SRP III - Project Journal

Brian LaMons

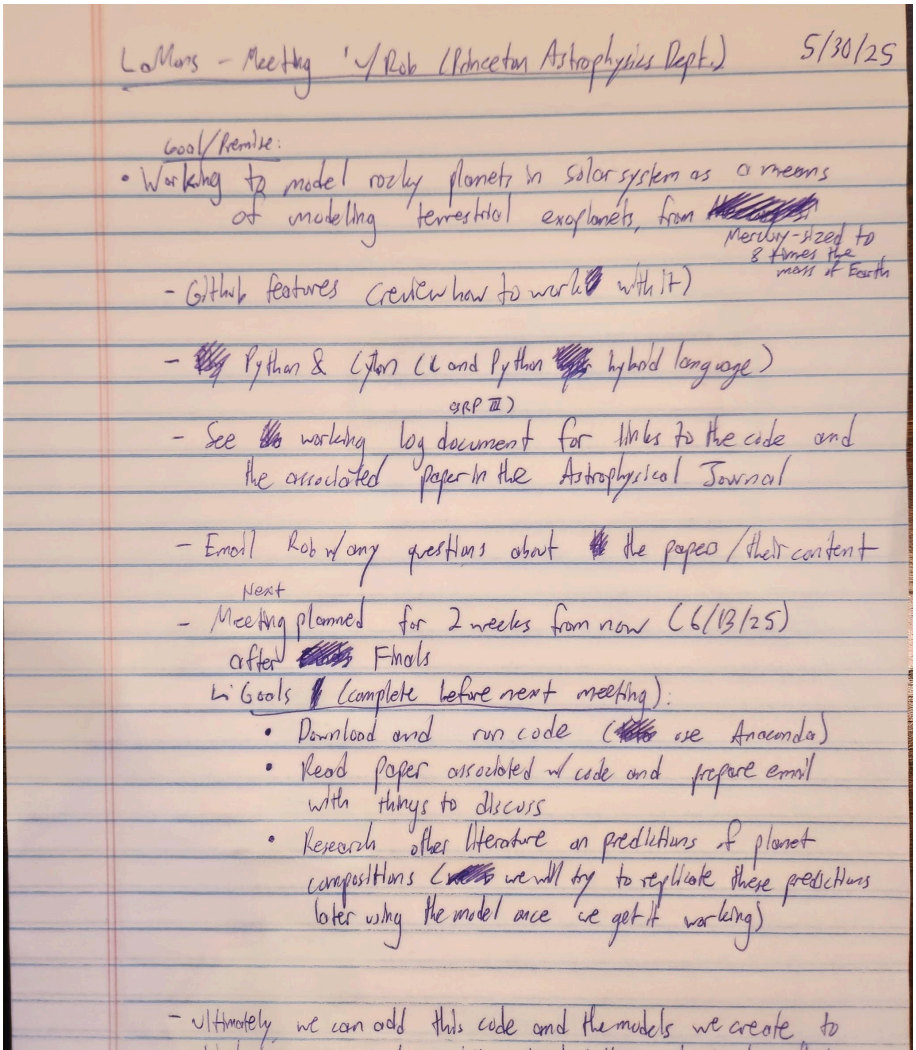
May-September 2025

I. 5.30.2025 Entry: Initial Meeting & Project Plan:

A. Notes on Zoom Meeting with Rob:

1. General project premise:
    a) Modeling the evolution of terrestrial exoplanets ranging from Mercury-sized to the scale of eight Earth masses by creating models from pre-developed code and adding onto that code to refine and optimize produced models.
    b) Eventually, we are going to try to combine my terrestrial modeling code with the gas giant evolution model Rob is currently working on so that we can model and predict the phenomena surrounding the internal rocky core of those gas giant planets.

2. See the image below, which details my notes on this first meeting with Rob, a 5th-year graduate student at the Princeton University Department of Astrophysical Sciences.



LaMons - Meeting w/ Rob (Princeton Astrophysics Dept.) 5/30/25

Goal/Premise:
- Working to model rocky planets in solar system as a means of modeling terrestrial exoplanets, from Mercury-sized to 8 times the mass of Earth
- GitHub features (review how to work with it)
- Python & Cython (C and Python hybrid language)
- See working log document for links to the code and the associated paper in the Astrophysical Journal (SRP III)
- Email Rob w/ any questions about the papers / their content
- Next Meeting planned for 2 weeks from now (6/13/25) after Finals

Goals (complete before next meeting):
- Download and run code (use Anaconda)
- Read paper associated w/ code and prepare email with things to discuss
- Research other literature on predictions of planet compositions (we will try to replicate these predictions later using the model once we get it working)

- Ultimately we can add this code and the models we create to

B. Goals to Complete Prior to Next Meeting (6/13/25):
   1. Download and run terrestrial planetary evolution code, linked here (use Anaconda installation): https://github.com/zhangjis/CMAPPER_rock/tree/main
   2. Read the paper associated with the code, linked here, and prepare a list of concepts / parts of the paper to discuss with Rob (specifically, the parts that are confusing or new): https://iopscience.iop.org/article/10.3847/1538-4357/ac8e65
   3. Research additional scientific literature on predictions of planet compositions and evolutions (predictions made by observation), as we will ultimately try to replicate these predictions by creating models from the above code

---

II. **6.13-7.4.25 Entry: Notes, Code Download, & Results of Preliminary Models:**

A. *Thermal Evolution and Magnetic History of Rocky Planets* (Zhang & Rogers 2022): Notes and Links to Useful Context:

1. Important terms are marked in **bold** and defined below.

2. Abstract Notes:
   a) The paper and attached code is a model to determine the conditions in which a terrestrial planet could host a magnetic field via a **dynamo** in a liquid iron core or magma ocean.
      (1) Here, a **dynamo** is a system that generates a magnetic field, seen on Earth as the movement of liquid iron in the outer core that generates electric current and a resulting magnetic field.
   b) Other specifics are discussed later in these notes even if they are briefly mentioned in the abstract.

3. Introduction Notes:
   a) Knowing about the presence of a magnetic field around an exoplanet would grant further insight into its largely unknown internal dynamics
   b) **CMB** = core-mantle boundary

c) Currently, a single mass-radius measurement from transit observation of an exoplanet can be evidence for multiple different kinds of interior dynamics, presenting inaccuracy
d) Magnetic field in terrestrial exoplanet ⇒ liquid convecting layer
e) Magnetic fields ⇒ atmosphere protection (and possibly liquid water as well); allows for a comparison of how different magnetic fields cause different atmospheres, potentially leading to a greater understanding of how our atmosphere preserves water on Earth and a better knowledge of how habitable any one exoplanet could be
f) Radio signals and radio aurora emissions have been used to try to study the magnetic fields of gas giants thus far
    (1) However, the strength of the magnetic fields for sub-Neptunes and super-Earths is expected to be lower, so ground-based instruments are not applicable
    (2) Space-based instruments have not been used to study magnetic fields yet, but the SunRISE probe could prove that it is possible (*check current updates*)
g) Previous groups have used a **box model** to study the evolution of terrestrial planets
    (1) One component system of the planet is a 'box'; here, the mantle is one, the core is another
    (2) Box models assume whole mantle convection and measure the temperature- they serve as so-called "zero-dimensional energy-balance models."
    (3) As a result, box models do not account for the effect of density and pressure changes in different regions of the core and mantle as a planet expands
h) The code this paper used thus is a one-dimensional model for planetary evolution with regard to either a liquid iron or molten silicate dynamo (below is points about the model that are confusing, and *I should ask about*):
    (1) The model solves for '**melt fraction**.'

> Fractional or Rayleigh melting: "increments of melt form from crystals at instantaneous chemical equilibrium with the melt, but the melt is continuously removed as it is formed (e.g., escapes by porous flow and its own buoyancy). Overall, the system is not at chemical equilibrium for all the original phases and materials involved."

(2) It uses a '**generalized Schwarzschild criterion**' to determine where convection in the mantle occurs.

There will be stability against convection as long as (here the scenario uses a star rather than a planet, but it is a similar situation when deriving the Schwarzschild criterion):

$$\left(\frac{dP}{d\rho}\right)_{star} < \left(\frac{dP}{d\rho}\right)_{adiabatic}$$

Where the slope (derivative of the pressure vs. density) in the star is less than the slope in an ideal adiabatic system.

Ultimately, the Schwarzschild criterion for a star can be written as:

$$\left(\frac{P}{T}\frac{dT}{dP}\right)_{star} < \nabla_{adiabatic}$$

Where nabla (∇) refers to the logarithmic derivative of temperature with respect to pressure.

Via this formula, one finds that convection will commence if the $\nabla_{adiabatic}$ exceeds 0.4.

Note that **adiabatic** refers to when a reaction occurs that does not transfer heat between itself and the surrounding environment (heat does not leave the system).

