Thermodynamic Modelling of Concentrating Solar Powered Supercritical CO2 Brayton Cycle Integrated with Thermal Storage

Lauren E. Rogers
University of Florida, Gainesville, FL 32601

11 December 2020

The objective of this thesis is to explore and demonstrate the compatibility of concentrating solar power (CSP), refractory brick thermal storage, and supercritical CO_2 Brayton cycles. Separately, these systems are rising in interest in the renewable energy industry as solutions to achieve high efficiency, dispatchable power outputs. In response to the U.S. Department of Energy 2030 roadmap for CSP, this work models a system pairing a molten lead (Pb) solar concentrator with brick thermal storage and a 1 MW sCO_2 Brayton cycle, achieving $<1\%$ power output variation across a 24-hour solar cycle. Thermal brick storage was sized to fully charge during a typical daytime cycle, and heat the sCO_2 during the night. The resultant thermal brick model was found to be more size efficient per kWh than both hydroelectric and battery storage, and 1/6th the cost of lithium-ion battery storage.

Contents

I. *Introduction*

As the globe grows warmer and carbon dioxide levels in the atmosphere increase annually, the need for clean energy production is exacerbated rapidly. One of the inhibitors of integration for solar power is the volatility in solar irradiance over time and location. The power output of the grid must meet the instantaneous demand of consumers, therefore requiring a predictable and controllable input to meet this demand. On the contrary, renewable energies such as wind and solar are dependent on natural phenomena that cannot be controlled. These renewable systems can be paired with batteries to provide control over power output across time, but batteries are significantly more expensive per kWh than conventional power generation such as natural gas or coal plants [1,2]. In order to be a viable option for power generation, energy systems must be competitively priced, dispatchable to meet demand, and scalable for large power outputs. This thesis investigates and proposes the design of a renewable energy system that seeks to meet these demands.

Motivation for this work comes from 2030 DOE Concentrating Solar Power roadmap, which aims to increase solar receiver output temperatures above 720 °C, integrate at least 12 hours of thermal storage, and specifies the sCO_2 Brayton cycle as best-fit for high efficiency systems [19,20]. Almost half of existing CSP utilizes thermal storage, and utilization in projects currently under construction surpasses 80%. Integration of thermal storage to CSP projects is desirable due to the provided improvements in dispatchability [3]. Due to the limited materials functional at high temperatures (>600 °C), pre-existing thermal storage studies largely focus on CSP cycles with molten salts as the working fluid [3]. Supercritical Brayton cycles are becoming more attractive due to recent demonstrations of efficiencies ranging from 50-60% [17], but such efficiencies require temperature inputs beyond 1000 °C. In anticipation of hardware compatibility for CSP applications at higher temperatures, this project models an introductory system to pair CSP with a supercritical CO2 Brayton Cycle and brick thermal energy storage.

II. *Analysis*

A) Overall System Model

The designed system is made up of 3 major subsystems: molten lead solar receiver, Brayton cycle, and brick thermal storage. During the day, molten lead flows through a loop where it is heated in the solar receiver, and transfers this heat to sCO_2 through a heat exchanger. This sCO_2 splits flow, with a fixed mass flow rate entering the Brayton cycle, and a variable mass flow rate through the thermal brick storage during the day to charge it. The mass flow rate entering the Brayton cycle must be constant to maintain a constant power output, but the flow rate going through the lead heat exchanger is variable due to variation in heat input from the sun throughout the day. At night, the fixed mass flow rate sCO_2 flows in the opposite direction through the thermal storage to be heated, and then enters the Brayton cycle. The sCO_2 will flow through the thermal storage as a bank of tubes, but will be modeled with a single tube due to assumption of radial uniformity.

Supercritical carbon dioxide (sCO_2) was selected as the working fluid due to its potential for high efficiency, minimized compressibility, and low volumetric flow rate when compared to other working fluids such as Helium for high-efficiency power cycles [17]. Beyond an expanded efficiency range, sCO_2 Brayton cycles have 1/10th the turbomachinery footprint than that of a steam Rankine cycle, significantly reducing hardware and maintenance costs [18].

Lead (Pb) was selected as the heat transfer fluid through the CSP receiver due to its stability at high temperatures, low oxidation rate, low cost, and extensive validation through nuclear applications [12].

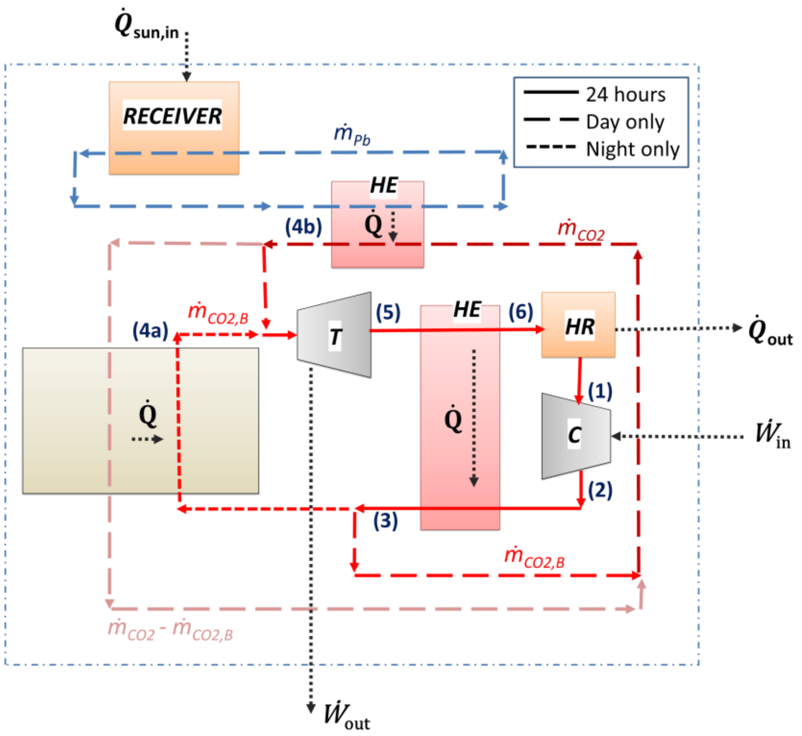

Figure 1: Complete system schematic

B) Generating Thermodynamics Properties of sCO_2

In order to model a Brayton cycle with supercritical carbon dioxide as the working fluid, thermodynamic properties first had to be generated for sCO2. Unlike more popular working fluids such as water and air, sCO_2 data is not easily accessible for computations. Span and Wagner developed an equation of state (EOS) valid for the supercritical region [16], which is utilized in the National Institute of Standards and Technology (NIST) database [4]. This data is only available through their web platform and has not been integrated into many open source or commercial software platforms due to the complexity of Span and Wagner's model [9]. Cantera, a tool that can be applied in Python, does offer thermodynamic property calculation, but does not utilize the Span and Wagner EOS. To test whether this could be utilized for modelling, thermodynamic property values were calculated using both Cantera and the NIST database, and then compared. As seen in Table 1, although Cantera reached accuracies as low as 0.14% for some data points, discrepancies up to 21% were found. Due to the reliability of the Span and Wagner EOS [9], the Cantera data was deemed unreliable for the high temperature and pressures used for high efficiency supercritical Brayton cycles.

TABLE I
Percent difference in parameters calculated by NIST and Cantera

Locations tested	500 K, 5 MPa to 700 K, 10 MPa	500 K, 10 MPa to 700 K, 25 MPa	700 K, 25 MPa to 500 K, 5 MPa
Enthalpy variation (%)	0.14	0.17	0.26
Internal energy variation (%)	0.17	0.21	0.27
Entropy variation (%)	3.89	11.89	21.08

Instead, a tool was generated in Python that utilizes the Span and Wagner EOS to calculate pressure P, entropy s, internal energy u, and enthalpy h of carbon dioxide. Utilizing (1)-(6), individual functions were generated to find each of these values as a function of temperature T and density ρ, as well as the critical temperature and density of CO_2 (T_c and ρ_c). All terms other than density and temperature are further sets of complex equations with 290 unique constants [16], but require no further inputs.

$$\delta = \frac{\rho}{\rho_c} \tag{1}$$

$$\tau = \frac{T_c}{T} \tag{2}$$

$$p(\delta,\tau) = \rho RT(1 + \delta \emptyset_\delta^\tau) \tag{3}$$

$$s\ (\delta,\tau) = \mathrm{R}[\tau(\emptyset_\tau^o + \emptyset_\tau^\tau) - \emptyset^o - \emptyset^\tau] \tag{4}$$

$$\mathrm{u}(\delta,\tau) = RT\tau(\emptyset_\tau^o + \emptyset_\tau^\tau) \tag{5}$$

$$\mathrm{h}(\delta,\tau) = RT[1 + \tau(\emptyset_\tau^o + \emptyset_\tau^\tau) + \delta\emptyset_\delta^\tau] \tag{6}$$

These functions are included in the Appendix. Often, the known parameters of the system are not temperature or density, so a larger function was made to solve for any thermodynamic parameter knowing any of the other two (*P, s, u, h, T, ρ*).

If one of the function parameters (density or temperature) is an input, the other value is calculated by an iterative solution procedure using SciPy's minimization function. An example is shown below to calculate the temperature when density and enthalpy are known. Once temperature is found, temperature and density can be plugged into any of the previously generated thermodynamic property functions to calculate the desired parameters.

```
def minenthalpy(x):
    return abs(enthalpy(rho, x) - h)
bnds = [(300, 1100)]
res = minimize(minenthalpy, [400], method='Nelder-Mead')
T = res.x[0]
```

When the inputs do not include temperature or density, the minimization must be conducted by choosing initial guesses for both parameters before minimizing. An example is shown below to calculate the density and temperature when pressure and enthalpy are known. Similar to the previous calculation, temperature and density can be plugged into any of the thermodynamic property functions after solving the minimization to generate the desired parameter.

```
def minPH(t):
    D, T = t
    return abs(pressure(D, T) - P) + 100 * abs(enthalpy(D, T) - H)
bnds = ((1, 1500), (1, 1100))
result = 10
while result > 1:
    guess = 20 * random.randint(1, 10), 100 * random.randint(1, 10)
    res = minimize(minPH, [guess], method='SLSQP', bounds=bnds, options={'maxiter': 6000,
'ftol': 1e-40})
    result = res.fun
D = res.x[0]
T = res.x[1]
```

The final result is a single function that can be implemented into any python code to calculate thermodynamic properties of carbon dioxide at temperatures up to 1100 K and pressures up to 800 MPa [16]. This function and the individual thermodynamic property functions it uses are included in the Appendix. When users have all functions saved to their computer, the CO2properties function can be used to generate any of the discussed properties accurate to the NIST database [4].

