple can be modeled as a lumped capacitance which is then governed by the first-order ordinary differential equation:

$$\frac{dT}{dt} + \frac{T}{\tau} = \frac{T_f}{\tau} \tag{39}$$

where T is the temperature recorded by the thermocouple, T_f is actual the temperature of the fluid, and τ is the time constant. As a first attempt at correcting the data, the ODE was approximated numerically using the backwards difference technique:

$$\frac{T_i - T_{i-1}}{\Delta t} + \frac{Ti}{\tau} = \frac{T_{f,i}}{\tau} \tag{40}$$

and solving for T_f:

$$T_{f,i} = \tau\left(\frac{T_i - T_{i-1}}{\Delta t} + \frac{Ti}{\tau}\right) \tag{41}$$

The time constant τ is defined as the product of the thermal resistance and total heat capacity which therefore corresponds to the time required to reach 63.2% of an instantaneous (step) temperature change. This step temperature change approximately occurs at the switching points and therefore the time constant can be approximately evaluated by using the temperature data from cold end of the regenerator during the CTHB. The cold side temperature of regenerator 1 (TI07) during the CTHB is plotted in Figure 4.4 along with lines showing an approximate step change and an interpolated point at 63.2% of the approximate step change.

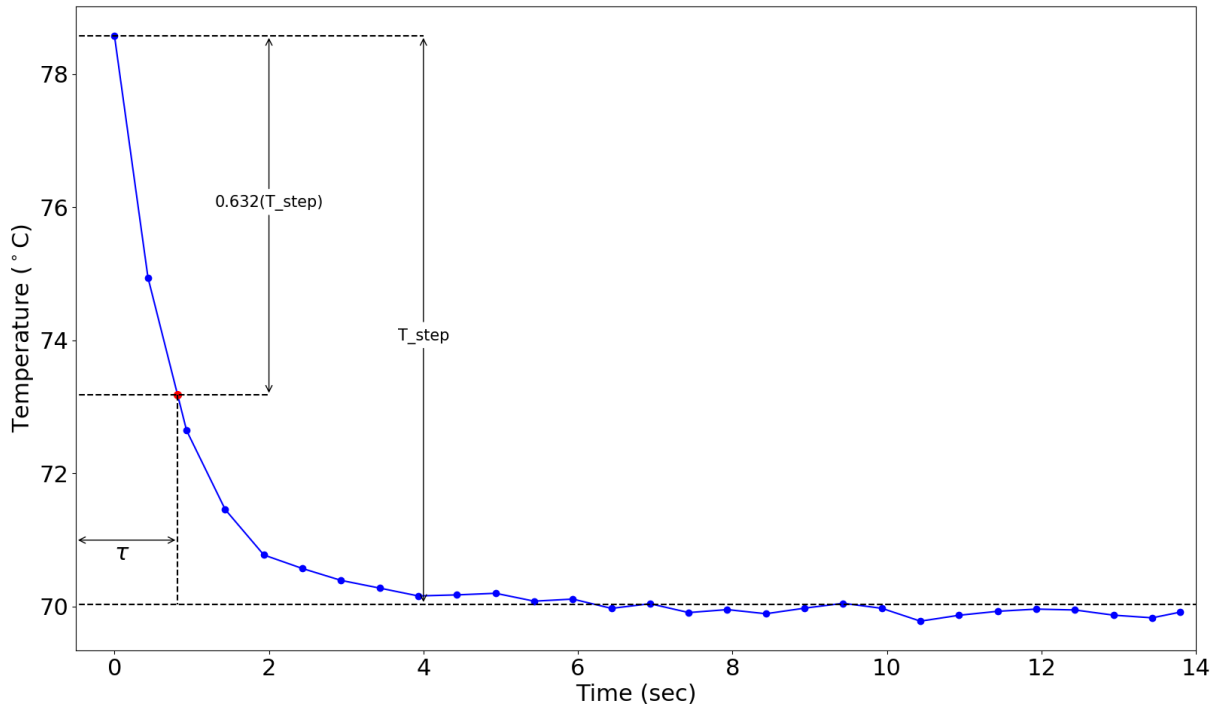


Figure 4.4 - Cold side temperature (TI07) of regenerator 1 during the CTHB. This can be used to approximate a time constant

The time constant shown in Figure 4.4 is approximately 0.82 seconds. Using Equation 41 the temperature data can be corrected to determine the actual fluid temperature from the thermocouple measurement, accounting for the time constant of the thermocouple. The heat transfer, maximum heat transfer, and effectiveness can then be calculated using these corrected fluid temperature values. A comparison between the original and corrected calculated effectiveness for the HTCB is shown in Figure 4.5 and the CTHB in Figure 4.6. Notice that the non-physical results (ε >1) are essentially eliminated.

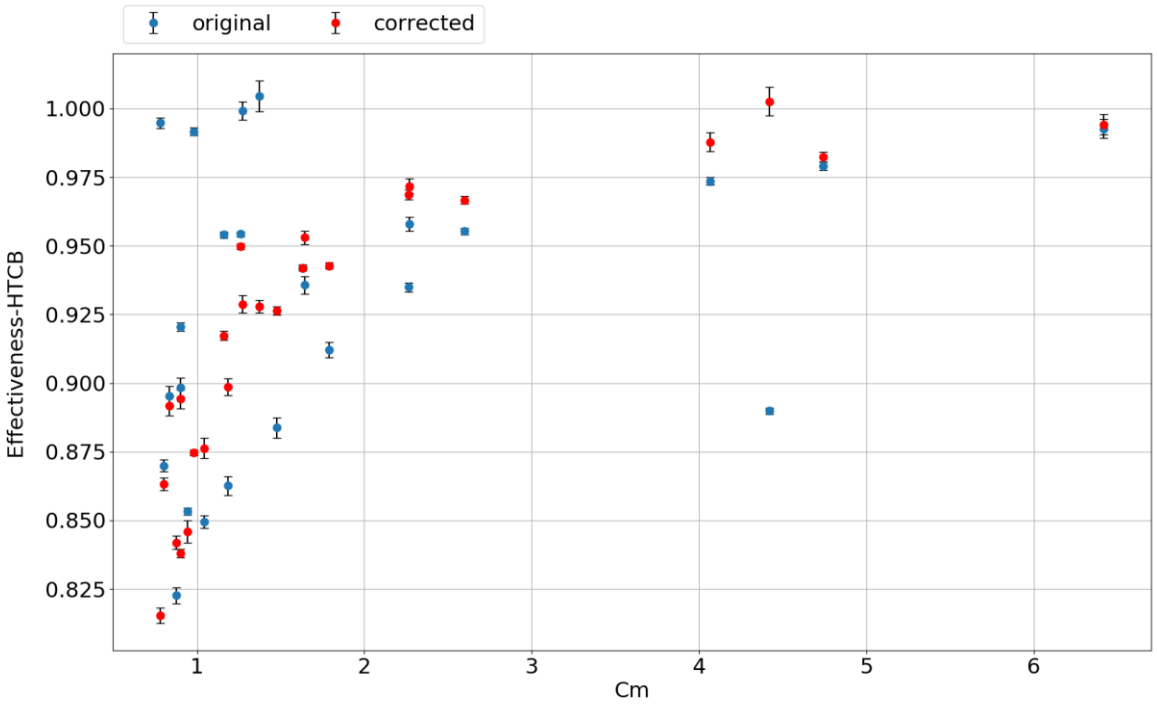


Figure 4.5 - Effectiveness vs Cm for HTCB original and corrected comparison

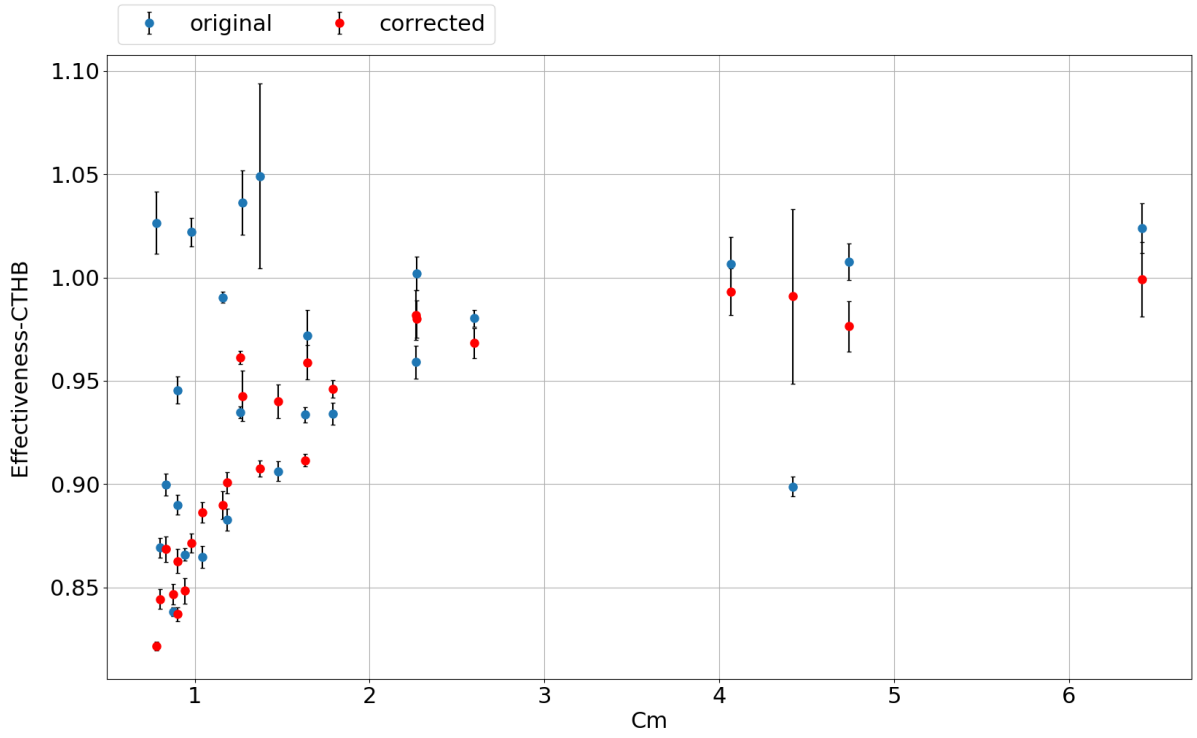


Figure 4.6 - Effectiveness vs Cm for CTHB original and corrected comparison

The NTU-Cm-Effectiveness model previously developed by colleagues at UW-Madison describes the performance of a regenerator system; the model is described in detail in [6]. Experimental information such as mass flow rate, average cycle temperatures and pressures, and switching time were fed into this model and values of effectiveness, pressure drop, and carryover were then compared with the experimental results (after correcting for the thermocouple dynamics). The effectiveness results for the CTHB are compared in Figure 4.7.