Also note that *stability*, in this case against convection, refers to the resistance of a system to experience the temperature change caused by convection (or any other type of heat transfer, for that matter).

Also note that a *logarithmic derivative* is, when *f* is a function of x and its domain is restricted to real positive numbers, equal to the derivative of the natural log of f(x):

$$\frac{f'}{f} = \frac{d}{dx} ln(f(x)) = \frac{1}{f(x)} \frac{df(x)}{dx}$$

i) The model is tested with 12 planets, 2 planetary masses, 2 CMFs (**core mass fractions**), and 3 equilibrium temperatures.

(1) **Core mass fraction** is a measure of how much of a planet's mass can be found in its core. Mercury is often used as a benchmark for this metric, as it has a uniquely high CMF of 0.7. Thus, rocky exoplanets with high CMFs are typically classified as Mercury analogues.

4. Methods notes:
   a) In this paper, they only consider planets with liquid iron cores and molten silicate mantles.
   b) Essentially, the model breaks a planet down radially into different cells with the same mass based on the planet's mass and CMF. All cells have the same internal structure until they are updated by a **Henyey solver**, which either confirms that the initial structure is accurate or modifies it to accurately fit the planetary parameters.
      (1) When attempting to break down planetary interiors, one must employ the following, known as the **structure equations**:

$$\frac{\mathrm{d}r}{\mathrm{d}m} = \frac{1}{4\pi r^2 \rho}$$
$$\frac{\mathrm{d}P}{\mathrm{d}m} = -\frac{Gm}{4\pi r^4}$$
$$\frac{\mathrm{d}l}{\mathrm{d}m} = \epsilon - \epsilon_\nu$$
$$\frac{\mathrm{d}T}{\mathrm{d}m} = -\frac{GmT}{4\pi r^4 P}\nabla.$$

      (2) *Note that the equations above and the information in the ensuing bullet notes below for this section were derived from notes on stellar interiors, not planetary ones. However, upon further research, I have found that they can be applied as they are (as well as the Henyey method) to discover more about the internal workings of exoplanets, especially gas giants. In the paper, though, they employ*

similar versions of these equations, just with partial derivatives.

(3) These allow us to derive equations for the **boundary conditions**, i.e. the different parameters on either side of the dividing line between planetary layers, say, the crust and the mantle.

(4) But because the structure equations are not *initial-value problems*, but rather, are ***boundary value problems, or BVPs*** (A set of differential equations defined over a region, with boundary conditions defined at both ends of the interval), they must be solved with what is known as finite differences, relaxation, or the **Henyey solver** (also called the **Henyey method**).

(a) With this method, the planet is divided into *n* mass shells (each with an index, *k*), and then we find all of the parameters for each respective shell. The structure equations are adapted for this purpose by replacing the derivatives with "numerical approximations based on finite differences," as seen in the one below, for example:

$$\frac{P_{k+1} - P_k}{m_{k+1} - m_k} = \frac{1}{4\pi r_{k+1/2}^2 \rho_{k+1/2}}$$

(b) They are then solved by using a version of the **Newton-Raphson method**:

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)},$$

(c) As one iteratively applies this method to the modified structure equations and the boundary conditions, one converges towards a solution of those structure equations. Convergence is typically "very rapid," and it works well, as it utilizes

previously approximated values to move closer and closer to a solution.

c) First, the model establishes initial values for pressure, temperature, etc. at the center of the planet via the *shooting* method, which is another way of solving the structure equations (but one that tends to be less efficient than the Henyey method). This only satisfies the boundary conditions at the core or at the surface, *not both*.
    (1) It is the Henyey solver that uses these initial values as inputs so that it can iterate over the 1000+ mass shells the model creates and simultaneously satisfy both boundary conditions. By doing so, we can actually model energy distributions throughout the planet and get closer to predicting the existence of a magnetic field.

d) **EoS** refers to an ***Equation of State***, an equation that takes into account the relationship between pressure, temperature, and volume within a segment of or a whole planet.
    (1) This program can be extended to include new and different EoSs, depending on the planet one would like to model.
    (2) They chose the semiempirical EoSs used in this study in order to model planets with molten silicate mantles and a solid iron inner core with a liquid iron outer core.
        (a) For the molten silicate, they use the EoS for liquids under high temperature and pressure conditions developed by Wolf & Bower 2018, consisting of an **isothermal** component and a **thermal perturbation** using a **Rosenfeld–Tarazona model**.
            (i) **Isothermal** describes a process/system in which temperature remains constant as other variables change.
            (ii) **Thermal perturbation** refers to isolated, local, or temporary deviations in temperature from normal / equilibrium values, something that can be caused by convection (i.e., in a mantle)
            (iii) The **Rosenfeld–Tarazona model** is an equation for determining certain properties of a fluid in extreme conditions (high density, high pressure).

(b) For the iron and solid silicate, EoSs were selected along similar lines, with both the same components (isothermal and thermal perturbation).
  (i) An **isothermal bulk modulus** is a constant that describes how resistant a substance is to compression when there is constant temperature.
  (ii) To account for density changes in the liquid iron and the silicate minerals (they selected olivine and perovskite), they employ a **Debye thermal pressure term**, calculated using the internal energy per unit mass.
  (iii) To account for density changes in solid iron, they incorporated **anharmonic** (the pressure caused by anharmonic [meaning non-linear] atomic vibrations at high temperatures) and **electrical thermal pressure** equations.
(c) Note that they do not consider the impurities that have been proven to occur in the boundary regions between these layers (i.e., molten silicate "leaking" into the iron outer core, and vice versa).

e) Cooling in the core is assumed to be adiabatic and is solved for using the heat conduction equation.

f) Heat transport in the mantle is written in terms of entropy in the model, as it is an appropriate reference point for substances of both the solid and liquid states.
  (1) However, there is not always a clear line between solid and liquid, so one must take into account both **viscous** (high viscosity, high resistance to flow) and **inviscid** (low viscosity, low resistance to flow) fluids in the boundaries between the core, the surface, and the mantle.

g) Level of mantle viscosity determines the cooling rate of the mantle itself and indicates when it is going to fully solidify. The cooling rate of the mantle also impacts the cooling rate of the core.
  (1) Viscosity for solid silicate is determined by an **Arrhenius formulation** (an equation primarily used to look at the impact on the rate of a chemical reaction as temperature changes) such that it is pressure and temperature dependent in the model. They selected this one method for mantle

viscosity, but the code can take a number of different ones (see pg. 9, Zhang & Rogers 2022).

h) Heat generation via radioactive decay could potentially change how a planet evolves, however here, they model element distribution similar to that of Earth through convection.

i) In this model, planets cool via radiation of heat from the surface, and so heat flow is modeled as **graybody radiation**.
   (1) A **Graybody** is "A body that emits radiation of all wavelengths at a given temperature, which is a constant fraction of the energy of black-body radiation of the same wavelength at the same temperature," (Oxford).
   (2) Rather than differentiating between different kinds of convection, this model automatically captures thermal conduction-dominant boundary layers using **mixing length** (a distance that a fluid will keep its original characteristics before dispersing into the surrounding fluid), chosen here to be Earth-like with **mobile-lid convection**.
      (a) **Mobile-lid convection**, observed on Earth, is what supports plate tectonics; essentially, the "lid" surfaces atop molten silicates in the mantle are separate plates, and shift as fluid flows through the mantle.
      (b) This paper does not explore the impact of changing atmospheric conditions on planetary evolution. Instead, surface pressure is assumed as 1 bar. (Perhaps this is a possible way forward, for improving the model?).

j) A dynamo is possible in terrestrial planets if their **magnetic Reynolds number**, Re_m, surpasses a critical value.
   (1) The **magnetic Reynolds number**, like the standard *Reynolds number* (which predicts whether or not a fluid will smooth out or flow in a turbulent fashion), determines whether a planet's magnetic field will diffuse away and spread out, possibly disappearing altogether (if Re_m is less than one), or it will move along with the plasma flow such that field lines move in the same direction as said plasma (if Re_m is very large).
   (2) Here, $Re_m = 50$ is set as the critical value.

(3) Re_m is evaluated in each cell to determine the possibility of a dynamo, and then it is calculated for the liquid iron core to check if the "dynamo can operate within its lifetime," which depends on convective heat flux in the core. But in general, as long as the magnetic Reynolds number exceeds the critical value and the planet has a liquid iron core, it can support a dynamo and thus a magnetic field.

B. [GUIDE] Code Installation & Initial Trials:

1. Note: after installation of Anaconda, remember to verify said installation using the *Anaconda Prompt* program (accessible from the Anaconda Navigator menu) rather than the computer's primary terminal.
2. I encountered difficulties using the 'make' command (installation step 3a), but with some research, I was able to install it through *conda-forge* directly through Anaconda.
    a) However, because the commands within the makefile are Unix-based, I had to install Git Bash, link it with Anaconda, and then run the 'make dependencies' command.
    b) Note: Paths in Git Bash should be in Unix format (different from the standard Windows file pathing convention. Simply drag and drop the file path into Git Bash from file explorer, and then it should automatically convert to Unix format.
        (1) C: becomes /c/
        (2) Replace backslashes with forward slashes
    c) IMPORTANT:
        (1) For organizational purposes, I created a Python virtual environment, CMAPPER_env. Activate it and deactivate it as shown below.