C) *Modelling the sCO_2 Brayton Cycle in Python*

Once thermodynamic properties could be readily generated for any temperature and pressure range, a model for the sCO_2 cycle could then be designed. For simplicity, a single flow cycle with recuperation was selected. For future applications, more complex models can be explored to optimize cycle efficiency, such as the compression cycle which has been found to generate efficiencies above 43% [12].

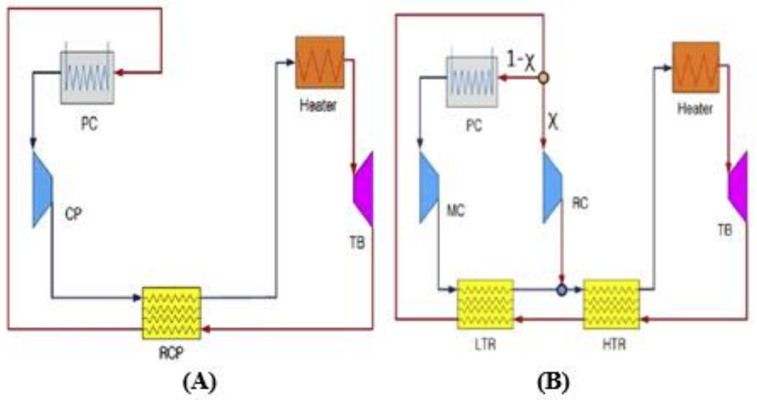

Figure 2: (a) Brayton cycle with recuperation (b) Brayton cycle with recompression. Figure from ref. [11]

Further models of sCO2 Brayton cycles have uncovered the potential to approach 60% [12] at high temperature and pressure, while steam Rankine cycles reach a maximum efficiency of 43% [13,14]. Models of these cycles have not been prevalently studied due to challenges with materials at high temperatures and pressures, further motivating the modeling and optimization of cycles through software to demonstrate the value and efficacy of further research.

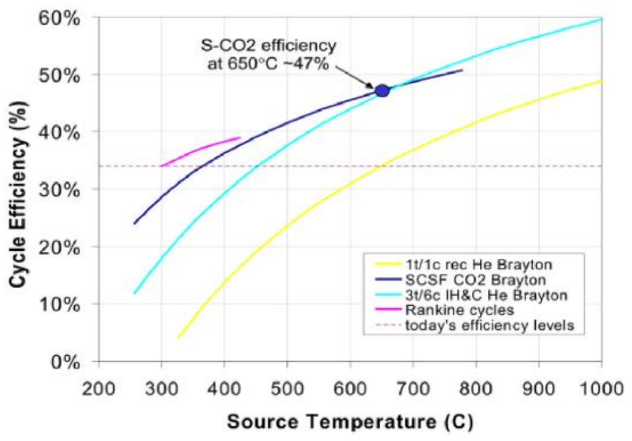

Figure 3: Rankine vs supercritical Brayton cycle efficiencies. Fig. from ref. [13]

To confirm the accuracy of the generated recuperation model, it was designed utilizing an existing study's parameters [11], and then scaled to produce the desired power output. Power output of 1 MW was modelled to represent a general test scale model. Pressure losses through the heat exchanger were assumed to be negligible.

TABLE II
Brayton Cycle Parameter inputs, from ref. [11]

Component	Parameter	Value
Turbine	$T_{t,i}$ Inlet temperature [K]	773
	$P_{t,i}$ Inlet pressure [MPa]	25
	η_t Isentropic efficiency [%]	92
Compressor	$T_{c,i}$ Inlet temperature [K]	305
	$P_{c,i}$ Inlet pressure [MPa]	7.5
	η_c Isentropic efficiency [%]	88
Recuperator	ε Effectiveness [%]	95

The resultant efficiency of the Python model with the provided inputs was 38%, matching the reference study [11] and therefore confirming accuracy of the model. The T-S diagram for the system is shown in Fig. 4, with the positions correlating to the overall system model of Fig. 1. Mass flow rate of sCO_2 for a 1 MW output, represented as $\dot{m}_{CO2,B}$ in Fig 1., was found to be 6.82 kg/s.

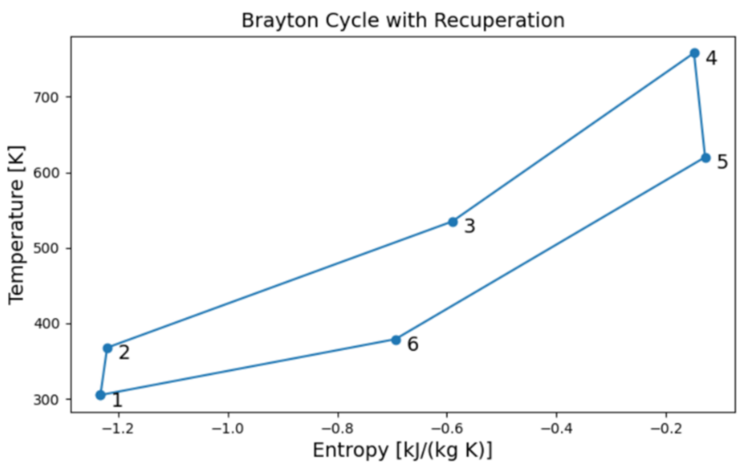

Figure 4: T-S diagram of modelled Brayton Cycle with recuperation

D) *Solar Concentrator Analysis*

To provide a constant power output out of the turbine, the temperature and mass flow rate of sCO_2 entering it must be constant. During the day, the sCO_2 is heated by molten lead flowing through a cross-flow heat exchanger.

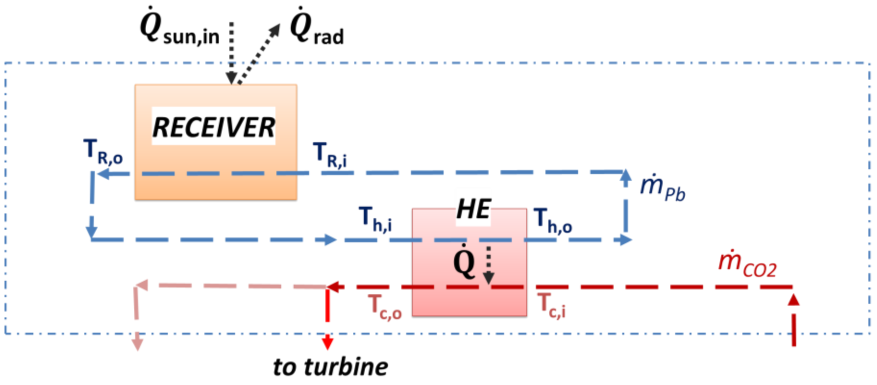

Figure 5: Solar receiver and heat exchanger representation

The effectiveness of this heat exchanger is represented by (7), with the cold side denoting the sCO_2 flow, and the hot side denoting the lead flow. The outlet of the heat exchanger, $T_{c,o}$, is the inlet temperature of the turbine and is therefore known. The inlet comes from the Brayton cycle heat exchanger, so $T_{c,i}$ is equivalent to T_3 from the T-S diagram (Fig. 4). Conservatively approximating the effectiveness of the heat exchanger to be 0.90, the temperature of lead exiting the heat exchanger can be found.

$$\varepsilon = \frac{h_{c,o} - h_{c,i}}{h(T_{h,i}) - h_{c,i}} \tag{7}$$

The absorption efficiency of the solar concentrator is dependent on the irradiance of the sun, and therefore varies throughout the day. The temperature previously found entering the heat exchanger is equivalent to the exit temperature of the receiver. Using this, irradiance data across time [11], a concentration ratio of 2000, and the Boltzmann constant, the absorber efficiency is calculated as a function of time. Results show an efficiency ranging from 91.2 - 97.3% for the 1 MW plant.

$$\eta_{abs} = 1 - \frac{\sigma T_{r,o}^4}{G_{in} C} \tag{8}$$

The power absorbed P_{abs} can then be calculated, with the aperture sized to generate a maximum power at the peak irradiance of the day. For the 1 MW plant, this max power was selected to be 10.5 MW, resulting in an aperture of 1.51 m radius. The absorbed power is found in Fig. 6., with a cubic interpolation to approximate the data at 1-minute intervals.

$$P_{abs} = G_{in} \cdot A \cdot \eta_{abs} \tag{9}$$

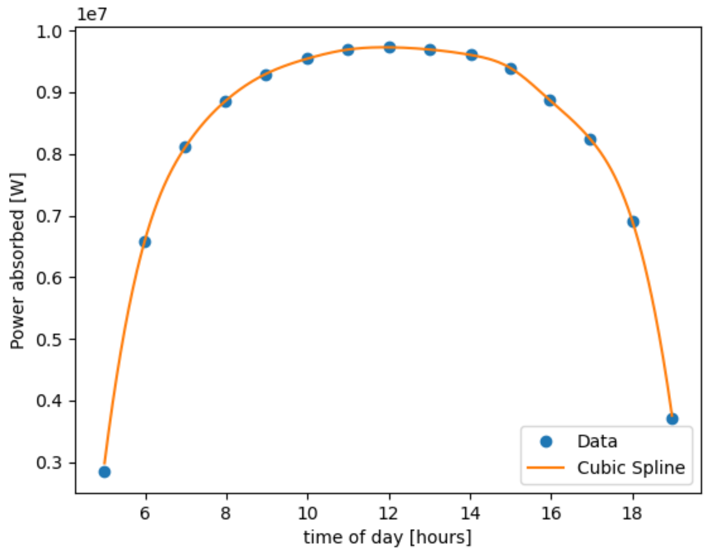

Figure 6: Data and interpolation fit of power absorbed across time

Conducting energy balances, this power is found equal to the energy absorbed by the molten lead, which is equal to the power transferred to the sCO2 (10-11).

$$P_{abs} = \dot{m_{pb}}(h_{r,o} - h_{r,i}) \tag{10}$$

$$\dot{m_{pb}}(h_{r,o} - h_{r,i}) = \dot{m_{CO2,HE}}(h_{c,o} - h_{c,i}) \tag{11}$$

These relations show that mass flow rate of lead and sCO_2 must vary if the temperatures at each position are fixed. The system was designed with the lead outlet temperature $T_{r,i}$ at 605 K to keep lead in the fluid range. With these parameters known, mass flow rate of lead and carbon dioxide across time are found.