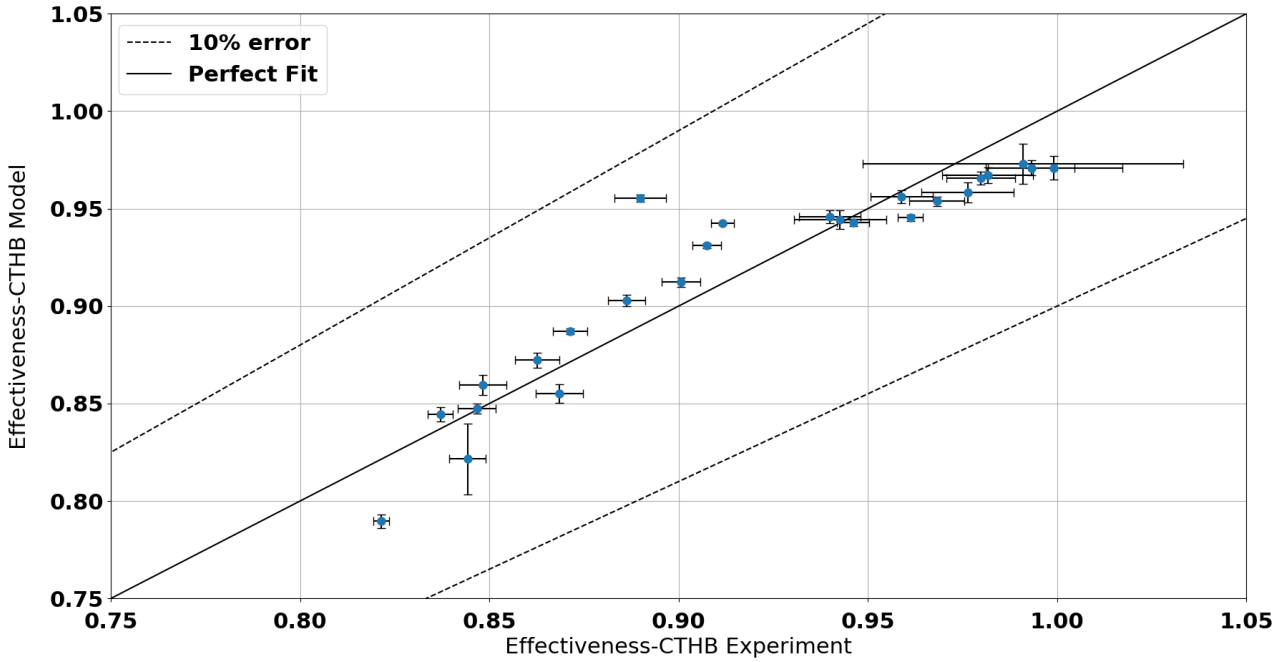


Figure 4.7 - Model vs experiment effectiveness

As shown in Figure 4.7 the model is generally in good agreement with the experimental results.

4.2 Pressure Drop Results

The experimental pressure drop through the regenerator is plotted against the predicted pressure drop in Figure 4.8 and Figure 4.9 for the HTCB and CTHB respectively. The Fahien and Schriver correlation is used for the 'model' value shown in these plots.

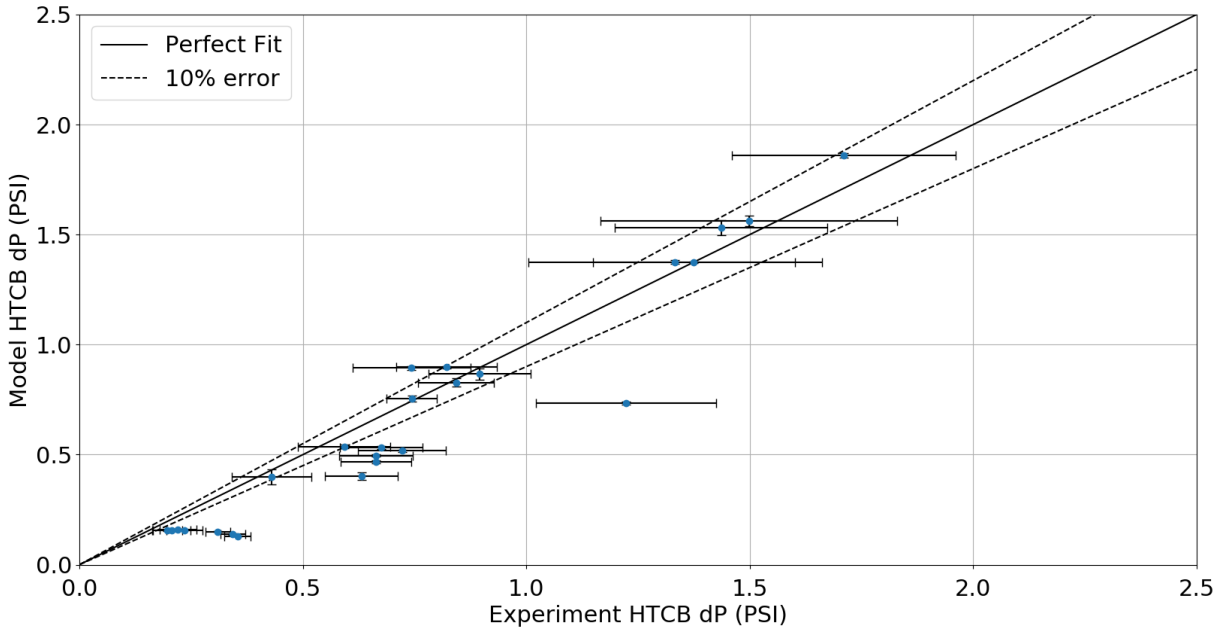


Figure 4.8 - Model vs Experimental Pressure drop through the regenerator for the HTCB

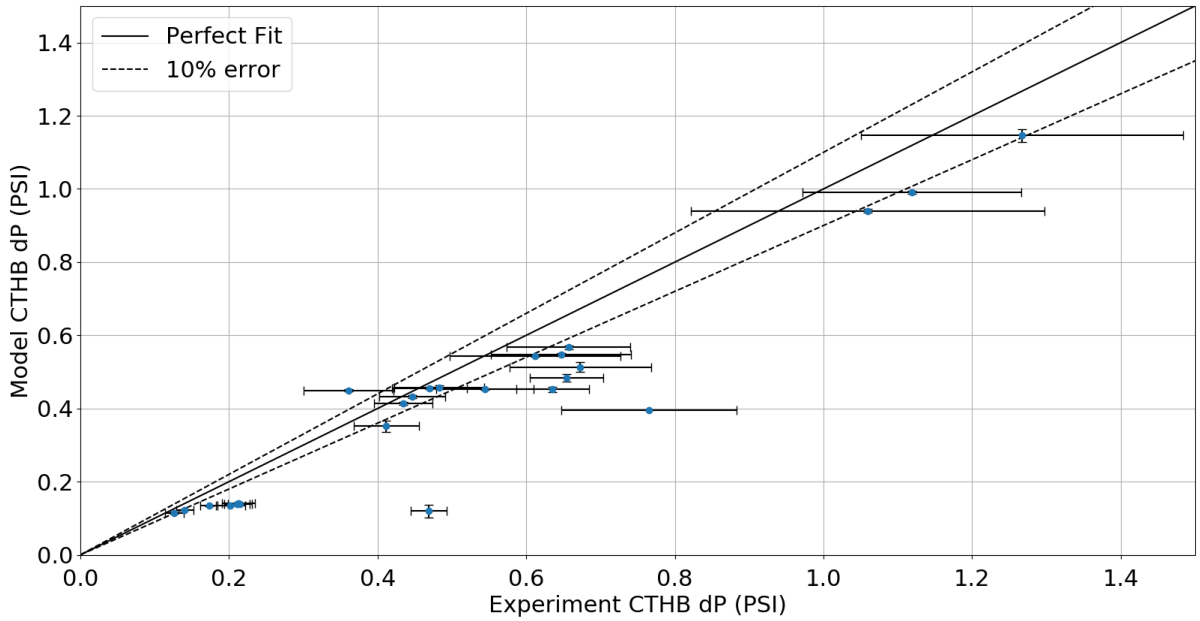


Figure 4.9 - Model vs experimental pressure drop through the regenerator for the CTHB

The pressure drop results show generally good agreement with the model predictions; however the pressure drop is frequently under-predicted, especially during the CTHB. The disagreement may be explained by the lack of an accurate temperature measurement for use in calculating the density used for the correction described in Equation 20. A simple sensitivity analysis was performed to determine to what extent the correction calculation (equation 20) is dependent on the assumed static fluid temperature in the instrument tubing. Three reasonable values of the temperature were assumed and the results are plotted in Figure 4.10 and Figure 4.11.

