Ivan Suminski
Prof. Mark Newman
SI507 - Project Data Checkpoint
19 April, 2020

Github Repository: https://github.com/isuminski/proj-final.git

Data Source: NIST Thermophysical Online Webbook

Origin: Homepage: https://webbook.nist.gov/chemistry/fluid/
Base-URL: https://webbook.nist.gov/cgi/fluid.cgi

Query String Parameters (Example): (specific parameters vary by search)

```
{'Action' : 'Load',
'ID' : <fluid ID>,
'Type' : 'IsoBar',
'Digits' : '5',
'P' : ##,
'TLow' : ##,
'THigh' : ##,
'TInc' : ##,
'RefState' : 'DEF',
'TUnit' : <unit expression>,
'PUnit' : <unit expression>,
'DUnit' : <unit expression>,
'HUnit' : <unit expression>,
'WUnit' : <unit expression>,
'VisUnit' : 'uPa*s',
'STUnit' : 'N%2Fm'}
```

Format: html

Access: Although an API-like URL + query string are used, the data presents as an html table for each search, which I then am able to scrape using the BeautifulSoup python package. In this case, every table for a given search type loads with the same columns in the same order, making it easier to scrape. Each thermodynamic state occupies a row in this table.

Summary: Because the tables represent a continuous set of data, it is impossible to determine "how many" records there are, but each thermodynamic cycle will involve four unique states. (However, more records may need to be searched to "solve" each state.) Each row of the table lists a unique set of thermodynamic state values. When the parameter "Type" is isobaric or isothermal (two of three search types used for this application), the table columns are as follows:

Temperature | Pressure | Density | Specific Volume | Internal Energy | Enthalpy | Cv | Cp | Speed of Sound | Joule-Thomson Coefficient | Viscosity | Thermal Conductivity | Phase

For the purposes of this application Density, Speed of Sound, Joule-Thomson Coefficient, Viscosity, and Thermal Conductivity will be ignored. Others, such as Internal Energy, Cp and Cv (heat capacities at constant volume and constant pressure respectively) will be stored in the database as they may prove useful for solving other aspects of a problem.

Caching: To avoid loading tables twice (and since it is likely that similar tables will be used for initial property searches), a cache of the raw page html will be kept in a json file, with a json string of the parameters used as a 'key' to describe a unique table. See the Github project repository for simple code that takes a single temperature and pressure and returns the enthalpy value and phase, using a cache to prevent duplicate searches. The filename for this is: 'practice-cache.py'

Database:

Structure: 2 tables - Cycles and States

Cycles:

Id	Efficiency	SearchTimes	s1_Id	s2_Id	s3_Id	s4_Id
INT (Key)	REAL	INT	INT	INT	INT	INT

Where Id is the primary key for a unique 4-stage rankine cycle, Efficiency is the cycle's thermodynamic efficiency, SearchTimes is the number of times that specific cycle has been searched since the creation of the database, and s#_Id is a foreign key indicating a record in the 'States' table.

States:

Id	Uses	Temp	Press	SpecVol	Energy
INT (Key)	INT	REAL	REAL	REAL	REAL

Enthalpy	Entropy	Cv	Cp	Phase
REAL	REAL	REAL	REAL	REAL

Where Id is the key for a unique thermodynamic state, Uses is the number of times that state has been used in a cycle calculation since the creation of the database, and all other columns correspond directly to state properties as found in the html tables.

Foreign Keys: The INT values for s#_Id in the 'Cycles' table correspond to the primary keys (Id) in the 'States' table.

Function: Due to a misunderstanding of the project requirements when I was coming up with my project, I don't have much real use for a database, since the NIST online Webbook from which I am accessing my data is essentially already a database. Therefore, my database will first-and foremost serve as a second level of caching, in which before any processing must be done on html data from the cache, it can first be noted whether a specific state (or cycle) has been searched before. The SearchTimes and Uses columns from each table are interesting because they will be modified for a record each time that record is read. The other use for the database is that it provides access to potentially useful information that is otherwise hidden in the program's calculations (such as entropy, internal energy, and heat capacity). This way, if additional calculations are needed to determine other aspects of the cycle, the necessary thermophysical properties for the specific states in question can be looked up all in one place rather than by going back to NIST and wading through the tables again. Finally, I will display to a user how many times a record has been accessed (either for a state or for a cycle), just so that I do SOMETHING with the data in the database for the purposes of this project! For code in which database creation is practiced, see 'practice-DB.py' in the github repository.

Presentation:

If all goes well, I'd like to make my 'calculator' entirely accessible through a Flask app. Basically, there will be a homepage that describes the functionality and provides links to a couple of different routes for different functions.

For the sake of usefulness and flexibility (and honestly because writing this code first will help with the debugging process), one route will be a simple state calculator, wherein state variables can be looked up (and records added to the database as a result) independently of any cycle calculations. On this page, a series of text boxes will be available. If any two state variables are provided, the program will determine the missing ones, locate the state in the NIST Online Webbook (if the record doesn't exist in the database) and populate the remaining fields once a "look-up" button has been pressed.

A second route will be created to "solve" four-state Rankine cycles. This page will simply have four sections corresponding to each state which will simply be the same process as the state calculator. (Hopefully I can figure out how to make this work on a single page. Otherwise, it will be done in four sequential pages.) The end result will be a completely defined power cycle and the cycle's efficiency. Additional error handling will be needed for this route, since not any four thermodynamic states will define a power-cycle.

Other development notes:

For this project, I am limiting the working fluid to water, and the application to power-cycles, though the app could easily be expanded to include other working fluids and refrigeration cycles as well. I will also be limiting my use of units to a single set of "metric" and "imperial" units that I have pre-determined. (They will be the most common units used for industrial power production calculations.) On the state-calculator route, the units can be specified for each search, but for the cycle-calculator route, the same set of units must be used for each of the four states, so this control will have to be altered slightly. The other implication of allowing two unit systems is that I will either have to create two databases or two additional tables in the existing one (likely the latter, though I am not knowledgeable enough to know which would be more optimal).

One other consideration for the cycle-efficiency route would be the ability to define isentropic efficiencies for the pump and turbine, which could pre-populate a state variable on either side of each leg of the cycle. However, for the purposes of this project, I think I will leave out this functionality.