E) Thermal Storage Sizing

At night, $_sCO2$ is heated by the thermal storage of the lead heat exchanger. To maintain the same temperature input into the Brayton cycle, the inlet of the brick must be equivalent to T_3, and the outlet equivalent to $T_{t,i}$. Assuming a perfectly insulated system, the energy transferred from the brick to CO2 is due to convection. Temperature distribution through the thermal storage is dependent on both time and position. Isolating one *dz* increment of the thermal storage and sCO_2, the energy balances are (13-15).

$$\frac{dU}{dt}_{CO2} = \dot{Q}_{conv} + \dot{m}_{CO2,B}(h_i - h_e) \tag{13}$$

$$\frac{dU}{dt}_{TS} = \dot{Q}_{cond} - \dot{Q}_{conv} \tag{14}$$

$$\dot{Q}_{cond} = -kA \cdot \frac{dT}{dz} \tag{15}$$

By setting Q_{conv} of each equal, the temperature distribution through the thermal storage was calculated iteratively. This distribution was represented by a 2D matrix, with one dimension representing time increments (*i*), and the other position increments (*j*).

$$\dot{m}_{CO2,B}\left[(h_{i,j} - h_{i-1,j}) + (u_{i,j} - u_{i,j-1})\right] = \frac{(\rho \cdot A \cdot dz)C_{v,TS}}{\Delta t}(T_{i,j} - T_{i,j-1}) + \frac{kA}{\Delta z}(h_{i,j} - h_{i-1,j}) \tag{16}$$

Two boundary conditions were set: the temperature at the beginning of the night throughout the whole receiver is the desired outlet temperature, $T_{t,i}$. Secondly, the base of the thermal storage is assumed equal to the inlet temperature at all times, which is constant: T_3.

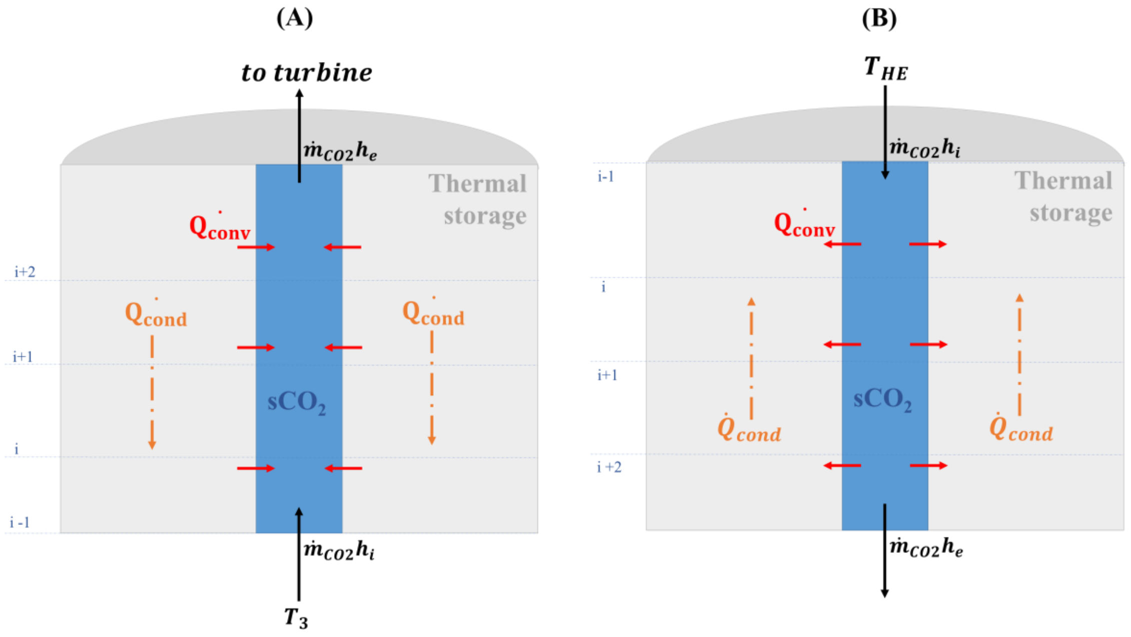

Figure 7: (A) brick storage discharging heat transfer during night (B) brick storage charging heat transfer during day

Temperature is solved at each position increment for each time interval, and the resultant temperature distribution is shown in Fig. 8. Across 11 hours, the temperature at the exit stays within 5 K of the desired turbine inlet temperature, maintaining the desired power output within 7.65 kW for the 1 MW system, or within 0.77%.

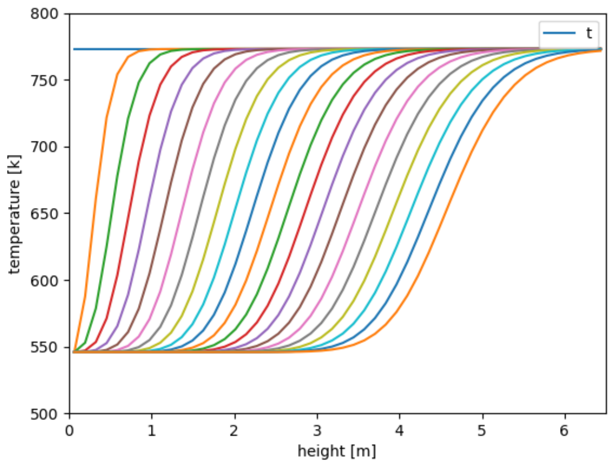

Figure 8: Temperature distribution of thermal brick at 30-minute time intervals from 7 pm to 5 am. Time intervals increase from left to right.

To analyze the charging of the thermal storage during the day, the same procedure is followed. Carbon dioxide flow during the day enters hot from the Pb heat exchanger at the top of the thermal storage, and exits to be reheated by the heat exchanger once again (Fig. 7). The direction of convection and conduction is flipped from the nighttime, as seen in (18-20).

$$\frac{dU}{dt}_{CO2} = -\dot{Q}_{conv} + \dot{m}_{CO2,day}(h_i - h_e) \tag{17}$$

$$\frac{dU}{dt}_{TS} = -\dot{Q}_{cond} + \dot{Q}_{conv} \tag{18}$$

$$\dot{Q}_{cond} = -kA \cdot \frac{dT}{dz} \tag{19}$$

For this application, the mass flow rate through the brick was the difference between the calculated mass flow rate out of the heat exchanger and the constant mass flow rate through the Brayton cycle.

$$\dot{m}_{CO2,day} = \dot{m}_{CO2} - \dot{m}_{CO2,B} \tag{20}$$

The temperature distribution for the initial time interval of the day is defined as equal to the distribution at the final time interval of night. The inlet temperature is set equal to the heat exchanger outlet temperature, which is equal to T_{ti}. A 2D temperature matrix is once again set up to represent the temperature distribution across time and position. With the defined boundary conditions, temperature values at each position interval in each time interval are solved using (21).

$$\begin{aligned}(\dot{m}_{CO2,day}\left[\left(h_{i-1,j} - h_{i,j}\right) + \left(u_{i,j-1} - u_{i,j}\right)\right] \\ = \frac{(\rho \cdot A \cdot dz)C_{v,TS}}{\Delta t}\left(T_{i,j} - T_{i,j-1}\right) - \frac{kA}{\Delta z}\left(h_{i,j} - h_{i-1,j}\right)\end{aligned} \tag{21}$$

Starting with an initial outlet temperature of T_3 from the nighttime discharge cycle, it takes 11.98 hours out of the 13 daytime hours to heat the thermal brick to be uniformly at T_{ti}. This effective charging of the thermal brick within one daytime cycle validates the aperture selection and resultant sCO_2 mass flow rate from the CSP side analysis.

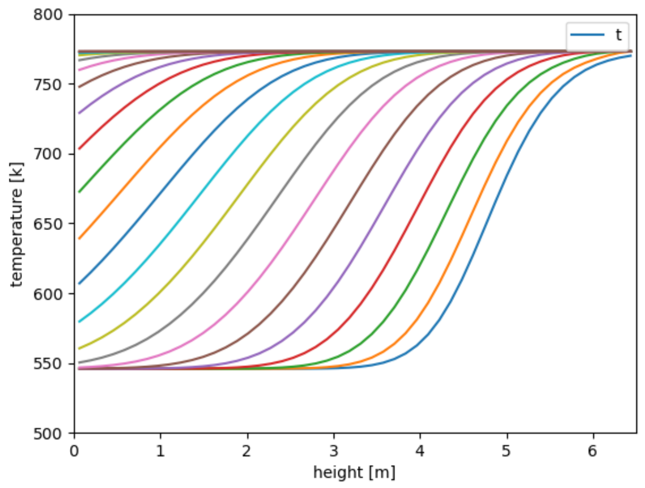

Figure 9: Temperature distribution of thermal brick at 30-minute time intervals from 5 am to 7 pm. Time intervals increase from right to left.