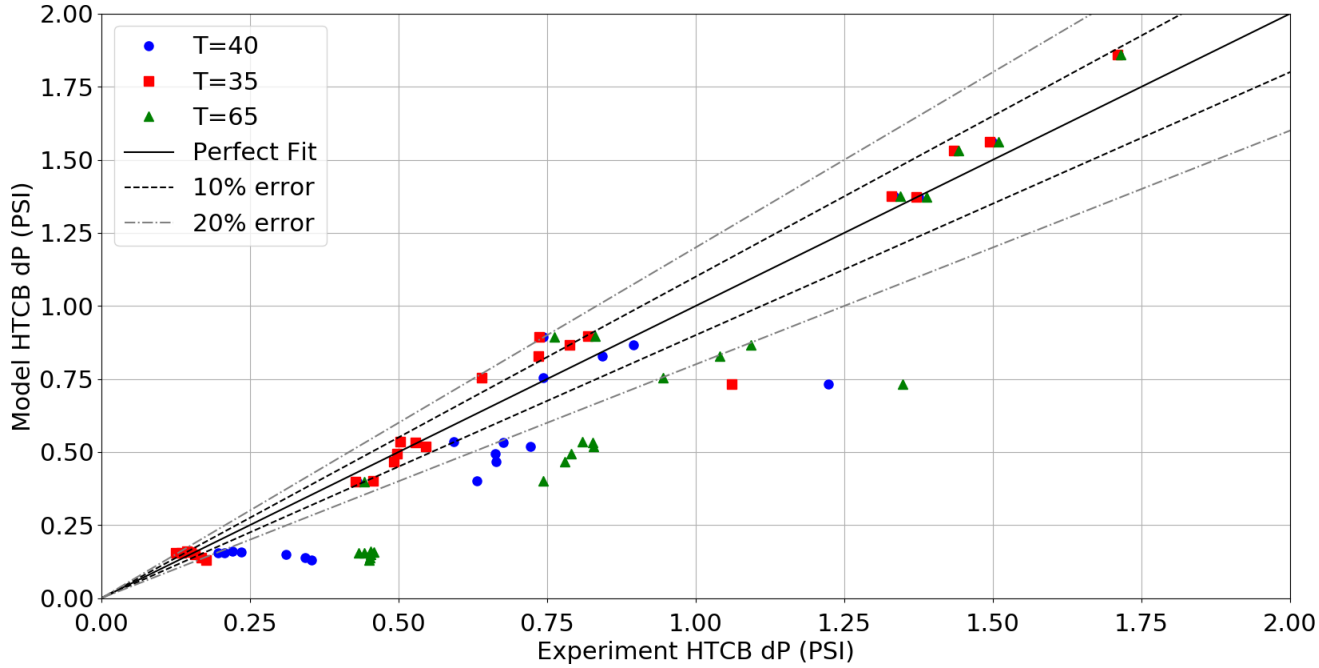


Figure 4.10 - HTCB pressure drop correction sensitivity to assumed static temperature (equation 20)

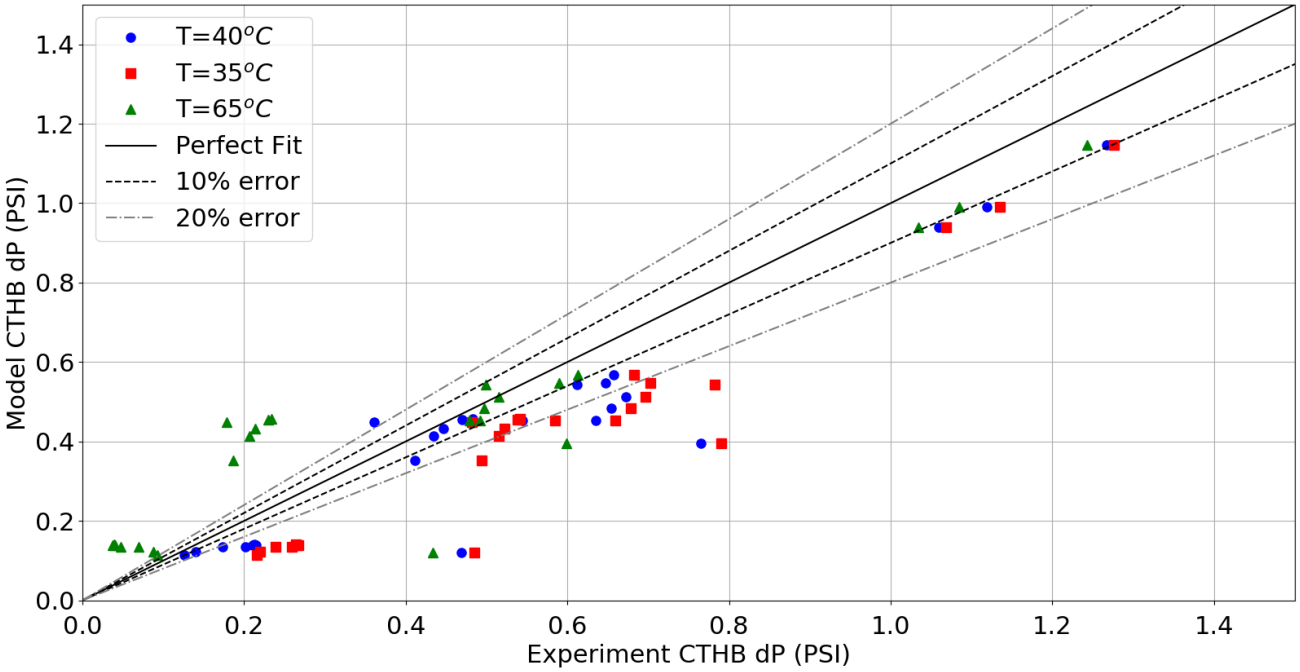


Figure 4.11 – CTHB pressure drop correction sensitivity to assumed static temperature (equation 20)

As can be seen in Figure 4.10 and Figure 4.11, the pressure drop is strongly dependent on the assumed temperature of the static fluid. Thus the assumption of a constant 40 °C in the instrumentation tubing is not a good approximation and a temperature measurement should be added to the experiment to improve the ability calculate a density of the static fluid.

4.3 CARRYOVER RESULTS

Figure 4.12 shows the carry over results versus a NTU-Cm-Effectiveness model as described in [6].

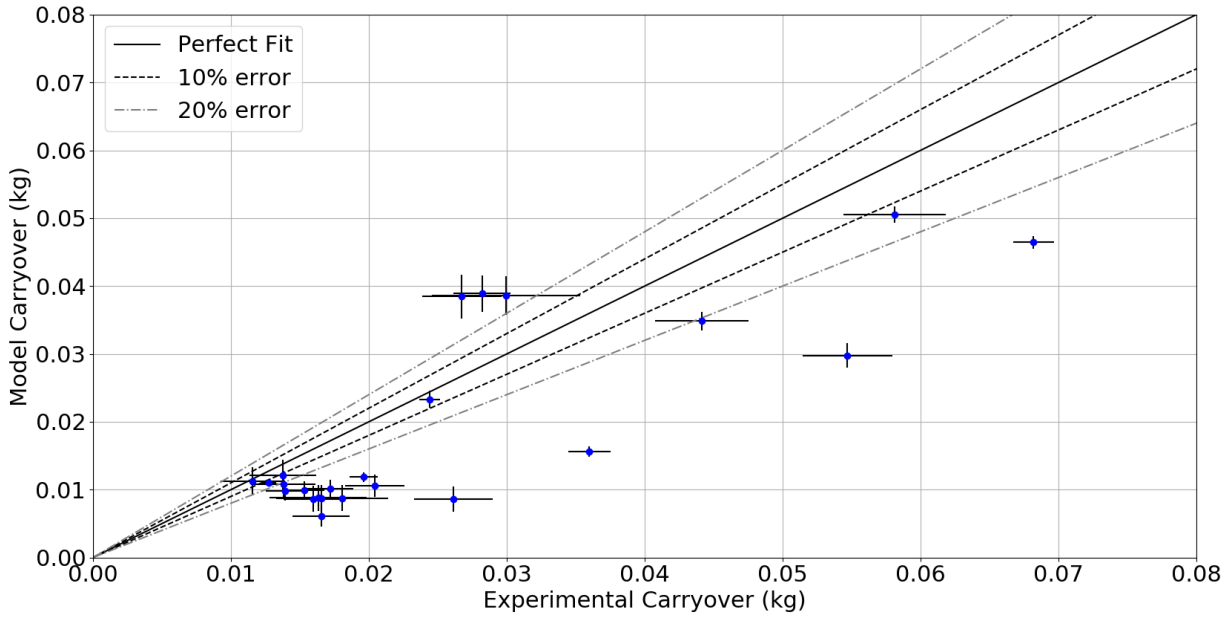


Figure 4.12 - Experimental vs Model carryover results

The carry over experimental results generally do not agree well with the model predications. The model assumes a linear temperature distribution whereas the experiment calculates the carry over using the actual, measured temperature distribution. The disagreement between model and experiment is therefore directly related to the non-linearity of the temperature

distribution associated with the experimental conditions. An example of the model versus experimental temperature distribution is shown in Figure 4.13.

