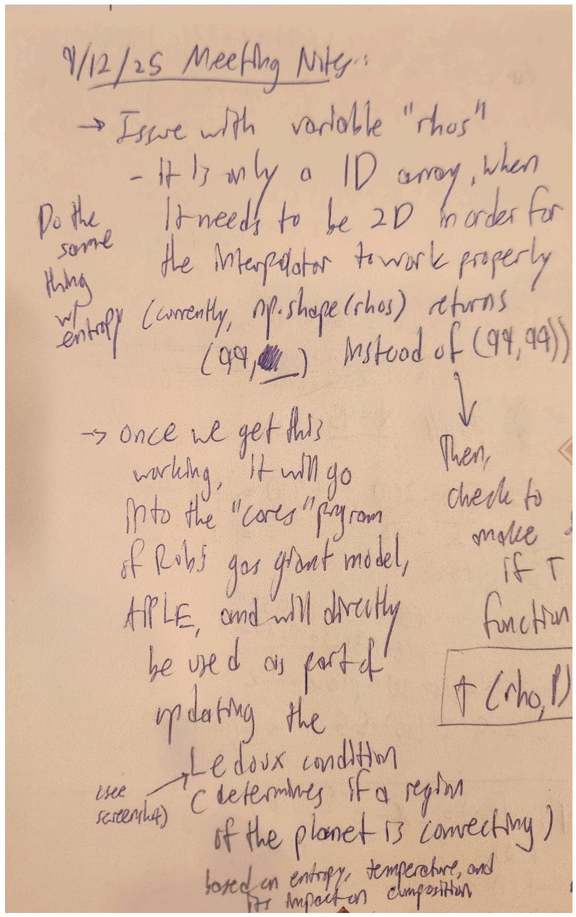
SRP III - Project Journal

Brian LaMons

October 2025 - Present

I. 9.13-10.8.2025 Entry: Mending Rho/Entropy List Errors:

A. Notes on 9.12.25 Meeting:



B. Fixing 2D Rho & Entropy Array Creation:

1. Goal: At the moment, as we discussed during our most recent meeting, the way in which the program reads the EoS table produces a 1D array for densities of iron, when it should create and fill a 2D array.
   a) In doing so, I took the opportunity to replace the current function, CloughTocher2D interpolator, with a regular grid interpolator (RGI), as that would allow for greater efficiency and produce the desired 2D arrays for rho and for entropy (ideally).

2. Re-coding the parse_eos_table(filename) function for RGIs and 2D rho and entropy arrays resulted in the following:

```python
def parse_eos_table(filename):
    pattern = re.compile(
        r"rho\[g/cc\]=\s*([\d.]+)\s*T\[K\]=\s*([\d.]+)\s*P\[GPa\]=\s*([\d.]+).*?S\[kB/atom\]=\s*([\d.]+)"
    )

    data = []
    with open(filename, "r") as f:
        for line in f:
            match = pattern.search(line)
            if match:
                rho, T, P, S = map(float, match.groups())
                data.append((P, T, rho, S))

    # Convert to DataFrame for sorting and reshaping
    df = pd.DataFrame(data, columns=["P", "T", "rho", "S"]).sort_values(["P", "T"])

    pressures_unique = np.sort(df["P"].unique())
    temps_unique = np.sort(df["T"].unique())

    nP, nT = len(pressures_unique), len(temps_unique)
    if nP * nT != len(df):
        print("[Warning] The EoS data is not a perfect rectangular grid — filling missing cells with NaN.")

        # Use pivot with reindex to handle missing points safely
        rho_df = df.pivot(index="P", columns="T", values="rho").reindex(index=pressures_unique, columns=temps_unique)
        S_df = df.pivot(index="P", columns="T", values="S").reindex(index=pressures_unique, columns=temps_unique)

        rho_grid = rho_df.to_numpy()
        S_grid = S_df.to_numpy()
    else:
        rho_grid = df.pivot(index="P", columns="T", values="rho").to_numpy()
        S_grid = df.pivot(index="P", columns="T", values="S").to_numpy()

    return pressures_unique, temps_unique, rho_grid, S_grid
```

3. Making this change also meant that I had to modify my build interpolators function with the new variables (updated to indicate shape) and the RGI component functions:

```python
def build_RGI_interpolator(pressures_unique, temps_unique, rho_grid, S_grid):
    """
    Build RegularGridInterpolators from 2D EoS tables.
    """
    rho_interp = rgi((pressures_unique, temps_unique), rho_grid, bounds_error=False, fill_value=None)
    S_interp = rgi((pressures_unique, temps_unique), S_grid, bounds_error=False, fill_value=None)
    return rho_interp, S_interp
```

4. Once I implemented these modifications, I no longer got a 1D array for the shape of rhos (now rhos_grid); instead, I got (99, 21) and an error indicating that because the EoS table is not a perfect rectangular grid, I

was not able to get a clear density and entropy value for T = 20000 K and P = 4000 GPa (which I verified are within the ranges found last time).