The brick thermal storage was sized to heat sCO2 within 5 K of the desired outlet temperature so that the power output at night varied by less than 1%. This result was dependent on the density and heat capacity of the selected thermal brick material. Influences on material properties on system size were demonstrated by sizing the system with 2 different thermal bricks: JM-23 Firebrick [5] and KS-4V PLUS castable refractory [6].

TABLE III
Influence of material properties on necessary thermal storage volume

Material	JM-30	KS-4V PLUS
Density [kg/m^3]	1020	2050
Heat capacity [kJ/kg*K]	1.1	1.187
Volume needed [m^3]	384.0	183.7

To extend nighttime storage to 12 hours to meet the DOE 2030 roadmap milestone [20], the thermal brick volume must be increased from 183.7 m^3 to 194.5 m^3.

F) *Cost and Size Analysis*

For one daytime cycle, the energy stored into the thermal brick is 21.4 MWh. To evaluate the effectiveness of this thermal storage method, the cost and size per kWh is compared to other competitive energy storage devices: battery and hydroelectric storage. Two lithium-ion battery metrics are used: the Tesla Powerpack [7] and data from the 2018 NREL Energy Storage Benchmark [15]. Hydroelectric storage is represented by data on the Hoover Dam [8]. As seen in Table 4, although the volume needed for the thermal brick appears significant, it is more space effective than both hydroelectric and battery storage. Cost for the thermal storage was quantified by material and installation costs provided by the manufacturer and scaled [6]. For consistency, the Hoover Dam cost only accounts for the construction and materials of the dam itself, not the supporting power infrastructure.

TABLE IV
Cost and volume comparison of energy storage solutions

Storage Source	KS-4V PLUS Refractory	Hoover Dam	Tesla Powerpack	NREL Lithium-ion battery
Cost per kWh [$/kWh/cycle]	30.27	4.26	504	209
Volume per kWh [m^3/kWh]	0.0086	0.292	0.0097	0.015

For the refractory material used in this application, the overall cost was less than 1/6th that of the most affordable battery metrics. This indicates that thermal brick storage may be a competitive option in cases where batteries are typically considered. The cost of hydroelectric power remains 7x more affordable than this design, which indicates that it is still the more desirable application when available. Costs of this system may be optimized further by pursuing low-cost refractory materials with high density, heat capacity, and temperature stability. Additionally, installation comprised 32% of the overall cost. Therefore, any optimizations to the construction process will also significantly impact the price per kWh.

G) *Overall System Efficiency*

Efficiency of the thermal storage system was found through the ratio of energy transferred to the sCO_2 at night to the energy used to charge the thermal storage during the daytime (22). This was calculated to be 98.4%.

$$\eta_{TS} = \frac{Q_{discharge}}{Q_{charging}} \times 100\% \tag{22}$$

As discussed previously, the absorption efficiency of the solar receiver varies with solar intensity over time (Fig. 10.), with a mean efficiency of 96.3%. The efficiency of the Brayton cycle modelled was 38%.

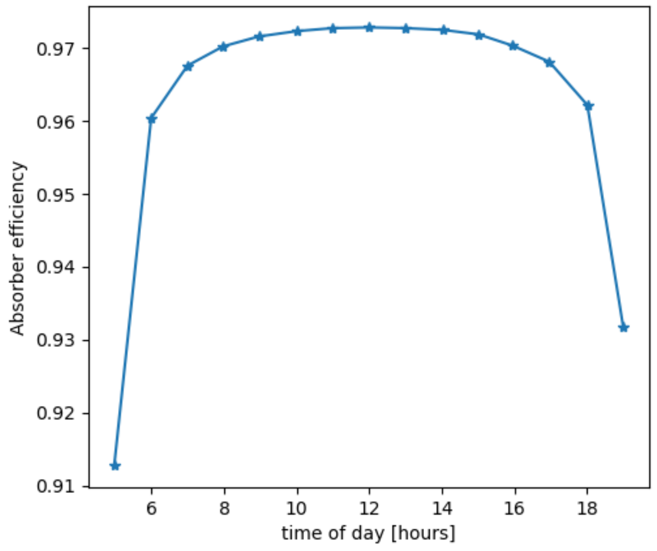

Figure 10: Absorber efficiency variation across time

Multiplying these components, the overall efficiency was found to be 36%. This efficiency accounts for the systems sized in this model, so it does not factor in optical losses from the solar field. Table 5 tabulates the efficiency of competitive power generation cycles for comparison. Currently, this model surpasses photovoltaic efficiencies and is approaching competitiveness with coal power. As discussed previously, sCO_2

Brayton cycles have the potential to exceed 50-60% at high temperatures and pressures [18]. If this is the case, the overall efficiency of this system could reach 47-56%, surpassing natural gas combined cycle (NGCC) efficiencies.

TABLE V
Efficiencies of power generation systems

Power System	Overall Efficiency
Thesis Model	36%
Photovoltaics	21%
Coal Plant	39%
NGCC with CO2 capture	43%
NGCC with no CO2 capture	50%

III. *Conclusion*

This work has demonstrated the feasibility and compatibility of using a molten lead solar receiver paired with refractory brick thermal storage to continuously power a sCO_2 Brayton cycle. A 1 MW output was shown to be maintained within <1% variation over a 24-hour period despite variability in solar input and a heat source transition. Overall system efficiency for a simple Brayton cycle was 36%, with the potential to achieve higher efficiencies approaching 50% if more advanced models with higher temperatures and recompression are used. Further studies are required to quantify the impact using this cycle would have on the overall system, so size, cost, and efficiency tradeoffs can be measured. The brick thermal storage was sized to be more space efficient than competitive energy storage systems while storing enough heat to maintain a constant temperature output within 5 K for 11 hours of no solar input. Depending on precision and safety factor desires of future applications, this system can be expanded to anticipate longer durations without solar input by increasing volume of the thermal storage and diameter of the CSP aperture, or allowing more variability in power output.

While the thermal storage modelled was more space efficient than both hydroelectric power and battery storage, it costs more than hydroelectric power per kWh while being significantly cheaper than battery storage. Only 2 materials were studied in this work, so further cost optimization can be achieved by inspecting low-cost material candidates with high heat capacity, density, and temperature stability.

Finally, a by-product of this project's analysis was the creation of an open-source Python function to generate thermodynamic properties of supercritical CO_2 to the same accuracy as the NIST database. Through creation of this function, future researchers can have an easily integrated tool to analytically study sCO_2 models and their thermodynamic properties.

This body of work was an exploration of a concept, and therefore has not optimized any one system. To further demonstrate the dispatchability and potential for high efficiency of this system, further work is recommended. The thermal storage was sized based on a single day's solar data, so further analysis should be done to study the system size and efficiency required for various times of year and weather conditions. Additionally, the sCO_2 was modelled as a single tube coming through the brick, and radial temperature variations were neglected. To make this assumption valid, analysis must be conducted to find the optimal size and spacing of a bank of tubes throughout the thermal storage.

IV. *Acknowledgements*

I would like to give thanks to Dr. Jonathan Scheffe for his tremendous contributions of support, guidance, patience, and knowledge throughout the entirety of my research project. Further thanks to Dr. Trainham and the rest of the Renewable Energy Conversion Laboratory group, as well as UF's Center for Undergraduate Research Board of Students.

V. References

[1] M. A. Green, "Commercial progress and challenges for photovoltaics," *Nature Energy,* 2016.

[2] "COST AND PERFORMANCE BASELINE FOR FOSSIL ENERGY PLANTS VOLUME 1: BITUMINOUS COAL AND NATURAL GAS TO ELECTRICITY," National Energy Technology Laboratory, 2010.

[3] "Thermophysical Properties of Fluid Systems," National Institute of Standards and Technology , 2018. [Online]. Available: https://webbook.nist.gov/chemistry/fluid/.

[4] "Thermal Ceramics Product Data Book," Morgan Advances Materials.

[5] "KS-4V PLUS Product Data," HarbisonWalker International.

[6] "Powerpack," Tesla, 2020. [Online]. Available: tesla.com/powerpack.

[7] "Hoover Dam," Bechtel, 2020. [Online]. Available: https://www.bechtel.com/projects/hoover-dam/.

[8] E. Heidaryan and A. Jarrahian, "Modified Redlich Kwong equation of state for supercritical carbon dioxide," *the Journal of Supercritical Fluids,* vol. 81, pp. 92-98, 2013.

[9] J. Pacio and T. Wetzel, "Assessment of liquid metal technology status and research paths," *Solar Energy,* vol. 93, pp. 11-22, 2013.

[10] J. Sarwar, G. Georgakis, K. Kouloulias and K. Kakosimos, "Experimental and numerical investigation of the aperture size effect on the efficient solar energy harvesting for solar thermochemical applications," *Energy Conversion and Management,* vol. 92, pp. 331-341, 2015.

[11] Y. AHN and S. JUN BAE, "Review of supercritical CO2 power cycle technology and current status of research and development," *Nuclear Engineering and Technology,* 2015.

[12] V. Cheang, R. Hedderwick and C. McGregor, "Benchmarking supercritical carbon dioxide cycles against steam Rankine cycles for Concentrated Solar Power," *Solar Energy,* vol. 113, pp. 199-211, 2015.

[13] Y. Liu, Y. Wang and D. Huang, "Supercritical CO2 Brayton cycle: A state-of-the-art review," *Energy,* 2019.

[14] R. Fu, T. Remo and R. Margolis, "2018 U.S. Utility-Scale PhotovoltaicsPlus-Energy Storage System Costs," National Renewable Energy Laboratory, 2018.

[15] R. Span and W. Wagner, "A New Equation of State for Carbon Dioxide Covering the Fluid Region from the Triple-Point Temperature to 1100 K at Pressures up to 800 MPa," *Journal of Physical and Chemical Reference Data,* 1996.