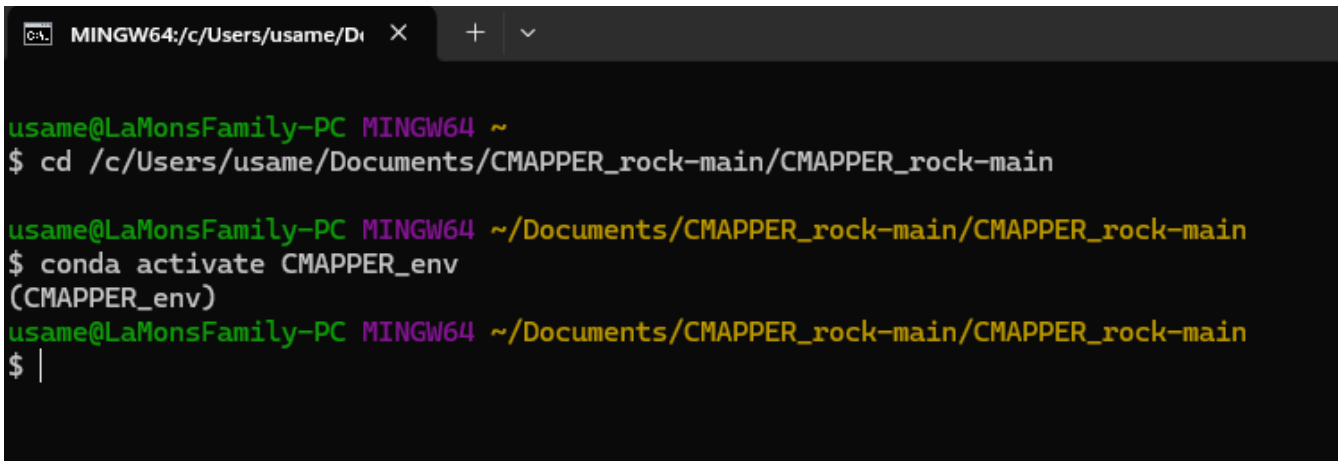
```
# To activate this environment, use
#
#     $ conda activate CMAPPER_env
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

    d) I had to then override the python variable within the Makefile itself in order to get it to recognize Conda's python. Below is the edited Makefile (with paths changed and environment codes removed, as

I am already using a virtual environment with Conda in Git Bash, so it is redundant), for reference:

```
PYTHON=/c/Users/usame/anaconda3/envs/CMAPPER_env/python
PIP=/c/Users/usame/anaconda3/envs/CMAPPER_env/Scripts/pip
CFLAGS="-Ofast -Wno-unreachable-code -Wno-unreachable-code-fallthrough"

all: check-python check-conda dependencies build run

check-python:
	@printf "%b\n" "\n*********************************************"\
		"Checking Python version..."\
		"*********************************************\n"
	@${PYTHON} --version > /dev/null 2>&1\
		&& printf "%b\n" "Note: Python version is $$(${PYTHON} --version) and pip version is $$(${PIP} --version | cut -d ' ' -f1-2). Looks like it should be good!\n"\
		|| ( printf "%b\n" "Python install not found, please install ${PYTHON}\n" && exit 1 )

check-conda:
ifeq ($(CMAP_CONDA_CHECK), FALSE)
	@printf "%b\n" "Skipping Anaconda \$$PATH check..."
else
	@if [ $$(which -a ${PYTHON} | grep "opt/anaconda*/envs/*/bin/${PYTHON}" | wc -l) -ge 2 ]; then\
		printf "%b\n" "Multiple Anaconda envs found in \$$PATH, exiting. Run \`export \$$CMAP_CONDA_CHECK=FALSE\` to disable this check" && exit 1;\
	fi
endif

dependencies:
	@printf "%b\n" "\n*********************************************"\
		"MAKEFILE: Trying to install dependencies..."\
		"*********************************************\n"
	${PIP} install -r requirements.pip

.PHONY: build
build: heat_transport.pyx rocky_class.pyx
	@printf "%b\n" "\n*********************************************"\
		"MAKEFILE: Building Cython code..."\
		"*********************************************\n"
	CFLAGS=${CFLAGS} ${PYTHON} setup_pre_adiabat.py build_ext --inplace

run: build
	@printf "%b\n" "\n*********************************************"\
		"MAKEFILE: Trying to run CMAPPER_rock"\
		"*********************************************\n"
	${PYTHON} test.py

# removes all python packages and compiled Cython code
clean:
	rm -rf venv .venv build *.so *.c

# removes everything, including results folders
clean-all:
	git clean -fdx
```

e) Example startup image (for reference):



Once the above has been performed, enter *make run*, and the simulation should run.

f) To change input parameters, such as CMF, edit the values found in the **input.txt** file within the CMAPPER_rock-main folder, shown below:

```
##### Do not change the order of input parameters
# Planet mass (M_pl) in earth mass (Valid range for the current version: 0.5 - 8.0)
8.0
# Core mass fraction (Valid range for the current version: 0.1 and 0.7)
0.326
# Evolutionary time in billion years (Gyr)
14.0
# Radiogenic heating relative to that of Earth's mantle. We currently don't consider Al24 for its short half life time.
# K
1.0
# Th
1.0
# U238
1.0
# U235
1.0
# viscosity models. For the current version, two rheology models for the viscosity of ppv is included. 1 is for diffusion creep (default) and
et al. 2013.
1.0
# Equilibrium temperature (Valid range for the current version: 255.0 (default) - 2700.0)
2200.0
# radiogenic heating from potassium alone in core. 1.0 being the concentration of potassium with 1 TW heating at 4.5 Gyr.
0.0
1.0
```

**Note that the values in the input.txt sample image above are the default.

III. **<u>7.22-7.29.25 Entry: Second Meeting, Additional Paper Notes, & Comparison of New Simulation Results:</u>**

A. <u>Notes / Discussion from July 22nd Meeting with Rob:</u>

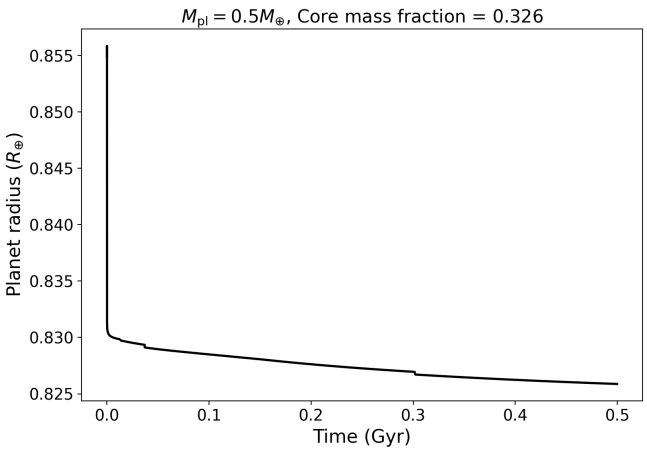
1. INSERT PICTURE OF HARD-COPY NOTES

B. <u>Further Simulation Trials: Comparing & Verifying Data with Table 1 (Fortney et al. 2006):</u>

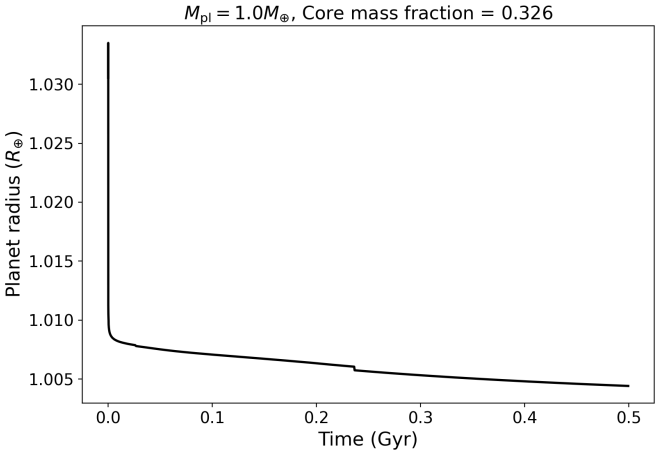
1. Set One: Series of Varying Mass Numbers and Results:

a) Note that for these trials, **CMF = 0.326** (the CMF of Earth) and all other values are held constant, with **equilibrium temp. set to 255k** and evolutionary time set to **t = 0.5 Gyr** (in Fortney et al. 2006, the paper uses 500 Myr, or 0.5 Gyr, as the highest measurement period).

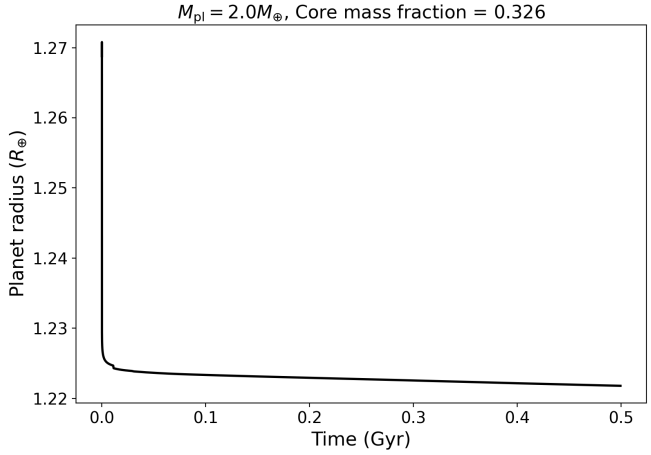
b) I: M = 0.5 Earth Masses



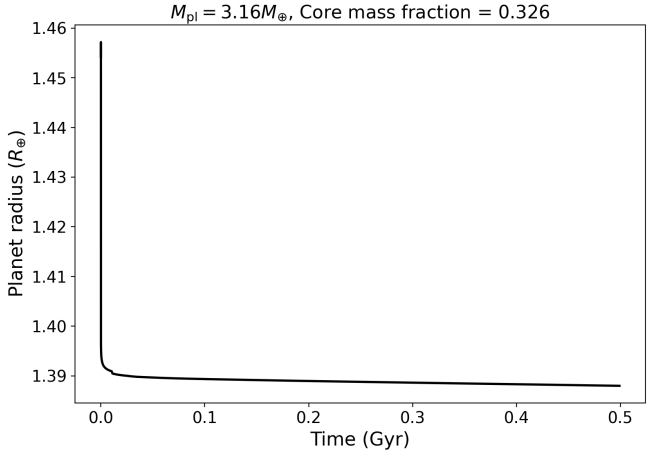
c) II: M = 1.0 Earth Masses



d) III: M = 2.0 Earth Masses



e) IV: M = 3.16 Earth Masses (value taken directly from table)



f) **V: M = 4.0 Earth Masses**
   (1) (Run additional simulations if time permits)

g) *Note: Additional Graphs and Full Results can be found in the CMAPPER_Results Archive backup folder, ordered chronologically.*

C. Planetary Radii Across Three Orders of Magnitude in Mass and Stellar Isolation: Application to Transits (Fortney et al. 2006): Notes and Links to Useful Context:
   1. Abstract Notes:
      a) The aim of this paper is to model different planets (ranging from 0.01 Earth masses to 10 Jupiter masses and 0.02 to 10 AU in distance from their respective host stars) so that radii determined from light curves can be used to shed light on other planetary conditions.

IV. 7.30-8.10.25 Entry: Implementing Updated Iron Data, & Initial Work on Entropy Optimization in CMAPPER:

A. Reading, Graphing, & Implementation of Gonzales' Iron EoS Tables:
   1. Testing in Jupyter Notebook using Rob's Code (sent by email as rock_props.py):
      a) I loaded Rob's file into a new notebook (named Initial_Rock_Tests.LaMons.ipynb) and first changed the path for the EoS data to the following: C:\Users\usame\Documents\CMAPPER_rock-main\CMAPPER_rock-main\EoS\mantle
         (1) The path, however, had to be implemented in Unix, so I added an r before the ‘ and the path, as well as a \\ at the end before the final ‘.
         (2) I then had to download Zhang's iron EoS files and place them at this path: C:\Users\usame\Documents\CMAPPER_rock-main\CMAPPER_rock-main\eos-main\zhang_eos
         (3) I also replaced all of the relative paths with absolute ones so that the code can be run from different locations and will encounter fewer errors [replaced all of the np.load and np.loadtxt calls with code such as np.loadtxt(path_in_project('EoS', 'mantle', 'solid_P.txt'))].

(4) I then encountered an error with regard to the **'Fe16Si_fischer.npz'** file, as I could not find it in the repository or on my downloaded copy. I realized that I had to clone the repository using the Anaconda prompt (see image below) rather than simply downloading the zip file.

```
Microsoft Windows [Version 10.0.22631.5624]
(c) Microsoft Corporation. All rights reserved.

(base) C:\Users\usame>git lfs install
Git LFS initialized.

(base) C:\Users\usame>git clone https://github.com/robtejada/eos.git
Cloning into 'eos'...
remote: Enumerating objects: 2425, done.
remote: Counting objects: 100% (446/446), done.
remote: Compressing objects: 100% (247/247), done.
remote: Total 2425 (delta 293), reused 262 (delta 199), pack-reused 1979 (from 2)
Receiving objects: 100% (2425/2425), 150.11 MiB | 31.34 MiB/s, done.
Resolving deltas: 100% (1258/1258), done.
Updating files: 100% (292/292), done.
Filtering content: 100% (194/194), 15.73 GiB | 27.46 MiB/s, done.

(base) C:\Users\usame>cd eos

(base) C:\Users\usame\eos>git lfs pull

(base) C:\Users\usame\eos>
```

(5) I then emailed Rob, and he sent me the file, as it turned out to be a separate item altogether. However, it was likely still beneficial that I learned how to clone GitHub repositories properly.

b) Testing:

(1) get_Tmix_en_Py(100, 30) returned `12461.08592536879`, but did note this error:

```
get_Tmix_en_Py(100, 30)
```

```
C:\Users\usame\AppData\Local\Temp\ipykernel_44696\3833366369.py:95: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, a
nd will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
  return float(vals) if scalar else vals
```

(2) get_rhomix_Py(35, 10) returned `689.8076636512869`, and I was able to solve the above 'DeprecationWarning' by modifying the return statement of the get_rhomix_Py() function to be **return rho_out.item() if scalar else rho_out** so that it would no longer have difficulty if the output was an array.

(3) get_rho_FeSi_pt(25, 400) returned `7.075230235468603`, and it worked properly, as I modified the fix above to fit the get_rho_FeSi_pt() function (replaced the return statement with: return vals.item() if scalar else vals).

(4) Essentially, all this is doing is reading the input tables and returning a value for the requested variable based on the table by estimating (interpolating) with trends across said tables.

c) NOTE: **Interpolation**, and the graphs created here that are *interpolated*, refers to estimating unknown values on a graph based on known data sets before and after the unknown (provided there is a relationship, such as a linear one).

d) NOTE: a **Grüneisen parameter**, or gamma (**γ**), is a "dimensionless combination of familiar properties, expansion coefficient, bulk modulus, density, and specific heat" and allows for helpful relations, such as the adiabatic variation of temperature with density, between these quantities to be drawn.

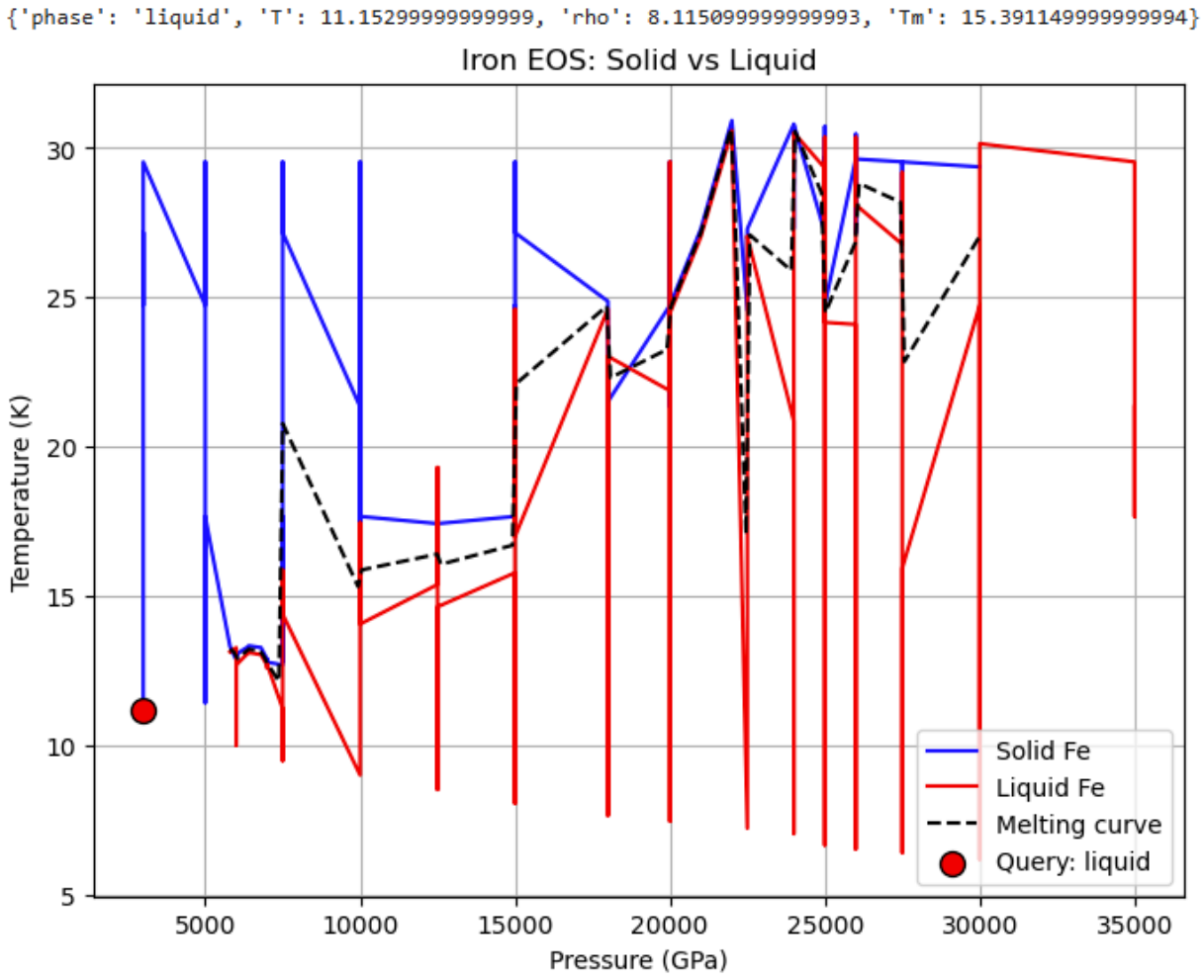
e) NOTE: **Radiogenic heat** is heat produced by the decay of radioactive elements within a planet. It plays a major role in a planet's internal heat production and is a contributing factor for volcanism and convection in the Earth's plastic mantle.

2. Reading, Writing Functions for, and Producing Graphs with the updated Gonzales Iron EoS tables:
   a) Written in a new Jupyter file named GonzalesIron_EoS_Tests.LaMons.ipynb, I first began by calling the necessary directories and setting up the modular file call function (path_in_project()) in a similar fashion to my modified version of Rob's code.
   b) First, I attempted to read and write interpolated functions for the liquid_eos.dat file, updating file paths as necessary (NOTE: Gonzales' iron EoS tables can be found at **C:\Users\usame\eos\gonzales_iron_eos**), but encountered some small ValueErrors (see image) during reading.

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[4], line 57
     54     return density_from_pressure, temp_from_pressure
     56 # Example usage:
---> 57 density_func, temp_func = load_liquid_eos(r"C:\Users\usame\eos\gonzales_iron_eos\liquid_eos.dat")
     59 P_test = 3500  # GPa
     60 print(f"Density at {P_test} GPa: {density_func(P_test):.3f} g/cc")

Cell In[4], line 32, in load_liquid_eos(filename)
     29 parts = line.split()
     31 # Extract relevant columns
---> 32 rho = float(parts[6])   # rho[g/cc]
     33 temp = float(parts[8])  # T[K]
     34 P = float(parts[10])    # P[GPa]

ValueError: could not convert string to float: 'rho[g/cc]='
```