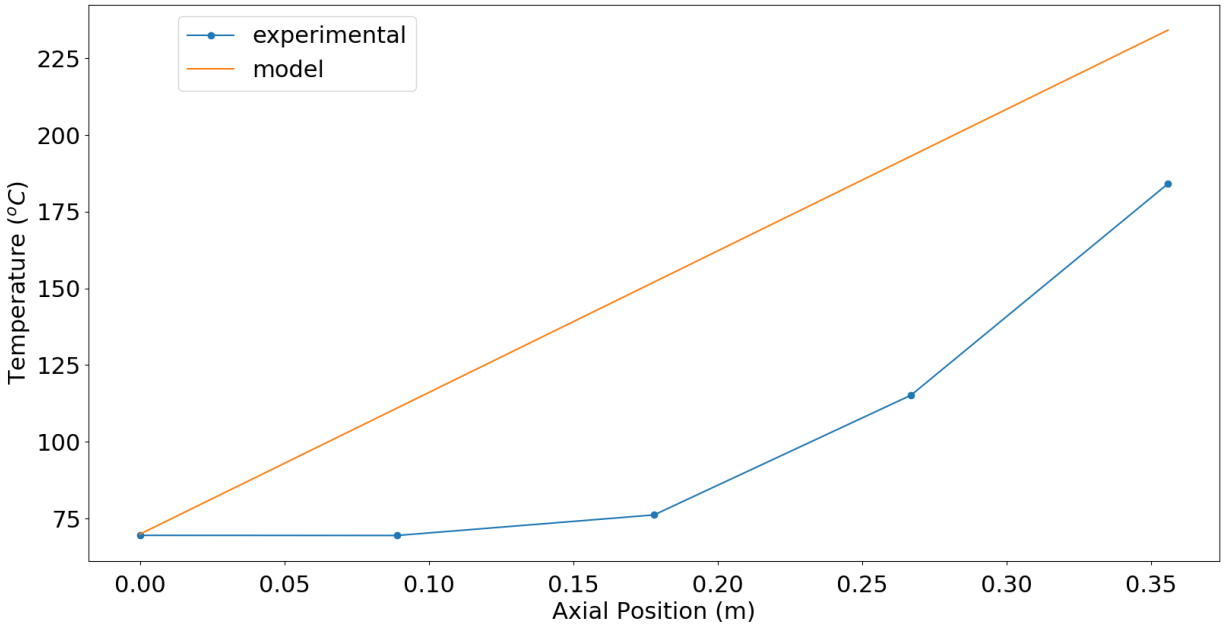


Figure 4.13 - Comparison of assumed linear temperature distribution and the experimental temperature distribution

The temperature distribution is clearly not linear and the model assumption that the fluid varies linearly from the hot inlet temperature to the cold inlet temperature is not correct.

5 CONCLUSIONS

An experimental system to test the performance of switched bed regenerators with sCO2 has been constructed at the University of Wisconsin-Madison. The first-generation regenerator design has been tested and the key results are presented in Section 4. The results generally agree with the NTU-Cm-effectiveness model for effectiveness and pressure drop and give confidence for its use in designing future, larger scale regenerators. The results suggest that the carryover model should be improved before it can be used for this purpose. The next stage

of the project will be scaling up from the current, approximately 10kW size system to a 50kW scale system to be built at Sandia National Laboratory. From the experience constructing and testing at the 10kW scale, a few recommendations for the larger system are:

- The connections at the high-temperature side of the regenerator should be sleeved as described in Section Binary Valves2.3.2 or welded to avoid leaks caused by thermal expansion/contraction
- The differential pressure across the regenerator bed should either be mounted in such a way as to minimize the static pressure in the instrument tubing, or temperature measurements should be made of the stagnant fluid in order to calculate accurate static fluid properties.

Additionally, the Python scripts written for the data analysis presented in this thesis should be adapted and used for data analysis on the larger scale system data. This will save significant time and also ensure consistency of methods. An attempt will be made to include an electronic package with the code, and many of the scripts used are attached to this document as Appendices.

6 BIBLIOGRAPHY

[1] G. Nellis and S. Klein, Heat Transfer, Cambridge, 2009.

[2] E. Erdim, O. Akgiray and I. Demir, "A revisit of pressure drop-flow rate correlations for packed beds of spheres," *Powder Technology,* vol. 283, pp. 488-504, 2015.

[3] R. Hicks, "Pressure Drop in packed beds of spheres," *Industrial & Engineering Chemistry Fundamentals,* vol. 9(3), pp. 530-502, 1970.

[4] V. Gnielinski, "Fluid-Particle Heat Transfer in Flow Through Packed Beds of Solids," in *VDI Heat Atlas*, Second ed., P. Stephan, S. Kabelac, M. Kind, H. Martin, D. Mewes and K. Schaber, Eds., Berlin, Spinger-Verlag, 2010.

[5] N. Wakao and S. Kaguei, Heat and Mass Transfer in Packed Beds, New York: Gordon and Breach, Science Publishers, Inc., 1982.

[6] J. F. Hinze, G. F. Nellis and M. H. Anderson, "Cost comparison of printed circuit heat exchanger to low cost periodic flow regenerator for use as recuperator in a s-CO2 Brayton cycle," *Applied Energy,* vol. 208, pp. 1150-1161, 2017.

[7] U. E. I. Adminsitration, "International Energy Outlook 2016," 2016.

[8] DOE, "Quadrennial Technology Review," 2015.

[9] Executive Office of the President of the United States, "Incorporating Renewables into the Electric Grid: Expanding Opportunities for Smart Markets and Energy Storage," 2016.

[10] M. Romero-Alvarez and E. Zarza, "Concentrating Solar Thermal Power," in *Handbook of Energy Efficiency and Renewable Energy*, Taylor and Francis Group, LLC, 2007, pp. 21-1 - 21-98.

[11] Department of Energy Office of Energy Efficiency and Renewable Energy, "Concentrating Solar Power: Advanced Projects Offering Low LCOE Opportunities - Funding Opportunity Announcement Number: DE-FOA-0001186," 2014.

[12] NREL, "Energy Storage - Possibilities for Expanding Electric Grid Flexibility," 2016.

[13] V. Dostal, M. J. Driscoll and P. Hejzlar, "A Supercritical Carbon Dioxide Cycle for Next Generation Nuclear Reacots," MIT , 2004.

[14] National Energy Technology Laboratory, "Analysis of Brayton Cycles Utilizing Supercritical Carbon Dioxide," 2014.

[15] A. Moisseytsev and J. Sienicki, "Investigation of a Dry Air Cooling Option for an S-CO2 Cycle," in *The 4th International Symposium - Supercritical CO2 Power Cycles*, Pittsburgh, 2014.

[16] V. T. Cheang, R. A. Hedderwick and C. McGregor, "Benchmarking supercritical carbon dioxide cycles against steam Rankine cycles for Concentrated Solar Power," *Solar Energy,* pp. 199-211, 2015.

[17] National Renewable Energy Laboratory, Sandia National Laboratories, "On the Path To SunShot," 2016.

[18] C. K. Ho, M. Carlson, P. Garg and P. Kumar, "Cost and Performance Tradeoffs of Alternative Solar-Driven s-CO2 Bratyon Cycle Configurations," in *Power and Energy Conversion Conference*, San Diego, California, 2015.

[19] C. S. Turchi, Z. Ma, T. W. Neises and M. J. Wager, "Thermodynamic Study of Advanced Supercritical Carbon Dioxide Power Cycles for Concentrating Solar Power Systems," *Journal of Solar Energy Engineering,* vol. 135, no. 041007, pp. 1-7, 2013.

7 APPENDIX A

Main python program – this program is run on the original raw data and performs all of the required analysis to calculate values of interest. While this program can be run from the command line, it was written and is most effective when used in the Python IDE Sypder, although a verison to be used in the command line is available on Github : https://github.com/lrapp/UW-Madison-Switched-Bed-Regenerator-Data-Analysis.git

```
##---- Main ----####
from class_def import props,Q,HT_results,main_char
import time as time

class Timer(object):
    def __init__(self, name=None):
        self.name = name

    def __enter__(self):
        self.tstart = time.time()
    def __exit__(self, type, value, traceback):
        if self.name:
            print('[%s]' % self.name,)
        print('Elapsed: %s' % (time.time() - self.tstart))

with Timer():
    from get_data2 import get_data2 #function used to import raw data
    from create_averaged_vars import create_averaged_vars #function to average pressure data s
o there are equal number of temperature and pressure points
    from cycle_analysis2 import split_cycles2
    from lin_int_cycle2 import lin_int_cycle2
    from fast_ave2 import fast_ave2
    from ave_for_fit import ave_for_fit
    from calc_Q import calc_Q
    from heat_transfer_coef_fn import ht_coef_fn
    import props5
    import os

    refresh_data=True
    refresh_sub=False
    calc = True

    file_date="10_4_17" #Specify file date
    root="C:\\Users\\lrapp\\OneDrive - UW-Madison\\Research\\Data Store\\Data\\"

    folder=root+file_date
    file_list=os.listdir(folder)

    if refresh_data==True:
        print("Getting Data...")
        with Timer():
            [Temp,Pressure,BV]=get_data2(folder)
        print("Get Data Complete")
        print("Averaging Data...")
        with Timer():
            [df]=create_averaged_vars(Temp,Pressure,folder)
        print("Averaged Data Complete")

    print("Interpolating Data...")
    with Timer():
```

```
        df_lin=lin_int_cycle2(df,BV,folder)
    print("Interpolating Complete")

    print("Getting thermophysical properties...")
    with Timer():
        df_full_cols=split_cycles2(df_lin,BV,folder)

##Used to correct dP, corrected dP was then written to file so this does not need to be run every time
#    with Timer():
#        L_top=5*0.0254
#        L_bottom=14*0.0254
#        g=9.81
#        T=40
#        dp_corrected=[]
#        for i in range(0,len(df_full_cols)):
#            Conditions=props(props5.f90wrap_tp(T+273.15,df_full_cols['PT01'][i]*6.89475729))
#            dp_top_i=(Conditions.density*g*L_top)*0.000145038
#            dp_bottom_i=(Conditions.density*g*L_bottom)*0.000145038
#            dp_corrected.append(df_full_cols['DP01'][i]-dp_top_i-dp_bottom_i)
#
    del df_full_cols['DP01']
    df_full_cols['DP01']=dp_corrected


    if calc == True:
        start=1020
        end=1260
        with Timer():
            [S1_ave,S2a_ave,S2b_ave,S2c_ave,S3_ave,S4a_ave,S4b_ave,S4c_ave,full_cycle]=fast_ave2(start,end,df_full_cols)


        Regen_Results=HT_results(ave_for_fit(full_cycle,1))

        q_dot_max_RE1=Regen_Results.q_dot_max_RE1
        q_dot_max_RE2=Regen_Results.q_dot_max_RE2

        Q_re=Q(calc_Q(start,end,df_full_cols))

        MC=main_char()

        MC=ht_coef_fn(start,end,df_full_cols)
```