[16] ZXiaoyi Chen, Z. Zhang and e. al., "State of the art on the high-temperature thermochemical energy storage systems," *Energy Conversion and Management,* vol. 177, pp. 792-815, 2018.

[17] M. T. Dunham and B. D. Iverson, "High-efficiency thermodynamic power cycles for concentrated solar power systems," *Renewable and Sustainable Energy Reviews,* 2013.

[18] M. Liu, N. S. Tay and e. al., "Review on concentrating solar power plants and new developments in high temperature thermal energy storage technologies," *Renewable and Sustainable Energy Reviews,* vol. 53, pp. 1411-1432, 2016.

[19] "Department of Energy Announces $125.5 Million in New Funding for Solar Technologies," Department of Energy, 5 February 2020. [Online]. Available: https://www.energy.gov/articles/department-energy-announces-1255-million-new-funding-solar-technologies.

[20] National Renewable Energy Laboratory, "Concentrating Solar Power Gen3 Demonstration Roadmap," U.S. Department of Energy.

VI. *Appendix: Python Functions*

Complete set of Python scripts to generate internal energy, entropy, enthalpy, and pressure of supercritical CO_2. To use, all 5 functions must be named and saved into the same folder. Then CO2properties.py can be called as a function in any code desired to calculate temperature, density, internal energy, enthalpy, entropy, or pressure given any 2 of those values.

```
CO2properties(prop1,prop2,v1,v2,out)
```

prop1 and *prop2* are the input property variables (‘T’, ‘D’,’U’,’H’,’S’,’P’), *v1* and *v2* are the respective numeric values of these properties, and *out* is the desired output (‘T’, ‘D’,’U’,’H’,’S’,’P’).

A) Internal Energy Function (int_energy.py)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
import thermopy
from thermopy import iapws

##tabulated values
T27 = [[] for _ in range(3)]
for i in range(1, 9):
    T27[0].append(i)  # i
T27[1] = [8.37304456, -3.70454304, 2.5, 1.99427042, .62105248, .41195293, 1.04028922, .08327678]  # a^o
T27[2] = [0, 0, 0, 3.15163, 6.1119, 6.77708, 11.32384, 27.08792]  # theta^o

T31a = [[] for _ in range(4)]
for i in range(1, 8):
    T31a[0].append(i)  # i
T31a[1] = [.38856823203161, .29385475942740e1, -.55867188534934e1, -.76753199592477, .31729005580416,
.54803315897767,
           .12279411220335]  # n
T31a[2] = [1, 1, 1, 1, 2, 2, 3]  # d
T31a[3] = [0, .75, 1, 2, .75, 2, .75]  # t

T31b = [[] for _ in range(5)]
for i in range(8, 35):
    T31b[0].append(i)  # i
T31b[1] = [.21658961543220e1, .15841735109724e1, -.23132705405503, .58116916431436e-1, -.55369137205382,
.48946615909422,
           -.24275739843501e-1,
           .62494790501678e-1, -.12175860225246, -.37055685270086, -.16775879700426e-1, -.11960736637987,
           -.45619362508778e-1, .35612789270346e-1, -.74427727132052e-2, -.17395704902432e-2,
           -.21810121289527e-1, .24332166559236e-1, -.37440133423463e-1, .14338715756878, -.13491969083286,
           -.23151225053480e-1, .12363125492901e-1,
           .21058321972940e-2, -.33958519026368e-3, .55993651771592e-2, -.30335118055646e-3]  # n
T31b[2] = [1., 2., 4., 5., 5., 5., 6., 6., 6., 1., 1., 4., 4., 4., 7., 8., 2., 3., 3., 5., 5., 6., 7., 8., 10., 4.,
           8.]  # d
T31b[3] = [1.5, 1.5, 2.5, 0., 1.5, 2., 0., 1., 2., 3., 6., 3., 6., 8., 6., 0., 7., 12., 16., 22., 24., 16., 24., 8.,
2.,
           28., 14.]  # t
T31b[4] = [1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2., 3., 3., 3., 4., 4., 4., 4., 4., 4., 5.,
           6.]  # c

T31c = [[] for _ in range(8)]
for i in range(35, 40):
    T31c[0].append(i)  # i
T31c[1] = [-.21365488688320e3, .26641569149272e5, -.24027212204557e5, -.28341603423999e3, .21247284400179e3]  # n
T31c[2] = [2, 2, 2, 3, 3]  # d
T31c[3] = [1, 0, 1, 3, 3]  # t
T31c[4] = [25, 25, 25, 15, 20]  # alpha
T31c[5] = [325, 300, 300, 275, 275]  # beta
```

```
T31c[6] = [1.16, 1.19, 1.19, 1.25, 1.22]  # lambda
T31c[7] = [1, 1, 1, 1, 1]  # eps

T31d = [[] for _ in range(9)]
for i in range(40, 43):
    T31d[0].append(i)  # i
T31d[1] = [-.66642276540751, .72608632349897, .55068668612842e-1]  # n
T31d[2] = [3.5, 3.5, 3.]  # a
T31d[3] = [.875, .925, .875]  # b
T31d[4] = [.3, .3, .3]  # beta
T31d[5] = [.7, .7, .7]  # A
T31d[6] = [.3, .3, 1.]  # B
T31d[7] = [10, 10, 12.5]  # C
T31d[8] = [275, 275, 275]  # D


def int_energy(density, temp):
    T_c = 304.128  # critical temperature, degrees kelvin
    rho_c = 467.6  # critical density, kg/m^3
    R = .1889241  # specific gas constant, kJ/(kg*K)

    tau = T_c / temp
    delt = density / rho_c

    ##inputs
    # from table 32
    def delta(delt, tau, i):
        return (theta(delt, tau, i)) ** 2 + T31d[6][i - 40] * ((delt - 1) ** 2) ** T31d[2][i - 40]

    def psi(delt, tau, i):
        return np.e ** (-T31d[7][i - 40] * (delt - 1) ** 2 - T31d[8][i - 40] * (tau - 1) ** 2)

    def theta(delt, tau, i):
        return (1 - tau) + T31d[5][i - 40] * ((delt - 1) ** 2) ** (1 / (2 * T31d[4][i - 40]))

    ##inputs
    # from table 32
    def ddeltab_dtau(delt, tau, i):
        return -2 * theta(delt, tau, i) * T31d[3][i - 40] * delta(delt, tau, i) ** (T31d[3][i - 40] - 1)

    def dpsi_dtau(delt, tau, i):
        return -2 * T31d[8][i - 40] * (tau - 1) * psi(delt, tau, i)

    ##functions
    def phi_ot(delt, tau):  # Table 28, row 1
        sum2 = 0
        for i in range(4, 9):
            sum2 = sum2 + T27[1][i - 1] * T27[2][i - 1] * (((1 - np.e ** (-T27[2][i - 1] * tau)) ** -1) - 1)
        return T27[1][1] + T27[1][2] / tau + sum2

    def phi_tt(delt, tau):  # Table 32, eqn 4
        suma = 0
        sumb = 0
        sumc = 0
        sumd = 0
        for i in range(1, 8):
            suma = suma + T31a[1][i - 1] * T31a[3][i - 1] * (delt ** T31a[2][i - 1]) * (
                    tau ** (T31a[3][i - 1] - 1))
        for i in range(8, 35):
            sumb = sumb + T31b[1][i - 8] * T31b[3][i - 8] * (delt ** T31b[2][i - 8]) * (
                    tau ** (T31b[3][i - 8] - 1)) * np.e ** (-delt ** T31b[4][i - 8])
        for i in range(35, 40):
            sumc = sumc + T31c[1][i - 35] * (delt ** T31c[2][i - 35]) * (tau ** T31c[3][i - 35]) * (
                    np.e ** (-T31c[4][i - 35] *
                             (delt - T31c[7][i - 35]) ** 2 - T31c[5][i - 35] * (tau - T31c[6][i - 35]) ** 2)) * (
                           T31c[3][i - 35] / tau - 2 * T31c[5][i - 35] * (tau - T31c[6][i - 35]))
        for i in range(40, 43):
            sumd = sumd + T31d[1][i - 40] * delt * (ddeltab_dtau(delt, tau, i) * psi(delt, tau, i) + (
                        delta(delt, tau, i) ** T31d[3][i - 40]) * dpsi_dtau(delt, tau, i))
        return suma + sumb + sumc + sumd