c) But other than that minor issue, it functioned well, and so I moved to writing code for both liquid and solid states of iron and added a function to produce and label plots at given pressure values (in GPa). After sorting out the few syntax and formatting errors, I also had to add a **removes_duplicates()** function and implement it *before interpolation* to prevent the other error of dividing by 0 that I received.

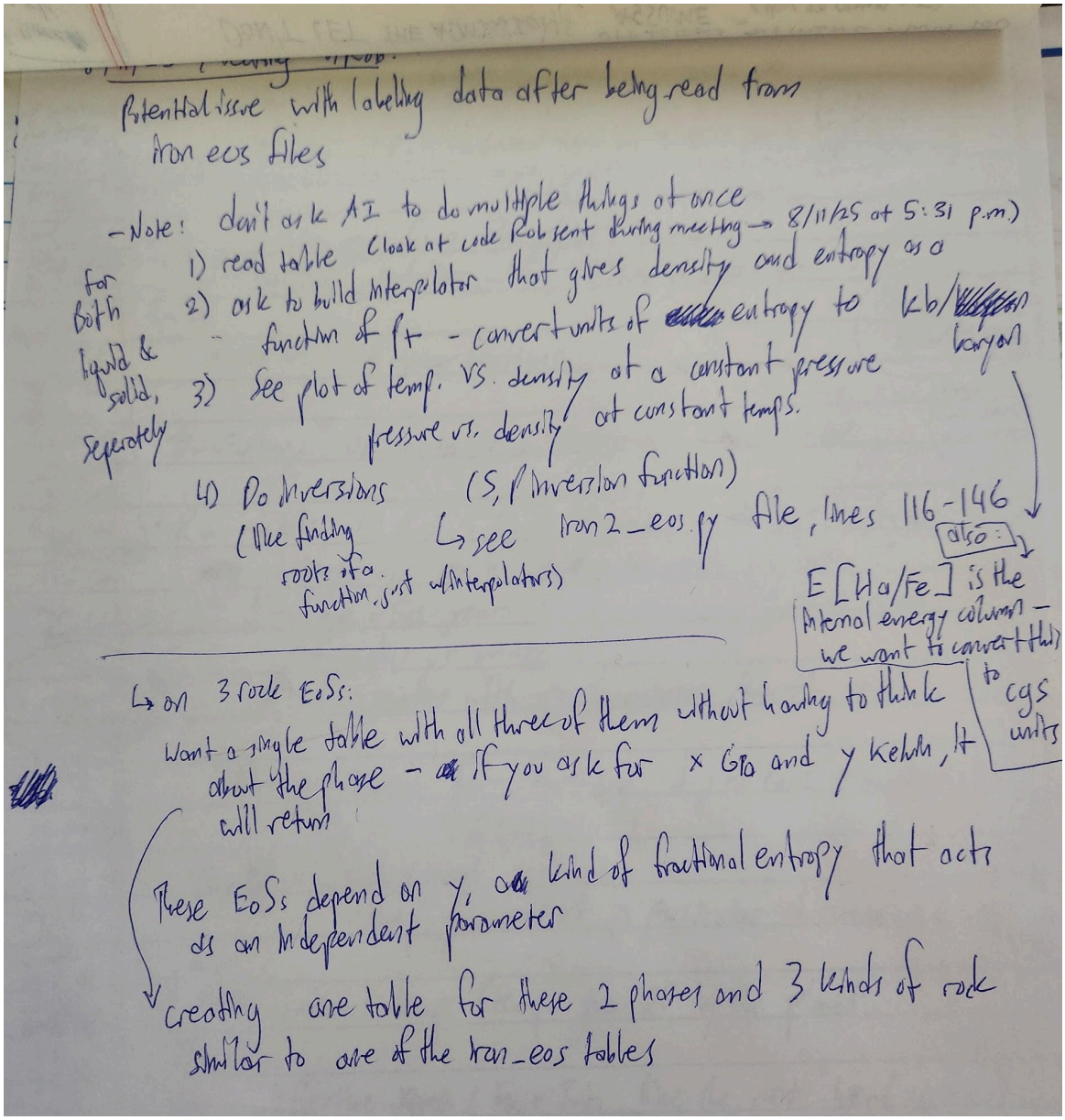
d) Once I did that, I tested the functions for P = 3000 GPa, and it produced the following graph with my point in question on said graph:



e) If I can improve the functionality of this algorithm, then I can implement it into CMAPPER to better the overall model. Doing this work also helped familiarize me with the code of this model and expanded my coding abilities.

V. 8.11-8.20.25 Entry: Secondary Trials with Iron Data Processing/Implementation & Research into Combining Rock EoS Tables:

A. Notes from 8.11.25 Meeting:



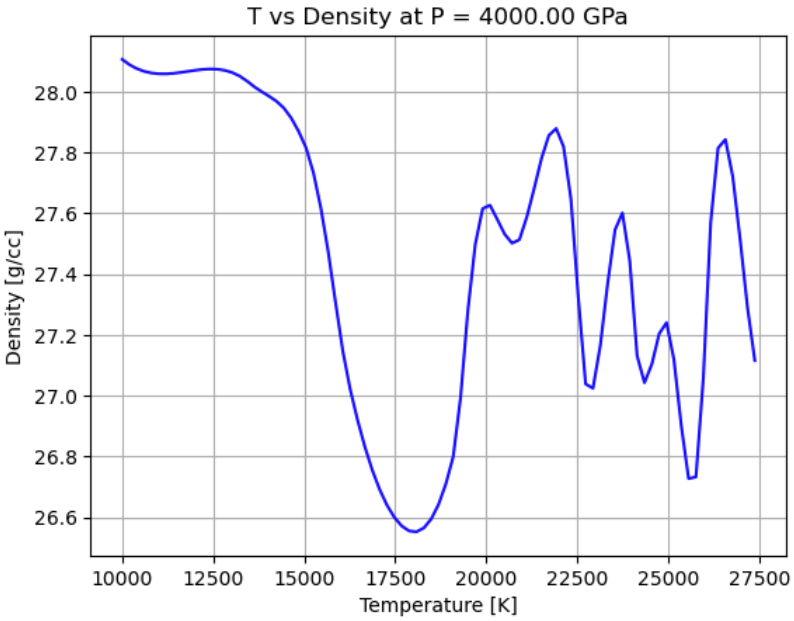
B. Attempt II: Interpreting and Graphing Gonzales' Iron EoS Tables:

1. First, I attempted to run the code Rob generated (a copy of which is below) during our meeting to read the tables, following the step-by-step approach we discussed.

```
import re
import pandas as pd
from typing import List, Dict

class IronTDITableReader:
    """
    Reads the ab initio iron TDI dataset from Gonzalez-Cataldo & Militzer (2023)
    and parses it into a pandas DataFrame.
    """

    # regex to match each data line
    _line_pattern = re.compile(
        r'^(?P<ID>\S+)\s+'
        r'(?P<isotope>\S+)\s+N=\s+(?P<N>\d+)\s+'
        r'V\[A\^3\]=\s+(?P<V>[\d\.]+)\s+'
        r'rho\[g/cc\]=\s+(?P<rho>[\d\.]+)\s+'
        r'T\[K\]=\s+(?P<T>\d+)\s+'
        r'P\[GPa\]=\s+(?P<P>[\d\.]+)\s+(?P<P_err>[\d\.]+)\s+'
        r'E\[Ha/Fe\]=\s+(?P<E>[-\d\.]+)\s+(?P<E_err>[\d\.]+)\s+'
        r'F_DFT\[Ha/Fe\]=\s+(?P<F>[-\d\.]+)\s+(?P<F_err>[\d\.]+)\s+'
        r'S\[kB/atom\]=\s+(?P<S>[-\d\.]+)\s+(?P<S_err>[\d\.]+)\s+'
        r'#\s+(?P<note>.+)$'
    )

    def __init__(self, filepath: str):
        self.filepath = filepath

    def read(self) -> pd.DataFrame:
        """Reads the file and returns a DataFrame of parsed values."""
        data: List[Dict] = []
        with open(self.filepath, 'r') as f:
            for line in f:
                line = line.strip()
                if not line or line.startswith("#"):
                    continue
                match = self._line_pattern.match(line)
                if match:
                    row = match.groupdict()
                    # convert numeric fields
                    for key in row:
                        if key not in ["ID", "isotope", "note"]:
                            row[key] = float(row[key])
                    data.append(row)
                else:
                    print(f"Warning: Could not parse line: {line}")

        return pd.DataFrame(data)

# Example usage:
reader = IronTDITableReader("eos/gonzales_iron_eos/solid_eos.dat")
df = reader.read()
print(df.head())
```

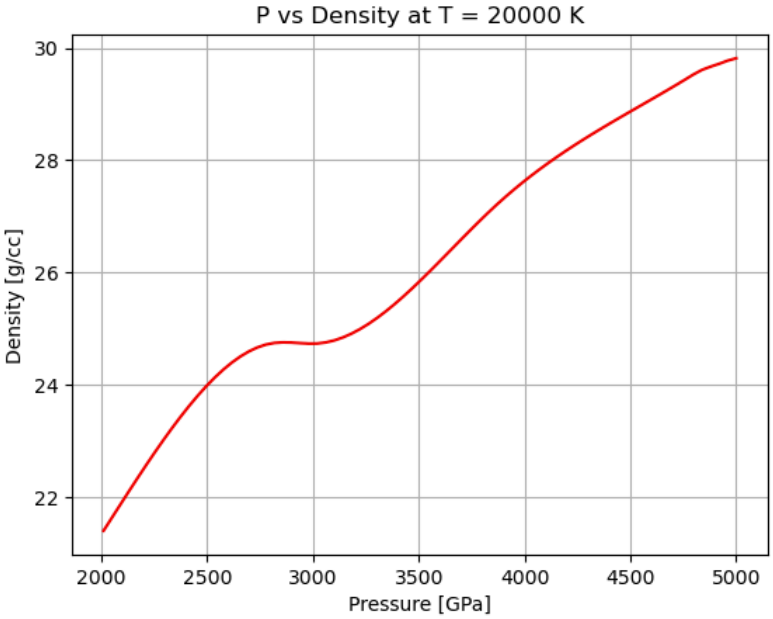
2. I checked the results of running the code with the solid_eos.dat file, and they were aligned. Note that, until stated otherwise, the subsequent notes here are for working with the solid iron EoS table.

3. In generating the interpolator, I also added code to convert the entropy into units of kB/baryon and the energy per atom of iron into cgs units.
    a) NOTE: The cgs unit for the internal energy of an atom is "ergs," which converts from Joules at a rate of $1\ erg\ =\ 1 \cdot 10^{-7}\ Joules$
    b) NOTE: The conversion rate for entropy (S) from kB/atom Fe to kB/Baryon is $1\frac{k_B}{Baryon} = \frac{k_B}{atom} \cdot \frac{1}{56}$, as there are 26 protons and 30 neutrons in one atom Fe, totalling 56 Baryon (three-quark) particles per atom.
4. After setting the file paths, the interpolator appeared to work correctly, outputting the following values for an input of 4000 GPa and 20000 K using the rho_interp() and the s_interp() functions:

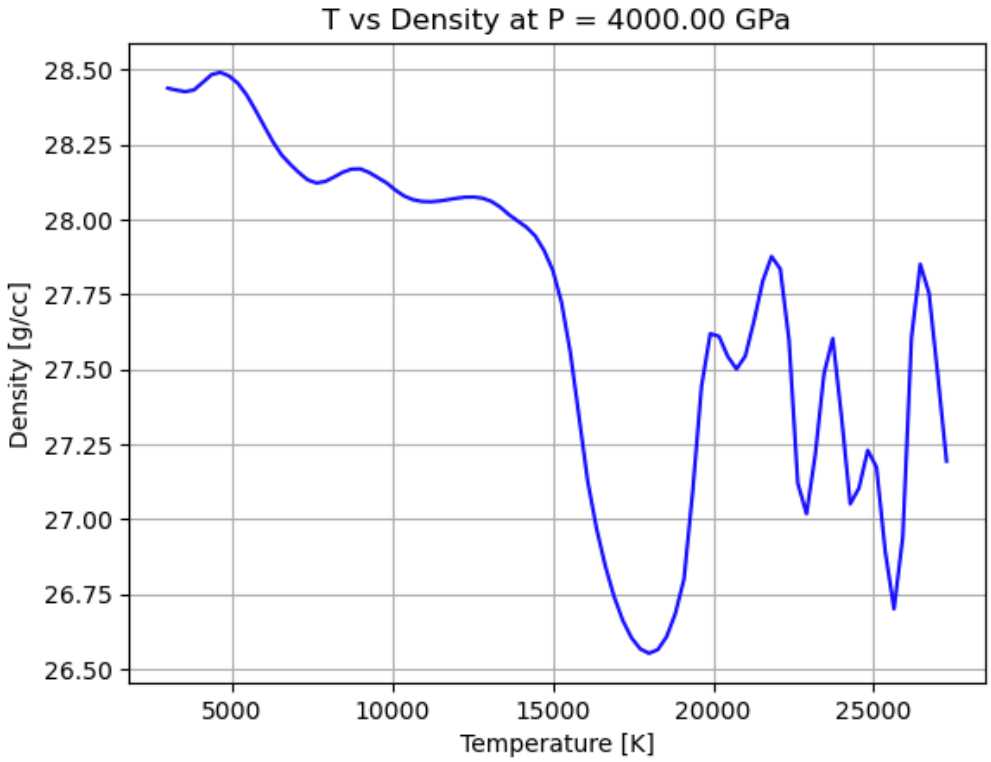


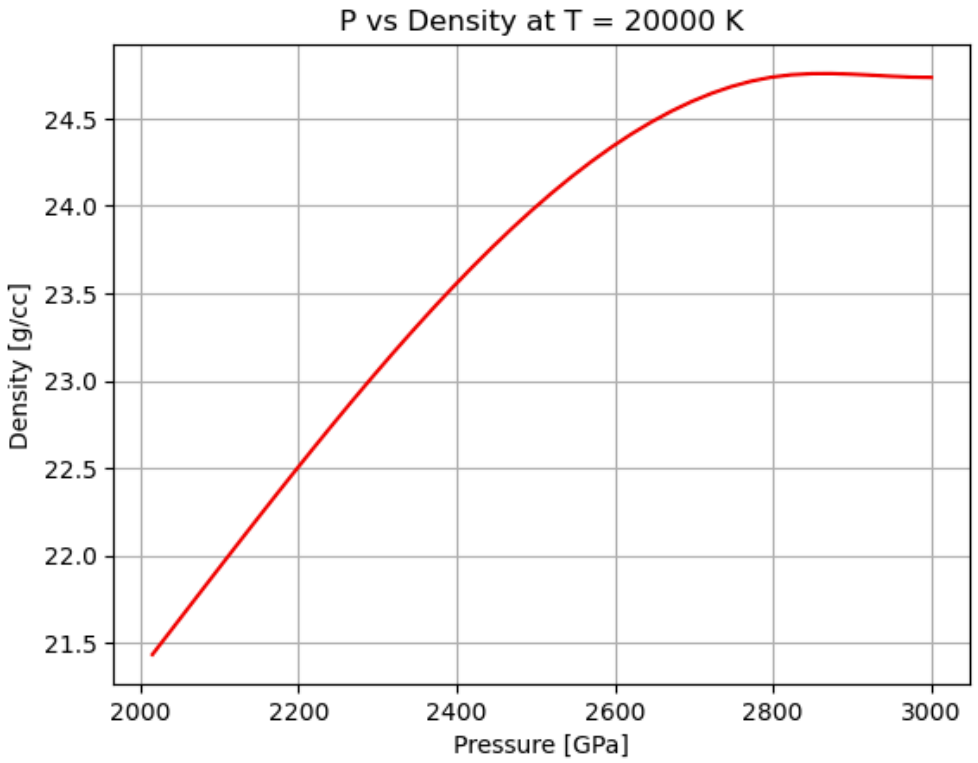

```
rho = 27.634117687557772
g/cc
S = 0.20802109616617126
kB/baryon
```

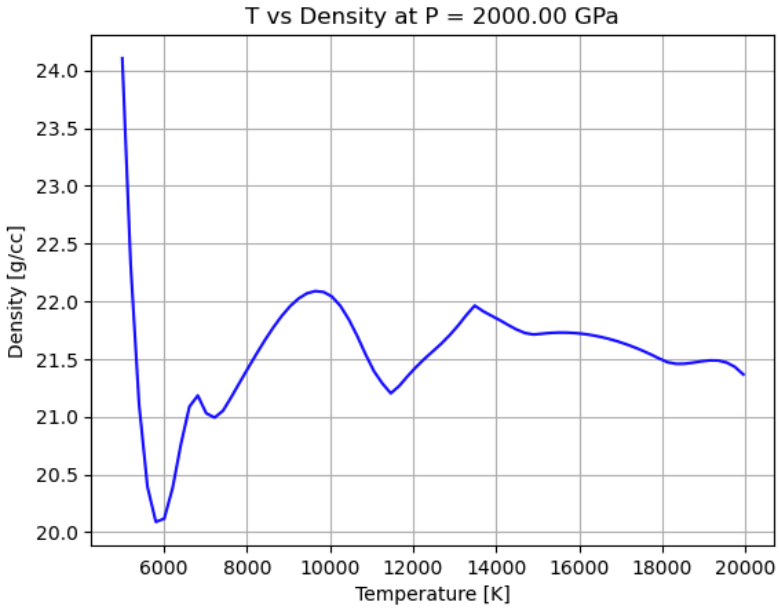
5. I then added several graphing functions, first for temperature vs. density and pressure vs. density (**plot_solidEoS_temperature**() and **plot_solidEoS_pressure**(), respectively), and produced the following initial plots:



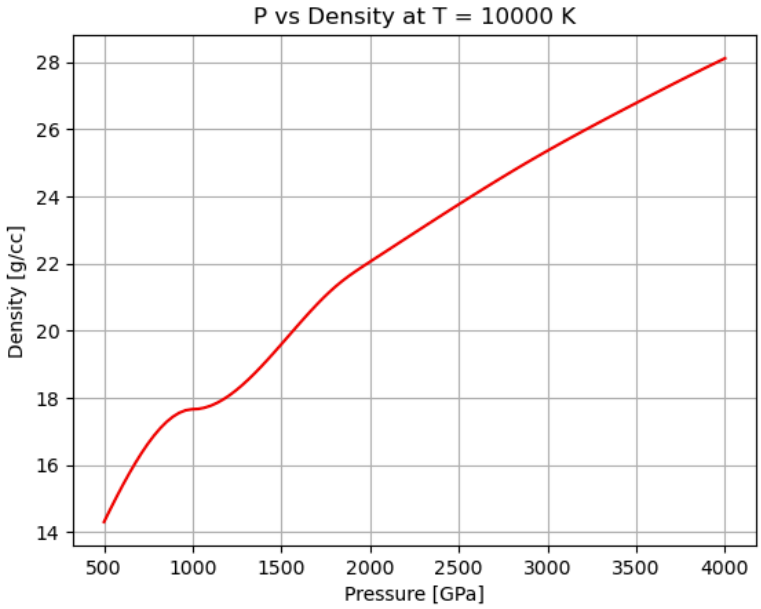
First, with constant pressure of 4,000 GPa and constant temperature of 20,000 K, respectively. I extended the range in different directions with the same constant numbers for the next two graphs.


T vs Density at P = 4000.00 GPa
Density [g/cc]
28.50
28.25
28.00
27.75
27.50
27.25
27.00
26.75
26.50
5000
10000
15000
20000
25000
Temperature [K]


P vs Density at T = 20000 K
Density [g/cc]
24.5
24.0
23.5
23.0
22.5
22.0
21.5
2000
2200
2400
2600
2800
3000
Pressure [GPa]



Second, with constant pressure of 2,000 GPa and constant temperature of 10,000 K, respectively.



6. Then, I attempted to create the S, P inversion function for this data based upon the one found after line 146 in the iron2_eos.py file on Rob's GitHub.
    a) NOTE: In order to do this, I needed to import an additional package using the following line: *from scipy.optimize import newton, brentq*.
    b) However, the first time I ran the inversion function, it returned this:
    `Recovered T = nan K`

(1) To remedy this issue, I first checked the boundaries of entropy data at the test value of P = 4000 GPa, which I found to be:

```
Entropy range at 4000 GPa: 0.18027977720780938 0.31448106625923267
```

(2) The test entropy value of S = 0.27 kB/Baryon is within this range, so I then checked if it was an initial guess problem (i.e., if brentq was failing because the sign of the function was not changing over the interval).

(a) Upon running the check, the cell returned this:

```
err(Tmin) = -0.1521326558528881
        err(Tmax) = nan
```

(b) I had located the issue and thus needed to decrease my temperature range from the initial 5000-40000 K to 20000-30000 K. When I tried to run the inversion with the new range, it still failed to produce a value, so I switched the CloughTocher2DInterpolator function out for the LinearNDInterpolator one.

(c) This did not work either, so I switched back to the original interpolator and added a dynamic bounds adjustment code to the inversion to make it NaN-safe. Yet, I still encountered errors. The following is my current inversion code as it stands:

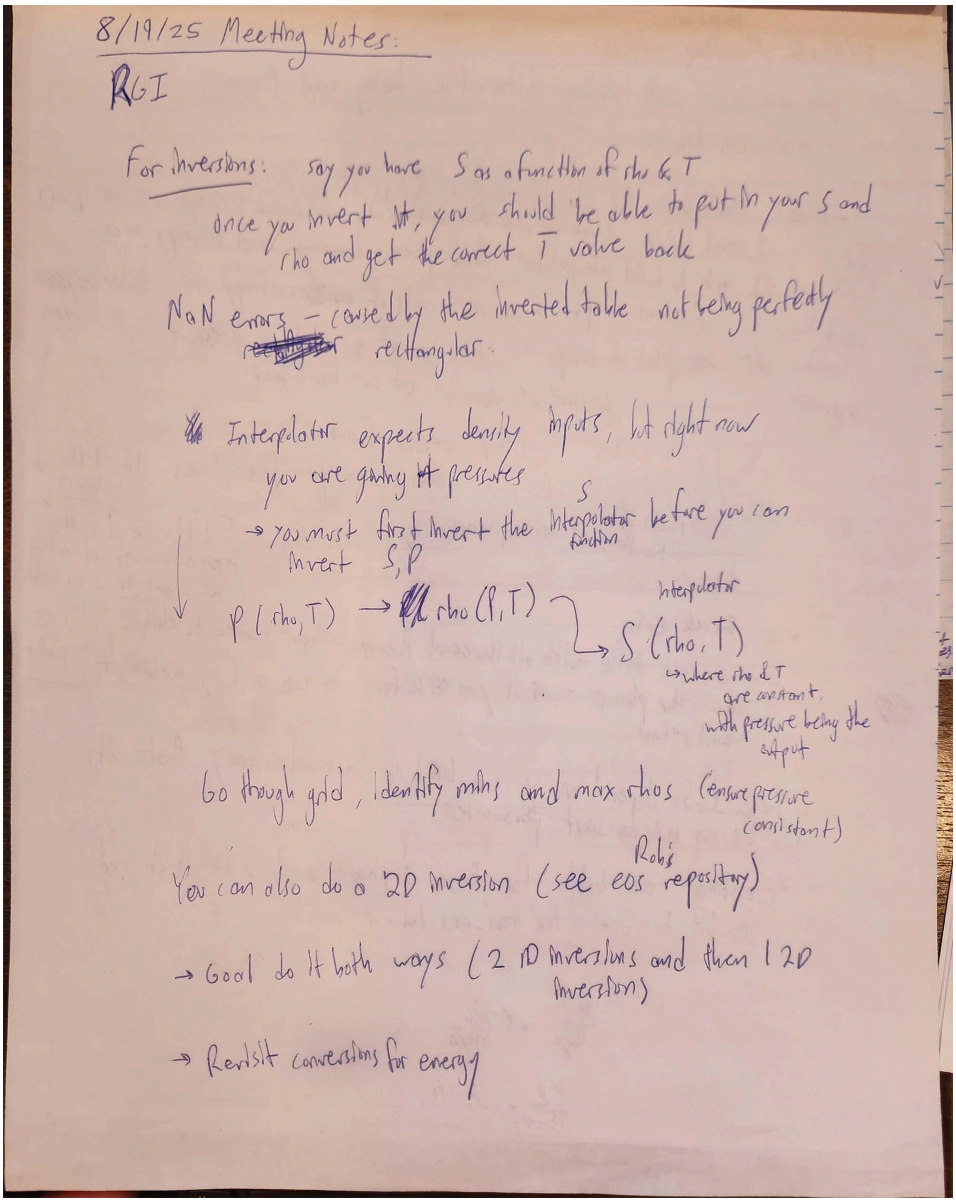
```
def get_logt_sp_inv_safe(s_target, p_target, guess_T=None, method='newton_brentq'):
    """
    Invert S(P,T) to recover T at given S, P using dynamic bracketing.

    Parameters
    ----------
    s_target : float
        Target entropy (kB/baryon or consistent units).
    p_target : float
        Pressure value (same units used in interpolator).
    guess_T : float, optional
        Initial guess for temperature [K]. If None, mid-point of valid interval is used.
    method : str
```

```
    Root-finding method: 'newton', 'brentq', or 'newton_brentq'.

Returns
-------
T : float
    Recovered temperature in Kelvin, or np.nan if no root is found.
"""

# --- Step 1. Find valid T range where s_interp is finite ---
def valid_T_bounds(P, Tmin=5_000, Tmax=26_000, step=500):
    Tgrid = np.arange(Tmin, Tmax+step, step)
    Svals = [s_interp(P, T) for T in Tgrid]
    finite_mask = np.isfinite(Svals)
    if not np.any(finite_mask):
        raise ValueError(f"No valid T range found at P={P}")
    Tmin_valid = Tgrid[finite_mask][0]
    Tmax_valid = Tgrid[finite_mask][-1]
    return Tmin_valid, Tmax_valid

Tmin, Tmax = valid_T_bounds(p_target)
if guess_T is None:
    guess_T = 0.5*(Tmin + Tmax)

# --- Step 2. Define residual function ---
def err(T):
    sval = s_interp(p_target, T)
    if not np.isfinite(sval):
        return np.nan
    return sval - s_target

# --- Step 3. Root-finding ---
try:
    if method == 'newton':
        return newton(err, x0=guess_T, tol=1e-5, maxiter=100)
    elif method == 'brentq':
        return brentq(err, Tmin, Tmax, xtol=1e-5, maxiter=100)
    elif method == 'newton_brentq':
        try:
            return newton(err, x0=guess_T, tol=1e-5, maxiter=100)
        except RuntimeError:
```

```
        return brentq(err, Tmin, Tmax, xtol=1e-5, maxiter=100)
    else:
        raise ValueError("Invalid method. Choose 'newton', 'brentq', or 'newton_brentq'.")
except Exception:
    return np.nan
```

VI. 8.20-9.12.25 Entry: Fixing Inversion Functions with Correct Order & Dual Methods:

A. Notes from 8.19.25 Meeting:



B. First Inversion Method: Two 1D Inversions with the S Interpolator:

1. Based upon the first set of notes after the NaN errors section in the photograph above. Thus, I went ahead and inverted the S interpolator to be P(S, T), or *pressure as a function of entropy and temperature*, and inverted the rho interpolator to be P(rho, T), or *pressure as a function of density and temperature* (see below).
    a) P(S, T):
        (1) def PST_invert(s_target, T_fixed, P_min, P_max, s_interp):
        (2) def f(P):
        (3) return s_interp(P, T_fixed) - s_target
        (4) #pick a sensible bracketing range - IMPORTANT (applies below as well)
        (5) return brentq(f, P_min, P_max)
    b) P(rho, T):
        (1) def PRhoT_invert(rho_target, T_fixed, P_min, P_max, rho_interp):
        (2) def f(P):
        (3) return rho_interp(P, T_fixed) - rho_target
        (4) return brentq(f, P_min, P_max)

2. Then, in order to get entropy as a function of density and temperature as discussed, I used the inverted density inversion within a new function: S(P(rho, T), T) = S(rho, T), as shown:

```python
#Component 2: Entropy as a function of density and temperature [S(rho, T), which effectively is S(P(rho, T), T)]
def SRhoT_2ndInvert(rho_target, T_fixed, rho_interp, s_interp,
                    P_min=None, P_max=None):
    """
    Return entropy at given (rho, T) using rho(P,T) inversion.
    """
    # Use global min/max if not provided
    if P_min is None: P_min = np.min(pressures)
    if P_max is None: P_max = np.max(pressures)

    # Step 1: solve for P such that rho(P,T)=rho_target
    P_sol = PRhoT_invert(rho_target, T_fixed,
                                    P_min, P_max, rho_interp)
    # Step 2: evaluate s(P,T)
    return s_interp(P_sol, T_fixed)
```

3. However, this code resulted in the following error: `ValueError: The function value at x=5105.893 is NaN; solver cannot continue.`
    a) This was due to the fact that the density value chosen was outside the minimum and maximum values at the given pressure–boundaries we discussed the importance of during our meeting.

b) So, I modified the S(rho, T) function to filter out NaNs and to check if the chosen density was within the range for the selected temperature:

```
def SRhoT_2ndInvert(rho_target, T_fixed,
                    rho_interp, s_interp,
                    P_all, n_bracket=50):
    """
    Compute entropy at (rho, T) by:
       1. finding P such that rho(P,T)=rho_target
       2. evaluating s(P,T)

    Parameters
    ----------
    rho_target : float
       Desired density [g/cc].
    T_fixed : float
       Temperature [K].
    rho_interp : callable
       rho(P,T) interpolator.
    s_interp : callable
       s(P,T) interpolator.
    P_all : 1D array
       The array of all pressure values from your table (used for bracketing).
    n_bracket : int
       Number of pressure samples for initial search.

    Returns
    -------
    s_val : float
       Entropy [kB/baryon].
    """

    # Step 1: Get pressure bounds from your table
    P_min = float(np.min(P_all))
    P_max = float(np.max(P_all))

    # Step 2: Sample rho along P at fixed T
    P_grid = np.linspace(P_min, P_max, n_bracket)
    rho_grid = np.array([rho_interp(P, T_fixed) for P in P_grid])
```

```
# Step 3: Remove NaNs (outside convex hull)
mask = ~np.isnan(rho_grid)
P_grid = P_grid[mask]
rho_grid = rho_grid[mask]

if len(rho_grid) < 2:
    raise ValueError("No valid (P,T) points found at this temperature.")

# Step 4: Ensure target rho is within range
if not (rho_grid.min() <= rho_target <= rho_grid.max()):
    raise ValueError(
        f"rho_target={rho_target} is outside "
        f"[{rho_grid.min()}, {rho_grid.max()}] at T={T_fixed}"
    )

# Step 5: Find a local bracket where rho crosses rho_target
for i in range(len(P_grid)-1):
    if (rho_grid[i]-rho_target)*(rho_grid[i+1]-rho_target) <= 0:
        P_lo, P_hi = P_grid[i], P_grid[i+1]
        break

# Step 6: Root-solve for exact pressure
def f(P): return rho_interp(P, T_fixed) - rho_target
P_sol = brentq(f, P_lo, P_hi)

# Step 7: Evaluate entropy at that (P,T)
return s_interp(P_sol, T_fixed)
```

c) This new function, when rho is outside the minimum and maximum range, returns a useful error in the following format:

```
rho_target=4.0 is outside [11.4246, 29.418504705650186] at T=3000.0
```

d) It includes the range at that temperature, allowing me to change the rho appropriately.

e) When I then changed the density input at T=3000 K to rho = 12.0, the code returned an entropy value, `1.6738e-01 kB/baryon`, without errors. Thus, I was able to successfully use a two-step process to create a S(rho, T) function.

Maximum and minimum Temperatures, respectively:

```
30000.0
3000.0
```

Maximum and minimum pressures, respectively:

```
5105.893
143.396
```