8 APPENDIX B – CO2 PROPERTIES

The thermodynamic properties of CO2 were calculated using “FIT Carbon Dioxide” developed by Northland Numerics for the Solar Energy Laboratory at the University of Wisconsin-Madison. This function was provided for this project as a Fotran90 module. The header information for this file is shown below:

module_CO2_properties.f90 :

Temperature Range: 216.592 K to 2,000.0 K
! Pressure Range: 0.001 Pa to 800.0 MPa

```
! FIT Version: 2a71ddde4717
!
! ------------------------------------------------------------------------------------------------------------------------
! Copyright (c) 2017, Northland Numerics LLC
! All rights reserved.
!
! THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ANY WARRANTY BE CREATED IN
CONNECTION WITH THE SALE OF SOFTWARE, UNLESS THE WARRANTY WAS CREATED SOLELY DUE TO A WRITTEN
AGREEMENT SIGNED BY SELLER. IN NO EVENT SHALL ANY WARRANTY BE IMPUTED OR PRESUMED. IN NO EVENT
SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, BUSINESS INTERRUPTION;
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; OR LOSS OF USE, DATA, OR PROFITS) HOWEVER CAUSED
AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF SUCH DAMAGE.
!
! This copy of FIT Carbon Dioxide is for use only by members of the Solar Energy Laboratory (SEL) at the University
of Wisconsin-Madison for SEL-related projects.
! ------------------------------------------------------------------------------------------------------------------------
!
! This Fortran module contains a number of subroutines that return carbon dioxide properties calculated using an
interpolated Helmholtz free energy and its analytical derivatives.  The main property subroutines are:
!
!   CO2_TD(T, D, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!   CO2_TP(T, P, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!   CO2_PH(P, H, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!   CO2_PS(P, S, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!   CO2_HS(H, S, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!   CO2_TQ(T, Q, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!   CO2_PQ(P, Q, error_code, temp, pres, dens, vol, qual, inte, enth, entr, cv, cp, ssnd, visc, cond)
!
! The first two arguments are required inputs that correspond to the known independent properties:
!
!   T -- temperature (K)
!   D -- density (kg/m3)
!   P -- pressure (kPa)
!   H -- enthalpy (kJ/kg)
!   S -- entropy (kJ/kg-K)
!   Q -- quality on a mass basis
!
! Each subroutine requires only a single output (error_code), which is an integer used to indicate
! success (value is 0), an error (positive value), or a warning (negative value).  If an error or warning occurs, the
function CO2_error_message(error_code), which returns a 255 character string, can be used to get more
information about the error code.
!
! All of the remaining arguments in the subroutines are optional outputs:
!
!   temp -- temperature (K)
!   pres -- pressure (kPa)
!   dens -- density (kg/m3)
!   vol  -- specific volume (m3/kg)
```

```
!  qual -- quality on a mass basis that follows the REFPROP convention of:
!        q < 0 is a subcooled liquid
!        q = 0 is a saturated liquid
!        q = 1 is a saturated vapor
!        q > 1 is a superheated vapor
!        q = 998 is a superheated vapor with temperature greater than the critical temperature
!        q = 999 is supercritical, with temperature and pressure greater than the critical point
!  inte -- internal energy (kJ/kg)
!  enth -- enthalpy (kJ/kg)
!  entr -- entropy (kJ/kg-K)
!  cv   -- specific heat at constant volume (kJ/kg-K)
!  cp   -- specific heat at constant pressure (kJ/kg-K)
!  ssnd -- speed of sound in the fluid (m/s)
!  visc -- viscosity (uPa-s)
!  cond -- thermal conductivity (W/m-K)
!
! Partial derivatives of various properties are available from the subroutine:
!  CO2_derivatives(T, D, error_code, dPdD_T, dhdD_T, dsdD_T, dPdT_D, dhdT_D, dsdT_D, dDdP_T, dDdT_P)
!   dPdD_T -- derivative of pressure with respect to density at constant temperature (m3-kPa/kg)
!   dhdD_T -- derivative of enthalpy with respect to density at constant temperature (m3-kJ/kg2)
!   dsdD_T -- derivative of entropy with respect to density at constant temperature  (m3-kJ/kg2-K)
!   dPdT_D -- derivative of pressure with respect to temperature at constant density (kPa/K)
!   dhdT_D -- derivative of enthalpy with respect to temperature at constant density (kJ/kg-K)
!   dsdT_D -- derivative of entropy with respect to temperature at constant density  (kJ/kg-K2)
!   dDdP_T -- derivative of density with respect to pressure at constant temperature (kg/m3-kPa)
!   dDdT_P -- derivative of density with respect to temperature at constant pressure (kg/m3-K)
!   (all the above derivative outputs are optional; only error_code is required)
!
! Additional functions that are available:
!  CO2_sat_pres(T) -- saturated pressure (kPa) as a function of temperature (K) [valid for sat_temp_min <= T < T_critical]
!  CO2_sat_temp(P) -- saturated temperature (K) as a function of pressure (kPa) [valid for sat_pres_min <= P < P_critical]
!  CO2_sat_temp_derivative(P) -- derivative of sat. temperature w.r.t. pressure (K/kPa) as a function of pressure (kPa)
!  CO2_surf_tension(T) -- surface tension (N/m) as a function of temperature (K) [valid for sat_temp_min <= T < T_critical]
! Warning: The above functions will return -9e99 if the input is not valid.
!
! Notes:
!  1) The thermodynamic state is only explicitly defined in temperature and density.  Therefore, any other combination of
!   known properties requires iteration and may result in a state that does not exactly correspond to the provided properties.
!    For this reason, the values that are returned for the specified properties may not be identical to the inputs.
However, the returned values do exactly correspond to the thermodynamic state defined by the returned
temperature and density.  2) The CO2_info subroutine provides the molar mass and the critical temperature,
pressure, and density of the fluid. 3) The CO2_limits subroutine provides temperature and pressure limits for this
implementation of FIT.
!  4) All values are double precision with the exception of 'error_code', which is an integer.
!  5) All subroutines and functions are elemental, meaning arrays of inputs and outputs can be used.
!
! Examples:
!  call CO2_TD(T=T, D=D, error_code=error_code, pres=pressure, cp=spec_heat)
!  call CO2_TP(T=temp, P=pres, error_code=err_flag, ssnd=speed_of_sound)
!  call CO2_PH(P=pres_array, H=enth_array, error_code=error_code_array, dens=D_array, temp=temp_array)
!  call CO2_PS(P=pres_out, S=entr_out, error_code=err, enth=enth_out, temp=temp_out)
!  call CO2_TQ(T=T, Q=1.0_dp, error_code=error_code, pres=sat_pres, cv=sat_vap_cv)
!
```

```
! For more information, contact Northland Numerics at: support@nnumerics.com
```

To use the various functions defined in “module_CO2_properties.f90”, another Fortran file was written with subroutines that called specific subroutines from the “moduel_CO2_properties.f90” file. For example, to calculate properties given temperature and pressure, a subroutine TP was defined as:

props5.f90 :

```
subroutine TP(T1,P1,temp1, pres1, dens1, vol1, qual1, inte1, enth1, entr1, cv1, cp1, ssnd1, visc1, cond1)
        use CO2_properties
        implicit none
        integer, parameter  :: dp = selected_real_kind(15)
        Real(dp), INTENT(IN) :: T1,P1
        Real(dp), INTENT(OUT) :: temp1, pres1, dens1, vol1, qual1, inte1, enth1, entr1, cv1, cp1, ssnd1, visc1,
                cond1
        integer :: er
        call CO2_TP(T=T1,P=P1,error_code=er,temp=temp1, pres=pres1, dens=dens1, vol=vol1,qual=qual1,
                inte=inte1,enth=enth1, entr=entr1, cv=cv1, cp=cp1, ssnd=ssnd1, visc=visc1, cond=cond1)
end SUBROUTINE TP
```

gfortran was then used to compile both “module_CO2_properties.f90” and “props5.f90”.

```
$gfortran –c module_CO2_properties.f90

$gfortran –c props5.f90
```

This creates an object and module file for “module_CO2_properties.f90” and an object file for “props5.f90”. Because derived data types were used in the “module_CO2_properties.f90” file, it can not be used directly with f2py. First, it must be run through another python script, f90wrap, which creates an interface which is suitable for wrapping with f2py. To use f90wrap, run from the command line:

```
$python f90wrap –m properties module_CO2_properties.f90 props5.f90
```

This command creates “wrapped” fortran90 files that can be read by f2py:

```
$python f2py.py –c –m props5 module_co2_properties.o props5.o
        f90wrap_moduel_CO2_properties.f90 f90wrap_toplevel.f90
```

This creates a props5.pyd file, which is similar to a windows DLL. The props5.pyd can be imported into python just like any other external package:

```
>>>import  props
```

running the python help with the custom module results in:

```
$ python
Python 3.6.1 |Anaconda custom (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import props5
>>> help(props5)
Help on module props5:

NAME
    props5

DESCRIPTION
    This module 'props5' is auto-generated with f2py (version:2).
    Functions:
      temp1,pres1,dens1,vol1,qual1,inte1,enth1,entr1,cv1,cp1,ssnd1,visc1,cond1 = f90wrap_tp(t1,p1)
      temp1,pres1,dens1,vol1,qual1,inte1,enth1,entr1,cv1,cp1,ssnd1,visc1,cond1 = f90wrap_ph(p1,h1)
      temp1,pres1,dens1,vol1,qual1,inte1,enth1,entr1,cv1,cp1,ssnd1,visc1,cond1 = f90wrap_ps(p1,s1)
      temp1,pres1,dens1,vol1,qual1,inte1,enth1,entr1,cv1,cp1,ssnd1,visc1,cond1 = f90wrap_tq(t1,q1)
    .

DATA
    f90wrap_ph = <fortran object>
    f90wrap_ps = <fortran object>
    f90wrap_tp = <fortran object>
    f90wrap_tq = <fortran object>

VERSION
    b'$Revision: $'
```

9 APPENDIX C – FAST AVERAGE

In order to get cyclic averaged experimental data, the various states were averaged over a steady state operating period as described in Section 3.3.