    return R * temp * (tau * (phi_ot(delt, tau) + phi_tt(delt, tau)))  # kJ/kg, equation from table 3
```

B) Enthalpy Function (enthalpy.py)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
import thermopy
from thermopy import iapws

##tabulated values
T27 = [[] for _ in range(3)]
for i in range(1, 9):
    T27[0].append(i)  # i
T27[1] = [8.37304456, -3.70454304, 2.5, 1.99427042, .62105248, .41195293, 1.04028922, .08327678]  # a^o
T27[2] = [0, 0, 0, 3.15163, 6.1119, 6.77708, 11.32384, 27.08792]  # theta^o
# print(T27)

T31a = [[] for _ in range(4)]
for i in range(1, 8):
    T31a[0].append(i)  # i
T31a[1] = [.38856823203161, .29385475942740e1, -.55867188534934e1, -.76753199592477, .31729005580416,
.54803315897767,
           .12279411220335]  # n
T31a[2] = [1, 1, 1, 1, 2, 2, 3]  # d
T31a[3] = [0, .75, 1, 2, .75, 2, .75]  # t
# print(T31a)

T31b = [[] for _ in range(5)]
for i in range(8, 35):
    T31b[0].append(i)  # i
T31b[1] = [.21658961543220e1, .15841735109724e1, -.23132705405503, .58116916431436e-1, -.55369137205382,
.48946615909422,
           -.24275739843501e-1,
           .62494790501678e-1, -.12175860225246, -.37055685270086, -.16775879700426e-1, -.11960736637987,
           -.45619362508778e-1, .35612789270346e-1, -.74427727132052e-2, -.17395704902432e-2,
           -.21810121289527e-1, .24332166559236e-1, -.37440133423463e-1, .14338715756878, -.13491969083286,
           -.23151225053480e-1, .12363125492901e-1,
           .21058321972940e-2, -.33958519026368e-3, .55993651771592e-2, -.30335118055646e-3]  # n
T31b[2] = [1., 2., 4., 5., 5., 5., 6., 6., 6., 1., 1., 4., 4., 4., 7., 8., 2., 3., 3., 5., 5., 6., 7., 8., 10., 4.,
           8.]  # d
T31b[3] = [1.5, 1.5, 2.5, 0., 1.5, 2., 0., 1., 2., 3., 6., 3., 6., 8., 6., 0., 7., 12., 16., 22., 24., 16., 24., 8.,
2.,
           28., 14.]  # t
T31b[4] = [1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2., 3., 3., 3., 4., 4., 4., 4., 4., 4., 5.,
           6.]  # c

T31c = [[] for _ in range(8)]
for i in range(35, 40):
    T31c[0].append(i)  # i
T31c[1] = [-.21365488688320e3, .26641569149272e5, -.24027212204557e5, -.28341603423999e3, .21247284400179e3]  # n
T31c[2] = [2, 2, 2, 3, 3]  # d
T31c[3] = [1, 0, 1, 3, 3]  # t
T31c[4] = [25, 25, 25, 15, 20]  # alpha
T31c[5] = [325, 300, 300, 275, 275]  # beta
T31c[6] = [1.16, 1.19, 1.19, 1.25, 1.22]  # lambda
T31c[7] = [1, 1, 1, 1, 1]  # eps

T31d = [[] for _ in range(9)]
for i in range(40, 43):
    T31d[0].append(i)  # i
T31d[1] = [-.66642276540751, .72608632349897, .55068668612842e-1]  # n
T31d[2] = [3.5, 3.5, 3.]  # a
T31d[3] = [.875, .925, .875]  # b
T31d[4] = [.3, .3, .3]  # beta
T31d[5] = [.7, .7, .7]  # A
T31d[6] = [.3, .3, 1.]  # B
T31d[7] = [10, 10, 12.5]  # C
T31d[8] = [275, 275, 275]  # D


def enthalpy(rho, T):

    R = .1889241  # specific gas constamt, kJ/(kg*K)
    T_c = 304.128  # critical temperature, kelvin
    tau = T_c / T
    rho_c = 467.6  # critical density, kg/m^3
    delt = rho / rho_c

    ##inputs
    # from table 32
    def ddeltab_dtau(delt, tau, i):
        return -2 * theta(delt, tau, i) * T31d[3][i - 40] * delta(delt, tau, i) ** (T31d[3][i - 40] - 1)

    def dpsi_dtau(delt, tau, i):
        return -2 * T31d[8][i - 40] * (tau - 1) * psi(delt, tau, i)

    ##functions
    def phi_ot(delt, tau):  # Table 28, row 5
        sum2 = 0
        for i in range(4, 9):
            sum2 = sum2 + T27[1][i - 1] * T27[2][i - 1] * (((1 - np.e ** (-T27[2][i - 1] * tau)) ** -1) - 1)
        return T27[1][1] + T27[1][2] / tau + sum2

    def phi_tt(delt, tau):
        suma = 0
        sumb = 0
        sumc = 0
        sumd = 0
        for i in range(1, 8):
            suma = suma + T31a[1][i - 1] * T31a[3][i - 1] * (delt ** T31a[2][i - 1]) * (
                    tau ** (T31a[3][i - 1] - 1))  ##FIX IN ALL CODES
        for i in range(8, 35):
            sumb = sumb + T31b[1][i - 8] * T31b[3][i - 8] * (delt ** T31b[2][i - 8]) * (
                    tau ** (T31b[3][i - 8] - 1)) * np.e ** (-delt ** T31b[4][i - 8])
        for i in range(35, 40):
            sumc = sumc + T31c[1][i - 35] * (delt ** T31c[2][i - 35]) * (tau ** T31c[3][i - 35]) * (
                    np.e ** (-T31c[4][i - 35] *
                             (delt - T31c[7][i - 35]) ** 2 - T31c[5][i - 35] * (tau - T31c[6][i - 35]) ** 2)) * (
                           T31c[3][i - 35] / tau - 2 * T31c[5][i - 35] * (tau - T31c[6][i - 35]))
        for i in range(40, 43):
            sumd = sumd + T31d[1][i - 40] * delt * (ddeltab_dtau(delt, tau, i) * psi(delt, tau, i) + (
                    delta(delt, tau, i) ** T31d[3][i - 40]) * dpsi_dtau(delt, tau, i))
        return suma + sumb + sumc + sumd

    ##inputs
    # from table 32
    def delta(delt, tau, i):
        return (theta(delt, tau, i)) ** 2 + T31d[6][i - 40] * ((delt - 1) ** 2) ** T31d[2][i - 40]

    def psi(delt, tau, i):
        return np.e ** (-T31d[7][i - 40] * (delt - 1) ** 2 - T31d[8][i - 40] * (tau - 1) ** 2)

    def theta(delt, tau, i):
        return (1 - tau) + T31d[5][i - 40] * ((delt - 1) ** 2) ** (1 / (2 * T31d[4][i - 40]))

    ##inputs
    # from table 32
    def dpsi_ddelt(delt, tau, i):
        return -2 * T31d[7][i - 40] * (delt - 1) * psi(delt, tau, i)

    def ddelta_ddelt(delt, tau, i):
        return (delt - 1) * (T31d[5][i - 40] * theta(delt, tau, i) * (
                2 / T31d[4][i - 40]) * ((delt - 1) ** 2) ** (-1 + 1 / (2 * T31d[4][i - 40])) + 2 * T31d[6][i -
40] *
                T31d[2][i - 40] * ((delt - 1) ** 2) ** (T31d[2][i - 40] - 1))

    def ddeltab_ddelt(delt, tau, i):
        return T31d[3][i - 40] * (delta(delt, tau, i) ** (T31d[3][i - 40] - 1)) * ddelta_ddelt(delt, tau, i)

    ##function
    def phi_td(delt, tau):  # Table 32, eqn 2
        suma = 0
        sumb = 0
        sumc = 0
        sumd = 0
        for i in range(1, 8):
            suma = suma + T31a[1][i - 1] * T31a[2][i - 1] * (delt ** (T31a[2][i - 1] - 1)) * (tau ** T31a[3][i - 1])
        for i in range(8, 35):
            sumb = sumb + T31b[1][i - 8] * np.e ** (-delt ** T31b[4][i - 8]) * (
                        (delt ** (T31b[2][i - 8] - 1)) * (tau ** T31b[3][i - 8]) * (
                            T31b[2][i - 8] - T31b[4][i - 8] * delt ** T31b[4][i - 8]))
        for i in range(35, 40):
            sumc = sumc + T31c[1][i - 35] * (delt ** T31c[2][i - 35]) * (tau ** T31c[3][i - 35]) * np.e ** (
                        -T31c[4][i - 35] * (delt - T31c[7][i - 35]) ** 2 - T31c[5][i - 35] * (
                            tau - T31c[6][i - 35]) ** 2) * (
                            T31c[2][i - 35] / delt - 2 * T31c[4][i - 35] * (delt - T31c[7][i - 35]))
        for i in range(40, 43):
            sumd = sumd + T31d[1][i - 40] * ((delta(delt, tau, i) ** T31d[3][i - 40]) * (
                        psi(delt, tau, i) + delt * dpsi_ddelt(delt, tau, i)) + ddeltab_ddelt(delt, tau, i) * delt *
psi(
                delt, tau, i))
        return suma + sumb + sumc + sumd

    return R * T * (1 + tau * (phi_ot(delt, tau) + phi_tt(delt, tau)) + delt * phi_td(delt, tau))  # kJ/kg, equation
from table 3
```

C) Entropy Function (entropy.py)

```
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp
from scipy.optimize import minimize
import thermopy
from thermopy import iapws

##tabulated values
T27 = [[] for _ in range(3)]
for i in range(1, 9):
    T27[0].append(i)  # i
T27[1] = [8.37304456, -3.70454304, 2.5, 1.99427042, .62105248, .41195293, 1.04028922, .08327678]  # a^o
T27[2] = [0, 0, 0, 3.15163, 6.1119, 6.77708, 11.32384, 27.08792]  # theta^o

T31a = [[] for _ in range(4)]
for i in range(1, 8):
    T31a[0].append(i)  # i
T31a[1] = [.38856823203161, .29385475942740e1, -.55867188534934e1, -.76753199592477, .31729005580416,
.54803315897767, .12279411220335]  # n
T31a[2] = [1, 1, 1, 1, 2, 2, 3]  # d
T31a[3] = [0, .75, 1, 2, .75, 2, .75]  # t

T31b = [[] for _ in range(5)]
for i in range(8, 35):
    T31b[0].append(i)  # i
T31b[1] = [.21658961543220e1, .15841735109724e1, -.23132705405503, .58116916431436e-1, -.55369137205382,
.48946615909422,
           -.24275739843501e-1,
           .62494790501678e-1, -.12175860225246, -.37055685270086, -.16775879700426e-1, -.11960736637987,
           -.45619362508778e-1, .35612789270346e-1, -.74427727132052e-2, -.17395704902432e-2,
           -.21810121289527e-1, .24332166559236e-1, -.37440133423463e-1, .14338715756878, -.13491969083286,
           -.23151225053480e-1, .12363125492901e-1,
           .21058321972940e-2, -.33958519026368e-3, .55993651771592e-2, -.30335118055646e-3]  # n
T31b[2] = [1., 2., 4., 5., 5., 5., 6., 6., 6., 1., 1., 4., 4., 4., 7., 8., 2., 3., 3., 5., 5., 6., 7., 8., 10.,
4., 8.]  # d
T31b[3] = [1.5, 1.5, 2.5, 0., 1.5, 2., 0., 1., 2., 3., 6., 3., 6., 8., 6., 0., 7., 12., 16., 22., 24., 16., 24.,
8., 2., 28., 14.]  # t
T31b[4] = [1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2., 3., 3., 3., 4., 4., 4., 4., 4., 4.,
5., 6.]  # c


T31c = [[] for _ in range(8)]
for i in range(35, 40):
    T31c[0].append(i)  # i
T31c[1] = [-.21365488688320e3, .26641569149272e5, -.24027212204557e5, -.28341603423999e3, .21247284400179e3]  # n
T31c[2] = [2, 2, 2, 3, 3]  # d
T31c[3] = [1, 0, 1, 3, 3]  # t
T31c[4] = [25, 25, 25, 15, 20]  # alpha
T31c[5] = [325, 300, 300, 275, 275]  # beta
T31c[6] = [1.16, 1.19, 1.19, 1.25, 1.22]  # lambda
T31c[7] = [1, 1, 1, 1, 1]  # eps

T31d = [[] for _ in range(9)]
for i in range(40, 43):
    T31d[0].append(i)  # i
T31d[1] = [-.66642276540751, .72608632349897, .55068668612842e-1]  # n
T31d[2] = [3.5, 3.5, 3.]  # a
T31d[3] = [.875, .925, .875]  # b
T31d[4] = [.3, .3, .3]  # beta
T31d[5] = [.7, .7, .7]  # A
T31d[6] = [.3, .3, 1.]  # B
T31d[7] = [10, 10, 12.5]  # C
T31d[8] = [275, 275, 275]  # D


def entropy(density, temp):
    T_c = 304.128  # critical temperature, degrees kelvin
    rho_c = 467.6  # critical density, kg/m^3
    R = .1889241  # specific gas constant, kJ/(kg*K)

    tau = T_c / temp
```

```
delt = density / rho_c

##inputs
# from table 32
def delta(delt, tau, i):
    return (theta(delt, tau, i)) ** 2 + T31d[6][i - 40] * ((delt - 1) ** 2) ** T31d[2][i - 40]

def psi(delt, tau, i):
    return np.e ** (-T31d[7][i - 40] * (delt - 1) ** 2 - T31d[8][i - 40] * (tau - 1) ** 2)

def theta(delt, tau, i):
    return (1 - tau) + T31d[5][i - 40] * ((delt - 1) ** 2) ** (1 / (2 * T31d[4][i - 40]))

# functions
def phi_o(delt, tau):  # Table 28, row 1
    sum1 = 0
    x = 0
    for i in range(4, 9):
        sum1 = sum1 + T27[1][i - 1] * np.log(1 - np.e ** (-T27[2][i - 1] * tau))
    return np.log(delt) + T27[1][0] + T27[1][1] * tau + T27[1][2] * np.log(tau) + sum1

def phi_t(delt, tau):  # from table 32, eqn 1
    suma = 0
    sumb = 0
    sumc = 0
    sumd = 0
    for i in range(1, 8):
        suma = suma + T31a[1][i - 1] * (delt ** T31a[2][i - 1]) * (tau ** T31a[3][i - 1])
    for i in range(8, 35):
        sumb = sumb + T31b[1][i - 8] * (delt ** T31b[2][i - 8]) * (tau ** T31b[3][i - 8]) * np.e ** (
            -delt ** T31b[4][i - 8])
    for i in range(35, 40):
        sumc = sumc + T31c[1][i - 35] * (delt ** T31c[2][i - 35]) * (tau ** T31c[3][i - 35]) * (np.e ** (
                -T31c[4][i - 35] * (delt - T31c[7][i - 35]) ** 2 - T31c[5][i - 35] * (tau - T31c[6][i - 35])
** 2))
    for i in range(40, 43):
        sumd = sumd + T31d[1][i - 40] * ((delta(delt, tau, i)) ** T31d[3][i - 40]) * delt * psi(delt, tau, i)
    return suma + sumb + sumc + sumd

##inputs
# from table 32

def ddeltab_dtau(delt, tau, i):
    return -2 * theta(delt, tau, i) * T31d[3][i - 40] * delta(delt, tau, i) ** (T31d[3][i - 40] - 1)

def dpsi_dtau(delt, tau, i):
    return -2 * T31d[8][i - 40] * (tau - 1) * psi(delt, tau, i)

##functions
def phi_ot(delt, tau):  # Table 28, row 5
    sum2 = 0
    for i in range(4, 9):
        sum2 = sum2 + T27[1][i - 1] * T27[2][i - 1] * (((1 - np.e ** (-T27[2][i - 1] * tau)) ** -1) - 1)
    return T27[1][1] + T27[1][2] / tau + sum2

def phi_tt(delt, tau):  # from table 32, eqn 4
    suma = 0
    sumb = 0
    sumc = 0
    sumd = 0
    for i in range(1, 8):
        suma = suma + T31a[1][i - 1] * T31a[3][i - 1] * (delt ** T31a[2][i - 1]) * (
                tau ** (T31a[3][i - 1] - 1))
    for i in range(8, 35):
        sumb = sumb + T31b[1][i - 8] * T31b[3][i - 8] * (delt ** T31b[2][i - 8]) * (
                tau ** (T31b[3][i - 8] - 1)) * np.e ** (-delt ** T31b[4][i - 8])
    for i in range(35, 40):
        sumc = sumc + T31c[1][i - 35] * (delt ** T31c[2][i - 35]) * (tau ** T31c[3][i - 35]) * (
                np.e ** (-T31c[4][i - 35] *
                         (delt - T31c[7][i - 35]) ** 2 - T31c[5][i - 35] * (tau - T31c[6][i - 35]) ** 2)) * (
                       T31c[3][i - 35] / tau - 2 * T31c[5][i - 35] * (tau - T31c[6][i - 35]))
    for i in range(40, 43):
        sumd = sumd + T31d[1][i - 40] * delt * (ddeltab_dtau(delt, tau, i) * psi(delt, tau, i) + (
                        delta(delt, tau, i) ** T31d[3][i - 40]) * dpsi_dtau(delt, tau, i))
        return suma + sumb + sumc + sumd

    return R * (tau * (phi_ot(delt, tau) + phi_tt(delt, tau)) - phi_o(delt, tau) - phi_t(delt, tau))  #
kJ/(kg*K), equation from table 3
```

D) Pressure Function (pressure.py)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
import thermopy
from thermopy import iapws

##tabulated values
T27 = [[] for _ in range(3)]
for i in range(1, 9):
    T27[0].append(i)  # i
T27[1] = [8.37304456, -3.70454304, 2.5, 1.99427042, .62105248, .41195293, 1.04028922, .08327678]  # a^o
T27[2] = [0, 0, 0, 3.15163, 6.1119, 6.77708, 11.32384, 27.08792]  # theta^o

T31a = [[] for _ in range(4)]
for i in range(1, 8):
    T31a[0].append(i)  # i
T31a[1] = [.38856823203161, .29385475942740e1, -.55867188534934e1, -.76753199592477, .31729005580416,
.54803315897767, .12279411220335]  # n
T31a[2] = [1, 1, 1, 1, 2, 2, 3]  # d
T31a[3] = [0, .75, 1, 2, .75, 2, .75]  # t

T31b = [[] for _ in range(5)]
for i in range(8, 35):
    T31b[0].append(i)  # i
T31b[1] = [.21658961543220e1, .15841735109724e1, -.23132705405503, .58116916431436e-1, -.55369137205382,
.48946615909422,
           -.24275739843501e-1,
           .62494790501678e-1, -.12175860225246, -.37055685270086, -.16775879700426e-1, -.11960736637987,
           -.45619362508778e-1, .35612789270346e-1, -.74427727132052e-2, -.17395704902432e-2,
           -.21810121289527e-1, .24332166559236e-1, -.37440133423463e-1, .14338715756878, -.13491969083286,
           -.23151225053480e-1, .12363125492901e-1,
           .21058321972940e-2, -.33958519026368e-3, .55993651771592e-2, -.30335118055646e-3]  # n
T31b[2] = [1., 2., 4., 5., 5., 5., 6., 6., 6., 1., 1., 4., 4., 4., 7., 8., 2., 3., 3., 5., 5., 6., 7., 8., 10.,
4., 8.]  # d
T31b[3] = [1.5, 1.5, 2.5, 0., 1.5, 2., 0., 1., 2., 3., 6., 3., 6., 8., 6., 0., 7., 12., 16., 22., 24., 16., 24.,
8., 2., 28., 14.]  # t
T31b[4] = [1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2., 3., 3., 3., 4., 4., 4., 4., 4., 4.,
5., 6.]  # c

T31c = [[] for _ in range(8)]
for i in range(35, 40):
    T31c[0].append(i)  # i
T31c[1] = [-.21365488688320e3, .26641569149272e5, -.24027212204557e5, -.28341603423999e3, .21247284400179e3]  # n
T31c[2] = [2, 2, 2, 3, 3]  # d
T31c[3] = [1, 0, 1, 3, 3]  # t
T31c[4] = [25, 25, 25, 15, 20]  # alpha
T31c[5] = [325, 300, 300, 275, 275]  # beta
T31c[6] = [1.16, 1.19, 1.19, 1.25, 1.22]  # lambda
T31c[7] = [1, 1, 1, 1, 1]  # eps

T31d = [[] for _ in range(9)]
for i in range(40, 43):
    T31d[0].append(i)  # i
T31d[1] = [-.66642276540751, .72608632349897, .55068668612842e-1]  # n
T31d[2] = [3.5, 3.5, 3.]  # a
T31d[3] = [.875, .925, .875]  # b
T31d[4] = [.3, .3, .3]  # beta
T31d[5] = [.7, .7, .7]  # A
T31d[6] = [.3, .3, 1.]  # B
T31d[7] = [10, 10, 12.5]  # C
T31d[8] = [275, 275, 275]  # D

def pressure(rho,T):
    R = .1889241  # specific gas constamt, kJ/(kg*K)
    T_c = 304.128  # critical temperature, kelvin
    tau = T_c / T
    rho_c = 467.6  # critical density, kg/m^3
    delt = rho / rho_c