The python script goes through the dataframe of experimental data and creates lists of each unique state between the given 'start' and 'end' cycle numbers as shown in Figure 9.1.

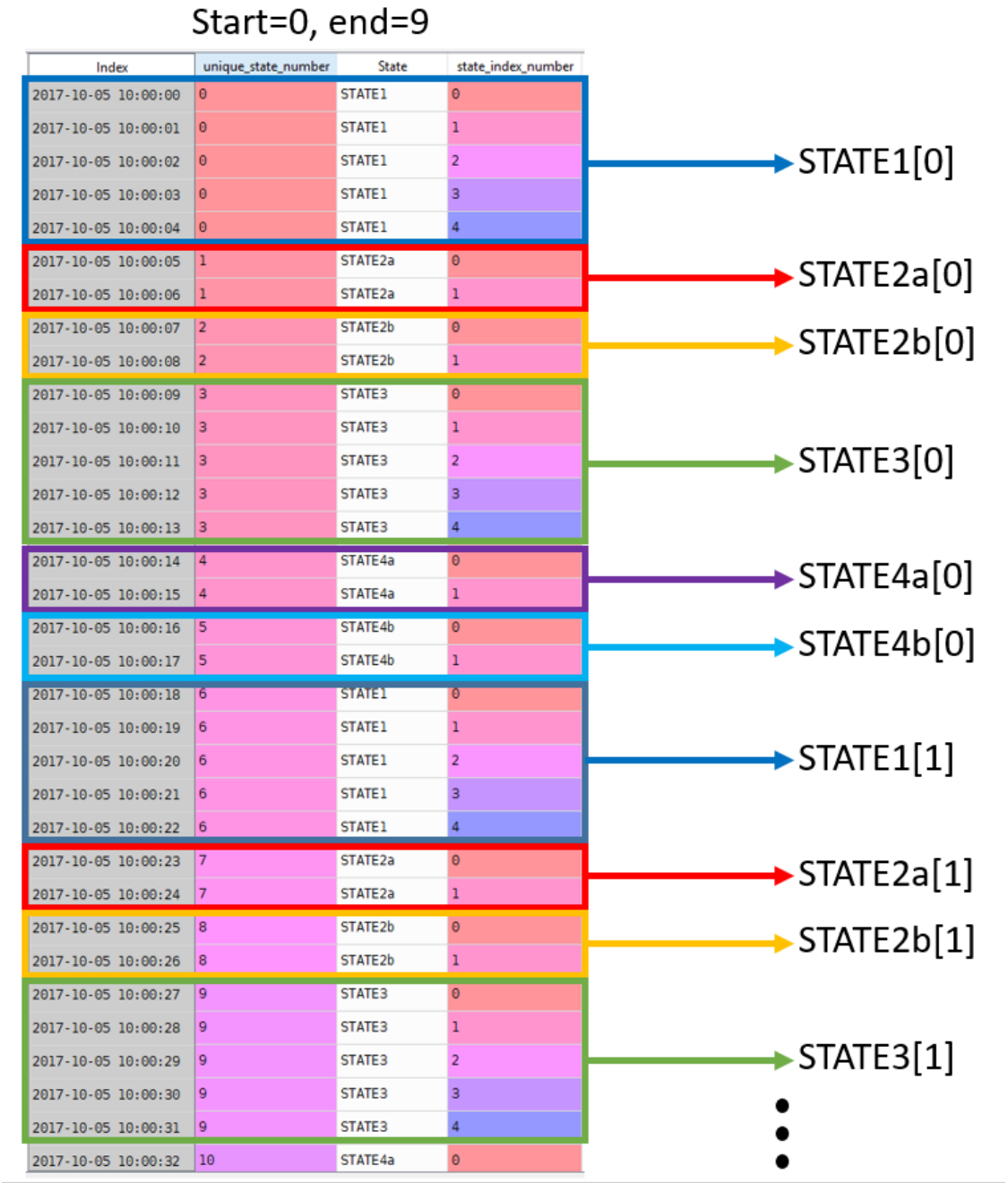


Figure 9.1 - Visual Depiction of fast_ave python script

Each list can then be averaged by the 'state_index_number' to create an averaged state. The "fast_ave" python script returns each averaged state as well as a dataframe called full_cycle which is each averaged state combined to form a full cycle.

```
def fast_ave2(start,end,df_full_cols):
    import numpy as np
    import pandas as pd

    st1_list=[]
    st2a_list=[]
    st2b_list=[]
    st2c_list=[]
    st3_list=[]
    st4a_list=[]
    st4b_list=[]
    st4c_list=[]

    ave_list=[st1_list,st2a_list,st2b_list,st2c_list,st3_list,st4a_list,st4b_list,st4c_list]
    possible_states=['STATE1','STATE2a','STATE2b','STATE2c','STATE3','STATE4a','STATE4b','STATE4c']

    for i in range(start,end):
        num=len(df_full_cols.loc[df_full_cols['unique_state_num']==i])
        for k in range(0,len(possible_states)):
            if df_full_cols.loc[df_full_cols['unique_state_num']==i]['state'][0] == possible_states[k]:
                df_i=df_full_cols.loc[df_full_cols['unique_state_num']==i]
                time_d=[]
                for j in range(0,num):
                    time_d.append((df_i.index[j]-df_i.index[0]).total_seconds())
                df_i.insert(0,'rel_time',time_d)
                ave_list[k].append(df_i)


    min_list=[]
    for i in range(0,len(ave_list)):
        min_list.append(len(ave_list[i]))

    min_array=np.asarray(min_list)
    min_length=np.min(min_array[np.nonzero(min_array)])

    cut_ave_list=[]
    for i in range(0,len(ave_list)):
        cut_ave_list.append(ave_list[i][0:min_length])

    mean_list=[]
    std_list=[]
    for i in range(0,len(ave_list)):
        if len(cut_ave_list[i])!=0:

            mean_list.append(pd.concat(cut_ave_list[i]).groupby('state_index_num').mean())
            std_list.append(pd.concat(cut_ave_list[i]).groupby('state_index_num').std())

            k=0
            if len(mean_list[-1])!=len(cut_ave_list[i][0]['state'].tolist()):

                while  k < len(cut_ave_list[i]) and len(mean_list[-1])!=len(cut_ave_list[i][k]['state'].tolist()):
                    k=k+1

```

```
            mean_list[-1]['state']=cut_ave_list[i][k]['state'].tolist()
    #         std_list[-1]['state']=cut_ave_list[i][0]['state'].tolist()

    for i in range(0,len(std_list)):
        del mean_list[i]['unique_state_num']
        del std_list[i]['unique_state_num']
        new_cols=[x+'_ave' for x in mean_list[i].columns[0:-
1].tolist()]+[mean_list[i].columns[-1]]
    #    new_cols_std=[x+'_std' for x in mean_list[i].columns[0:-
1].tolist()]+[mean_list[i].columns[-1]]
        new_cols_std=[x+'_std' for x in mean_list[i].columns[0:-1].tolist()]
        mean_list[i].columns=new_cols
        std_list[i].columns=new_cols_std

    mean_list_state=[]
    for i in range(0,len(mean_list)):
        mean_list_state.append(mean_list[i]['state'][0])

    average_state_list=[]
    for i in range(0,len(mean_list)):
        average_state_list.append(pd.concat((mean_list[i],std_list[i]),axis=1))

    for i in range(1,len(average_state_list)):
        if average_state_list[i-1]['TI16_ave'][-
1:].values[0] == average_state_list[i]['TI16_ave'][0:].values[0]:
            average_state_list[i]=average_state_list[i].drop(0)
            average_state_list[i].reset_index()

    #clean up
    full=pd.concat(average_state_list,ignore_index=True)
    full1=full
    for i in range(1,len(full1)):
        if full1['rel_time_ave'][i]==0:
            full1=full1.drop(i)
    cols=full1.columns.tolist()
    st=cols.pop(cols.index('state'))
    cols2=cols[0:1]+[st]+cols[1:-1]
    full1=full1[cols2]
    full1=full1.reset_index()
    del full1['index']

    #check if any states are empty, if so do not include them in the final full dataframe
    asl2=[]
    if len(full.loc[full['state']=='STATE1'])!=0:
        st1=full.loc[full['state']=='STATE1']
        asl2.append(st1)
    if len(full.loc[full['state']=='STATE2a'])!=0:
        st2a=full.loc[full['state']=='STATE2a']
        asl2.append(st2a)
    if len(full.loc[full['state']=='STATE2b'])!=0:
         st2b=full.loc[full['state']=='STATE2b']
         asl2.append(st2b)
     if len(full.loc[full['state']=='STATE2c'])!=0:
         st2c=full.loc[full['state']=='STATE2c']
         asl2.append(st2c)
     if len(full.loc[full['state']=='STATE3'])!=0:
         st3=full.loc[full['state']=='STATE3']
         asl2.append(st3)
     if len(full.loc[full['state']=='STATE4a'])!=0:
         st4a=full.loc[full['state']=='STATE4a']
         asl2.append(st4a)
```

```
    if len(full.loc[full['state']=='STATE4b'])!=0:
        st4b=full.loc[full['state']=='STATE4b']
        asl2.append(st4b)
    if len(full.loc[full['state']=='STATE4c'])!=0:
        st4c=full.loc[full['state']=='STATE4c']
        asl2.append(st4c)


    for i in range(1,len(asl2)):
        asl2[i]['rel_time_ave']=asl2[i]['rel_time_ave'].values+asl2[i-1]['rel_time_ave'][-1:].values[0]

    full2=pd.concat(asl2,ignore_index=True)
    cols=full2.columns.tolist()
    st=cols.pop(cols.index('state'))
    cols2=cols[0:1]+[st]+cols[1:-1]
    full2=full2[cols2]

    S1_ave=full.loc[full['state']=="STATE1"]
    S2a_ave=full.loc[full['state']=="STATE2a"]
    S2b_ave=full.loc[full['state']=="STATE2b"]
    S2c_ave=full.loc[full['state']=="STATE2c"]
    S3_ave=full.loc[full['state']=="STATE3"]
    S4a_ave=full.loc[full['state']=="STATE4a"]
    S4b_ave=full.loc[full['state']=="STATE4b"]
    S4c_ave=full.loc[full['state']=="STATE4c"]

    return [S1_ave,S2a_ave,S2b_ave,S2c_ave,S3_ave,S4a_ave,S4b_ave,S4c_ave,full2]
```

10 APPENDIX D – CALCULATE Q

```
import numpy as np
import pandas as pd
from scipy import integrate

def calc_Q2(start,end,df_full_cols):


    def integrate_method(self, how='trapz', unit='s'):
        '''''Numerically integrate the time series.

        @param how: the method to use (trapz by default)
        @return
        Available methods:
         * trapz - trapezoidal
         * cumtrapz - cumulative trapezoidal
         * simps - Simpson's rule
         * romb - Romberger's rule

        See http://docs.scipy.org/doc/scipy/reference/integrate.html for the method details.
        or the source code
        https://github.com/scipy/scipy/blob/master/scipy/integrate/quadrature.py
        '''
        available_rules = set(['trapz', 'cumtrapz', 'simps', 'romb'])
        if how in available_rules:
            rule = integrate.__getattribute__(how)
        else:
            print('Unsupported integration rule: %s' % (how))
            print('Expecting one of these sample-
based integration rules: %s' % (str(list(available_rules))))
            raise AttributeError

        result = rule(self.values, self.index.astype(np.int64) / 10**9)
        #result = rule(self.values)
        return result

#    pd.TimeSeries.integrate = integrate_method
    pd.Series.integrate = integrate_method


    Q_C1=[]
    Q_H1=[]
    Q_C2=[]
    Q_H2=[]
    Q_H2_ST2a=[]
    Q_C1_ST2a=[]
    Q_H2_ST2b=[]
    Q_C1_ST2b=[]

    Q_H2_ST4a=[]
    Q_C1_ST4a=[]
    Q_H2_ST4b=[]
    Q_C1_ST4b=[]

    Q16=0
    Q12=0
    Q11=0
    Q07=0
```

```
57.
58.     possible_states=['STATE1','STATE2a','STATE2b','STATE2c',
59.                      'STATE3','STATE4a','STATE4b','STATE4c']
60.
61.     possible_states=['STATE1','STATE3','STATE2a']
62.     Q_combos={'STATE1' : [[Q_H2,Q16,Q12],[Q_C1,Q11,Q07]],
63.               'STATE3' : [[Q_H1,Q11,Q07],[Q_C2,Q16,Q12]],
64.               'STATE2a' : [[Q_H2_ST2a,Q16,Q12],[Q_C1_ST2a,Q11,Q07]],
65.               'STATE2b' : [[Q_H2_ST2b,Q16,Q12],[Q_C1_ST2b,Q11,Q07]],
66.               'STATE4a' : [[Q_H2_ST4a,Q16,Q12],[Q_C1_ST4a,Q11,Q07]],
67.               'STATE4b' : [[Q_H2_ST4b,Q16,Q12],[Q_C1_ST4b,Q11,Q07]]}
68.
69.     half_cycle_time=[]
70.     for i in range(start,end):
71.         df_i=df_full_cols.loc[df_full_cols['unique_state_num']==i]
72.         Q16=(integrate.trapz(df_i['h16']*df_i['FI01'],df_i.index.astype(np.int64)/10**9))
73.         Q12=(integrate.trapz(df_i['h12']*df_i['FI01'],df_i.index.astype(np.int64)/10**9))
74.         Q11=(integrate.trapz(df_i['h11']*df_i['FI01'],df_i.index.astype(np.int64)/10**9))
75.         Q07=(integrate.trapz(df_i['h07']*df_i['FI01'],df_i.index.astype(np.int64)/10**9))

76.         Q_combos={'STATE1' : [[Q_H2,Q16,Q12],[Q_C1,Q11,Q07]],
77.                   'STATE3' : [[Q_H1,Q11,Q07],[Q_C2,Q16,Q12]],
78.                   'STATE2a' : [[Q_H2_ST2a,Q16,Q12],[Q_C1_ST2a,Q11,Q07]],
79.                   'STATE2b' : [[Q_H2_ST2b,Q16,Q12],[Q_C1_ST2b,Q11,Q07]],
80.                   'STATE4a' : [[Q_H2_ST4a,Q16,Q12],[Q_C1_ST4a,Q11,Q07]],
81.                   'STATE4b' : [[Q_H2_ST4b,Q16,Q12],[Q_C1_ST4b,Q11,Q07]]}
82.         for k in range(0,len(possible_states)):
83.             if df_full_cols.loc[df_full_cols['unique_state_num']==i]['state'][0] == possible_s
    tates[k]:
84.                 Q_combos[possible_states[k]][0][0].append(Q_combos[possible_states[k]][0][1]-
    Q_combos[possible_states[k]][0][2])
85.                 Q_combos[possible_states[k]][1][0].append(Q_combos[possible_states[k]][1][1]-
    Q_combos[possible_states[k]][1][2])
86.             if df_full_cols.loc[df_full_cols['unique_state_num']==i]['state'][0] == 'STATE1':

87.                 half_cycle_time.append((df_i.index[-1]-
    df_i.index[0]).total_seconds())
88.
89.
90.     Q_C1_ave=np.mean(Q_C1)
91.     Q_H1_ave=np.mean(Q_H1)
92.     Q_C2_ave=np.mean(Q_C2)
93.     Q_H2_ave=np.mean(Q_H2)
94.     half_cycle_time_ave=np.mean(half_cycle_time)
95.     Q_C1_std=np.std(Q_C1)
96.     Q_H1_std=np.std(Q_H1)
97.     Q_C2_std=np.std(Q_C2)
98.     Q_H2_std=np.std(Q_H2)
99.
100.        Q_C1_st2a_ave=np.mean(Q_C1_ST2a)
101.        Q_C1_st2b_ave=np.mean(Q_C1_ST2b)
102.
103.
104.
105.
106.        return [Q_C1_ave,Q_H1_ave,
107.                Q_C2_ave,Q_H2_ave,
108.                Q_C1_std,Q_H1_std,
109.                Q_C2_std,Q_H2_std,
110.                half_cycle_time_ave,
111.                Q_C1_st2a_ave,Q_C1_st2b_ave]
```

11 APPENDIX E – FIT_HXR

The FIT_HXR Fortran function was developed by Northland Numerics for use by the Solar Energy Laboratory at the University of Wisconsin-Madison. It employs a sub-heat exchanger model as described in [1] to calculate the performance of a counterflow heat exchanger given inlet conditions and a minimum delta T, a UA, or an effectiveness target value. The Fortran subroutines were made callable in Python by a similar procedure as described in Appendix B – CO2 Properties. The python function "ave_for_fit2" is shown below. This function is used to calculate q_max using the FIT_HXR Fortran subroutine with the averaged cycle data and target effectiveness as inputs.

ave_for_fit2.py

```
import FIT_HXR
from scipy import integrate

def ave_for_fit2(full,target):
#    target=1
    target_code=1
    n_hxrs=25

    dp_1=0
    dp_2=dp_1

    fluid_1='carbondioxide'
    fluid_2='carbondioxide'

    P_convert=6.89475729

    S1_time=list(full.loc[full['state']=='STATE1']['rel_time_ave'])[-1]
    valve_time=0

    #t_in_1 is T12_ave of state1
    t_in_1_RE2=full.loc[full['state']=='STATE1']["TI12"+"_ave"].mean()
    t_in_1_RE2_std=full.loc[full['state']=='STATE1']["TI12"+"_std"].mean()

    #t_in_2 is T16_ave of state3
    t_in_2_RE2=full.loc[full['state']=='STATE3']["TI16"+"_ave"].mean()
    t_in_2_RE2_std=full.loc[full['state']=='STATE3']["TI16"+"_std"].mean()

    #p_in_1 is PT03_ave of state1
    p_in_1_RE2=full.loc[full['state']=='STATE1']["PT03"+"_ave"].mean()*P_convert

    p_in_1_RE2_std=full.loc[full['state']=='STATE1']["PT03"+"_std"].mean()*P_convert

    #p_in_2 is PT03_ave of state3
    p_in_2_RE2=full.loc[full['state']=='STATE3']["PT03"+"_ave"].mean()*P_convert
    p_in_2_RE2_std=full.loc[full['state']=='STATE3']["PT03"+"_std"].mean()*P_convert

    m_dot_1_RE2=integrate.trapz((1/(S1_time+valve_time))*full.loc[full['state']=='STATE1']["FI01"+"_ave"],full.loc[full['state']=='STATE1']['rel_time_ave'])

    m_dot_1_RE2_std=full.loc[full['state']=='STATE1']["FI01"+"_std"].mean()
    m_dot_2_RE2=integrate.trapz((1/(S1_time+valve_time))*full.loc[full['state']=='STATE3']["FI01"+"_ave"],full.loc[full['state']=='STATE3']['rel_time_ave'])

    m_dot_2_RE2_std=full.loc[full['state']=='STATE3']["FI01"+"_std"].mean()