    ##inputs
    # from Table 32
    def delta(delt, tau, i):
        return (theta(delt, tau, i)) ** 2 + T31d[6][i - 40] * ((delt - 1) ** 2) ** T31d[2][i - 40]

    def psi(delt, tau, i):
        return np.e ** (-T31d[7][i - 40] * (delt - 1) ** 2 - T31d[8][i - 40] * (tau - 1) ** 2)

    def theta(delt, tau, i):
        return (1 - tau) + T31d[5][i - 40] * ((delt - 1) ** 2) ** (1 / (2 * T31d[4][i - 40]))

    ##inputs
    # from table 32
    def dpsi_ddelt(delt, tau, i):
        return -2 * T31d[7][i - 40] * (delt - 1) * psi(delt, tau, i)

    def ddelta_ddelt(delt, tau, i):
        return (delt - 1) * (T31d[5][i - 40] * theta(delt, tau, i) * (
                2 / T31d[4][i - 40]) * ((delt - 1) ** 2) ** (-1 + 1 / (2 * T31d[4][i - 40])) + 2 * T31d[6][i -
40] * T31d[2][i - 40] * ((delt - 1) ** 2) ** (T31d[2][i - 40] - 1))

    def ddeltab_ddelt(delt, tau, i):
        return T31d[3][i - 40] * (delta(delt, tau, i) ** (T31d[3][i - 40] - 1)) * ddelta_ddelt(delt, tau, i)

    ##function
    def phi_td(delt, tau): #table 32, eqn 2
        suma = 0
        sumb = 0
        sumc = 0
        sumd = 0
        for i in range(1, 8):
            suma = suma + T31a[1][i - 1] * T31a[2][i - 1] * (delt ** (T31a[2][i - 1] - 1)) * (tau ** T31a[3][i -
1])
        for i in range(8, 35):
            sumb = sumb + T31b[1][i - 8] * np.e ** (-delt ** T31b[4][i - 8]) * (
                        (delt ** (T31b[2][i - 8] - 1)) * (tau ** T31b[3][i - 8]) * (
                            T31b[2][i - 8] - T31b[4][i - 8] * delt ** T31b[4][i - 8]))
        for i in range(35, 40):
            sumc = sumc + T31c[1][i - 35] * (delt ** T31c[2][i - 35]) * (tau ** T31c[3][i - 35]) * np.e ** (
                        -T31c[4][i - 35] * (delt - T31c[7][i - 35]) ** 2 - T31c[5][i - 35] * (
                            tau - T31c[6][i - 35]) ** 2) * (
                               T31c[2][i - 35] / delt - 2 * T31c[4][i - 35] * (delt - T31c[7][i - 35]))
        for i in range(40, 43):
            sumd = sumd + T31d[1][i - 40] * ((delta(delt, tau, i) ** T31d[3][i - 40]) * (
                        psi(delt, tau, i) + delt * dpsi_ddelt(delt, tau, i)) + ddeltab_ddelt(delt, tau, i) * delt
* psi(delt, tau, i))
        return suma + sumb + sumc + sumd
    return rho * R * T * (1 + delt * phi_td(delt, tau)) #kPa, eqn from table 3
```

E) Complete CO_2 Properties Function (CO2properties.py)

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
import thermopy
from thermopy import iapws
from scipy.optimize import fsolve
from sympy import symbols, Eq, solve

from entropy import entropy
from int_energy import int_energy
from enthalpy import enthalpy
from pressure import pressure
import random as random

def CO2properties(prop1,prop2,v1,v2,out):
    #p1 and p2 are properties: temperature (T), pressure, (P), density (D), enthalpy (H), entropy (S), and internal
energy (U)
    #v1 and v2 are the values of p1 and p2
    #out is desired output property (T,P,D,U,H,S)
    #UNITS: T [K], P {kPa], D [kg/m^3], U [kJ/kg], H [kJ/kg], S [kJ/(kg*K)]

    if prop1 == "T":
        T = v1
    elif prop1 == "D":
        D = v1
    elif prop1 == "U":
        U = v1
    elif prop1 == "H":
        H = v1
    elif prop1 == "S":
        S = v1
    elif prop1 == "P":
        P = v1

    if prop2 == "T":
        T = v2
    elif prop2 == "D":
        D = v2
    elif prop2 == "U":
        U = v2
    elif prop2 == "H":
        H = v2
    elif prop2 == "S":
        S = v2
    elif prop2 == "P":
        P = v2

    if prop1 == "T" and prop2 == "D" or prop1 == "D" and prop2 == "T":
        rho = D

        P = pressure(rho, T)
        H = enthalpy(rho, T)
        S = entropy(rho, T)
        U = int_energy(rho, T)

    elif prop1 == "T" and prop2 == "P" or prop1 == "P" and prop2 == "T":
        def minpressure(x):
            return abs(pressure(x, T) - P)

        result = 2
        while result > 1:
            guess = random.randint(100,1000)
            bnds = [(1, 1500)]
            res = minimize(minpressure, guess, method='SLSQP', bounds=bnds, options={'maxiter': 3000, 'ftol': 1e-20})
            result = res.fun
        D = res.x[0]
        rho = D
        H = enthalpy(rho, T)
        S = entropy(rho, T)
        U = int_energy(rho, T)

    elif prop1 == "T" and prop2 == "U" or prop1 == "U" and prop2 == "T":
        def minenergy(x):
            return abs(int_energy(x, T) - U)

        bnds = [(1, 1500)]
        res = minimize(minenergy, [200], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
        D = res.x[0]
        rho = D
        H = enthalpy(rho, T)
        S = entropy(rho, T)
        P = pressure(rho, T)

    elif prop1 == "T" and prop2 == "H" or prop1 == "H" and prop2 == "T":
        def minenthalpy(x):
            return abs(enthalpy(x, T) - H)

        bnds = [(1, 1500)]
        res = minimize(minenthalpy, [400], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
        D = res.x[0]
        rho = D
        S = entropy(rho, T)
        P = pressure(rho, T)
        H = int_energy(rho, T)

    elif prop1 == "T" and prop2 == "S" or prop1 == "S" and prop2 == "T":
        def minentropy(x):
            return abs(entropy(x, T) - S)

        bnds = [(1, 1500)]
        res = minimize(minentropy, [10], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
        D = res.x[0]
        rho = D
        S = entropy(rho, T)
        P = pressure(rho, T)
        U = int_energy(rho, T)

    elif prop1 == "D" and prop2 == "P" or prop1 == "P" and prop2 == "D":
        rho = D
        def minpressure(x):
            return abs(pressure(rho, x) - P)

        bnds = [(300, 1100)]
        res = minimize(minpressure, [10000], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
        T = res.x[0]
        H = enthalpy(rho, T)
        S = entropy(rho, T)
        U = int_energy(rho, T)

    elif prop1 == "D" and prop2 == "U" or prop1 == "U" and prop2 == "D":
        rho = D
        def minenergy(x):
           return abs(int_energy(rho, x) - U)

        bnds = [(300,1100)]
        res = minimize(minenergy, [200], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
        T = res.x[0]
        # print('found U = ',int_energy(rho,T))
        # print(('int energy with correct T',int_energy(rho,300)))
        # print('found T = ',T)
        H = enthalpy(rho, T)
        S = entropy(rho, T)
        P = pressure(rho, T)

    elif prop1 == "D" and prop2 == "H" or prop1 == "H" and prop2 == "D":
        rho = D
        def minenthalpy(x):
            return abs(enthalpy(rho, x) - h)

        bnds = [(300, 1100)]
        res = minimize(minenthalpy, [400], method='Nelder-Mead')
        T = res.x[0]
        S = entropy(rho, T)
        P = pressure(rho, T)
        U = int_energy(rho, T)

    elif prop1 == "D" and prop2 == "S" or prop1 == "S" and prop2 == "D":
        rho = D

        def minentropy(x):
            return abs(entropy(rho, x) - S)

        bnds = [(300, 1100)]

        res = minimize(minentropy, [10], method='Nelder-Mead')
        T = res.x[0]
        H = enthalpy(rho, T)
        P = pressure(rho, T)
        U = int_energy(rho, T)

    elif  prop1 == "P" and prop2 == "S" or prop1 == "S" and prop2 == "P":

        def minPS(t):
            D, T = t
            return abs(pressure(D, T) - P) + 1000 * abs(entropy(D, T) - S)

        bnds = ((1, 1500), (1, 1100))

        result = 10
        while result > 1:
            guess = 20 * random.randint(1, 10), 100 * random.randint(1, 10)
            res = minimize(minPS, [guess], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
            result = res.fun

        D = res.x[0]
        T = res.x[1]
        H = enthalpy(D, T)
        U = int_energy(D, T)


    elif  prop1 == "P" and prop2 == "H" or prop1 == "H" and prop2 == "P":

        def minPH(t):
            D, T = t
            return abs(pressure(D, T) - P) + 100 * abs(enthalpy(D, T) - H)

        bnds = ((1, 1500), (1, 1100))

        result = 10
        while result > 1:
            guess = 20 * random.randint(1, 10), 100 * random.randint(1, 10)
            res = minimize(minPH, [guess], method='SLSQP', bounds=bnds, options={'maxiter': 6000, 'ftol': 1e-40})
            result = res.fun

        D = res.x[0]
        T = res.x[1]
        S = entropy(D, T)
        U = int_energy(D, T)


    if out == "T":
        final = T
    elif out == "P":
        final = P
    elif out == "H":
        final = H
    elif out == "S":
        final = S
    elif out == "U":
        final = U
    elif out == "D":
        final = D

    return final
```