```

```
    #t_in_1 is T12_ave of state1
    t_in_1_RE1=full.loc[full['state']=='STATE3']["TI07"+"_ave"].mean()
    t_in_1_RE1_std=full.loc[full['state']=='STATE3']["TI07"+"_std"].mean()

    #t_in_2 is T16_ave of state3
    t_in_2_RE1=full.loc[full['state']=='STATE1']["TI11"+"_ave"].mean()
    t_in_2_RE1_std=full.loc[full['state']=='STATE1']["TI11"+"_std"].mean()

    #p_in_1 is PT03_ave of state1
    p_in_1_RE1=full.loc[full['state']=='STATE3']["PT01"+"_ave"].mean()*P_convert

    p_in_1_RE1_std=full.loc[full['state']=='STATE3']["PT01"+"_std"].mean()*P_convert

    #p_in_2 is PT03_ave of state3
    p_in_2_RE1=full.loc[full['state']=='STATE1']["PT01"+"_ave"].mean()*P_convert
    p_in_2_RE1_std=full.loc[full['state']=='STATE1']["PT01"+"_std"].mean()*P_convert

    m_dot_1_RE1=integrate.trapz((1/(S1_time+valve_time))*full.loc[full['state']=='STATE3']["FI
01"+"_ave"],
                                full.loc[full['state']=='STATE3']['rel_time_ave'])
    m_dot_1_RE1_std=full.loc[full['state']=='STATE3']["FI01"+"_std"].mean()

    m_dot_2_RE1_std=full.loc[full['state']=='STATE1']["FI01"+"_std"].mean()
    m_dot_2_RE1=integrate.trapz((1/(S1_time+valve_time))*full.loc[full['state']=='STATE1']["FI
01"+"_ave"],
                                full.loc[full['state']=='STATE1']['rel_time_ave'])


    [epsilon_target_RE1,dt_min_RE1,ua_RE1,q_dot_RE1,q_dot_max_RE1,t_1_RE1,t_2_RE1]=FIT_HXR.cou
nter_flow(fluid_1,fluid_2,target,target_code,n_hxrs,m_dot_1_RE1,m_dot_2_RE1,t_in_1_RE1,t_in_2_
RE1,p_in_1_RE1,p_in_2_RE1,dp_1,dp_2)

    [epsilon_target_RE2,dt_min_RE2,ua_RE2,q_dot_RE2,q_dot_max_RE2,t_1_RE2,t_2_RE2]=FIT_HXR.cou
nter_flow(fluid_1,fluid_2,target,target_code,n_hxrs,m_dot_1_RE2,m_dot_2_RE2,t_in_1_RE2,t_in_2_
RE2,p_in_1_RE2,p_in_2_RE2,dp_1,dp_2)


    return [[epsilon_target_RE1,dt_min_RE1,ua_RE1,q_dot_RE1,q_dot_max_RE1,t_1_RE1,t_2_RE1],
            [epsilon_target_RE2,dt_min_RE2,ua_RE2,q_dot_RE2,q_dot_max_RE2,t_1_RE2,t_2_RE2]]
```

12 APPENDIX F – CREATE_AVERAGED_VARS

This program takes in the raw temperature and pressure data and averages the pressure data to be the same length as the temperature data.

```
def create_averaged_vars(Temp,Pressure,folder):
    import pandas as pd
    import os
    from dateutil import parser

    file_list=os.listdir(folder) #List the files in the directory

    end_search=""
    file_found=""

    for file in file_list:
        if file == "df.csv":
            end_search=True
        if file == "Pressure_ave.csv" or file == "Temp.ave.csv":
            file_found="True"

    if end_search!=True:
        if file_found=="True":
            Pressure_average=pd.read_csv(folder+"\\Pressure_ave.csv", index_col=[0])

            Pressure_average.index=Pressure_average.index.map(parser.parse)
            Temp_average=pd.read_csv(folder+"\\Temp_ave.csv", index_col=[0])
            Temp_average.index=Temp_average.index.map(parser.parse)
        else:
            Temp_average=Temp[0:1]
            Pressure_average=pd.DataFrame()
            old_k=0
            k=0
            for i in range(0,len(Temp)-1):
                print(i)
                Temp_i_index=Temp.index[i]
                Pressure_i_index=Pressure.index[k]
                while Pressure_i_index<=Temp_i_index:
                    k=k+1
                    Pressure_i_index=Pressure.index[k]

                df=pd.DataFrame(Pressure[old_k:k].mean().to_dict(),index=[Temp_i_index]
)
                Pressure_average=Pressure_average.append(df)
                Temp_average=Temp_average.append(Temp[i:i+1])
                old_k=k
        df=pd.concat([Temp_average,Pressure_average],axis=1)
        df.to_csv(folder+"\\df.csv")
    else: #if df.csv is found, read in file
        df=pd.read_csv(folder+"\\df.csv",index_col=[0],parse_dates=True)

    return [df]
```

13 APPENDIX G – LINEAR INTERPOLATION

This function was used to ensure a data point existed exactly on each switching time by using linear interpolation.

```
def lin_int_cycle2(df,BV,folder):

#old args    cycle_original,BV,folder
    import pandas as pd
    import os
    import dateutil.parser
    import numpy as np

    file_list=os.listdir(folder)


    if 'DFF.csv' in file_list:
        DFF=pd.read_csv(folder+"\\DFF.csv",index_col=[0],parse_dates=True)
        return DFF

    cols=df.columns

    unique_cntr=0

    DFF=pd.DataFrame()
    for i in range(0,len(BV)-2):

        df_i=df[df.index>=dateutil.parser.parse(str(BV.index[i])) ] #select all data wi
th index greater than the ith valve swtiching start time
        df_i=df_i[df_i.index<=dateutil.parser.parse(str(BV.index[i+1]))]

        df_iplus=df[df.index>=dateutil.parser.parse(str(BV.index[i+1])) ] #select all d
ata with index greater than the ith+1 valve swtiching start time
        df_iplus=df_iplus[df_iplus.index<=dateutil.parser.parse(str(BV.index[i+2]))]

        x_bv0=BV.index[i]
        x_bv1=BV.index[i+1]
        d0={}
        d1={}
        if i==0:
            for j in range(0,len(cols)):
                y0=df_i[cols[j]].values[0]                  #select the first value
                d0.update({cols[j]:y0})
                if cols[j]=='state':

                    y_bv1=df_i[cols[j]].values[-1]
                else:

                    if len(df_iplus)!= 0:
                        y3=df_iplus[cols[j]].values[0]
                        y2=df_i[cols[j]].values[-1]
                        x2=df_i.index[-1]
                        x3=df_iplus.index[0]

                        y_bv1=(y2*(x3-x_bv1)+y3*(x_bv1-x2))/(x3-x2) #interpolate
                    d1.update({cols[j]:y_bv1})


        else:
```

```
            df_iminus=df[df.index>=dateutil.parser.parse(str(BV.index[i-1])) ] #select all data with index greater than valve swtiching time
            df_iminus=df_iminus[df_iminus.index<=dateutil.parser.parse(str(BV.index[i]))]

            for j in range(0,len(cols)):
                if cols[j]=='state':
                    y_bv0=df_i[cols[j]].values[-1]
                    y_bv1=df_i[cols[j]].values[-1]
                    d0.update({cols[j]:y_bv0})
                    d1.update({cols[j]:y_bv1})

                else:
                    y1=df_i[cols[j]].values[0]
                    y0=df_iminus[cols[j]].values[-1]
                    x0=df_iminus.index[-1]
                    x1=df_i.index[0]


                    y_bv0=(y0*(x1-x_bv0)+y1*(x_bv0-x0))/(x1-x0)
                    d0.update({cols[j]:y_bv0})

                    if len(df_iplus)!= 0:
                        y3=df_iplus[cols[j]].values[0]
                        y2=df_i[cols[j]].values[-1]
                        x2=df_i.index[-1]
                        x3=df_iplus.index[0]
                        y_bv1=(y2*(x3-x_bv1)+y3*(x_bv1-x2))/(x3-x2)
                        d1.update({cols[j]:y_bv1})


        df_j0=pd.DataFrame(d0,index=[x_bv0]) #at x_bv0 index add d0 values
        df_j1=pd.DataFrame(d1,index=[x_bv1])
        df_j=df_j0.append(df_i) #append original cycle values to df_j0 and call it df_j

        df_j=df_j.append(df_j1)

        state_index_cntr=list(np.arange(0,len(df_j)))
        df_j['state_index_num']=state_index_cntr

        state_i=[]
        for ii in range(0,len(df_j)):
            state_i.append(BV.Position[i])

        df_j['state']=state_i
        unique_state_num=list(np.ones(len(df_j))*unique_cntr)
        df_j['unique_state_num']=unique_state_num
        unique_cntr=unique_cntr+1
        DFF=DFF.append(df_j)

           DFF.to_csv(folder+"\\DFF.csv")

           return DFF
```