a) To fix this, I further modified the parse_eos function to resample the (99, 21) grid to a regular (99, 99) grid, which fixed the NaN error the code was returning when I requested density and entropy at T = 20000 K and P = 4000 GPa.
   (1) To do so, I brought back the CloughTocher2D interpolator code and utilized that on the raw data as part of the new build_RGI function (see below):

```python
def build_RGI_interpolator(pressures, temps, rhos, entropies, nP=99, nT=99, fill_value=None):
    """
    Build RegularGridInterpolators for rho(P,T) and S(P,T),
    automatically resampling the scattered EoS data onto
    a uniform (nP x nT) rectangular grid.

    Parameters
    ----------
    pressures, temps : 1D arrays
        Raw EoS pressure [GPa] and temperature [K] data.
    rhos, entropies : 1D arrays
        Corresponding density [g/cc] and entropy [kB/atom] data.
    nP, nT : int
        Desired grid resolution in pressure and temperature.
    fill_value : float or None
        Value to use for extrapolation (default None → extrapolate).

    Returns
    -------
    rho_interp, s_interp : RegularGridInterpolator objects
        Interpolators for rho(P,T) and S(P,T).
    P_grid, T_grid : 1D arrays
        Uniform pressure and temperature grids used for interpolation.
    rho_grid, S_grid : 2D arrays
        Rectangular 2D data arrays used to build the interpolators.
    """

    #Define uniform grid ranges based on data
    P_min, P_max = np.min(pressures), np.max(pressures)
    T_min, T_max = np.min(temps), np.max(temps)
    P_grid = np.linspace(P_min, P_max, nP)
    T_grid = np.linspace(T_min, T_max, nT)
    PP, TT = np.meshgrid(P_grid, T_grid, indexing='ij')

    #Build smooth scattered interpolators for rho and S
    rho_scattered = CloughTocher2DInterpolator(
        np.column_stack([pressures, temps]), rhos
    )
    S_scattered = CloughTocher2DInterpolator(
        np.column_stack([pressures, temps]), entropies
    )

    #Evaluate these on the regular grid
    rho_grid = rho_scattered(PP, TT)
    S_grid = S_scattered(PP, TT)

    #Fill any NaNs (at edges or sparse regions)
    if np.isnan(rho_grid).any():
        mean_rho = np.nanmean(rho_grid)
        rho_grid = np.where(np.isnan(rho_grid), mean_rho, rho_grid)
        print(f"[Info] Filled {np.isnan(rho_grid).sum()} NaN cells in rho grid.")

    if np.isnan(S_grid).any():
        mean_S = np.nanmean(S_grid)
        S_grid = np.where(np.isnan(S_grid), mean_S, S_grid)
        print(f"[Info] Filled {np.isnan(S_grid).sum()} NaN cells in S grid.")

    #Build RegularGridInterpolators
    rho_interp = RegularGridInterpolator(
        (P_grid, T_grid), rho_grid, bounds_error=False, fill_value=fill_value
    )
    s_interp = RegularGridInterpolator(
        (P_grid, T_grid), S_grid, bounds_error=False, fill_value=fill_value
    )

    print(f"[Success] Built RegularGridInterpolators on {nP}×{nT} grid.")
    print(f"ρ grid shape: {rho_grid.shape}, S grid shape: {S_grid.shape}")

    return rho_interp, s_interp, P_grid, T_grid, rho_grid, S_grid
```

(2) After that, I applied the RG interpolators and was able to get a 2D rho array and a 2D entropy array with shape (99, 99).

(3) I tested it with the following code:

```
# 10/6/25 2D Rho & S Array Testing:
print("Rho grid shape:", rho_grid.shape)
print("S grid shape:", S_grid.shape)
print("Rho NaN count:", np.isnan(rho_grid).sum())

P_test, T_test = 4000, 20000
print("Rho =", rho_interp((P_test, T_test)), "g/cc")
print("S =", S_interp((P_test, T_test)), "kB/atom")
```

5. After some minor debugging, cleaning the code of *SyntaxError: invalid non-printable character U+00A0* using the script contained in the lower cell of the 2.1 version of the program on Jupyter (and seen in the screenshot below),

[22]:

```
infile = "GonzalesIron_EoS_Tests.LaMons2.1.Solid.ipynb"       # change to your filename
outfile = "GonzalesIron_EoS_Tests.LaMons2.1.Solid_clean.ipynb"

with open(infile, "r", encoding="utf-8") as f:
    text = f.read()

# Remove ALL invisible or non-printable Unicode chars (except tabs and newlines)
text_clean = re.sub(r"[^\S\n\t]|[\u200B-\u200F\u202A-\u202E\u2060\uFEFF]", " ", text)

with open(outfile, "w", encoding="utf-8") as f:
    f.write(text_clean)

print(f"✅ Cleaned and saved to: {outfile}")
```

6. …and changing some variable names, the code ran completely without errors and returned the following results (which is what we wanted!):

```
[Info] Filled 0 NaN cells in rho grid.
[Info] Filled 0 NaN cells in S grid.
[Success] Built RegularGridInterpolators on 99×99 grid.
ρ grid shape: (99, 99), S grid shape: (99, 99)
Rho grid shape: (99, 99)
S grid shape: (99, 99)
Rho NaN count: 0
Rho = 27.61305631052566 g/cc
S = 11.762255804195782 kB/atom
```

II. <u>**10.9-10.24.25 Entry: Testing T(S, P) Function, Researching the Ledoux Criterion, & Addressing Continued Interpolator Issues:**</u>

A. <u>Notes on 10.8.25 Meeting:</u>

1. At present, there is still a slight issue with the way the interpolators are functioning, causing some NaNs and errors to occur when one attempts to use them. To remedy this, I am going to re-write the interpolation portion of my code to use the interp1D SciPy function so that I can break down the process into smaller steps that we can then troubleshoot individually in order to better solve any issues that continue / arise.
2. In addition, I am continuing work on my temperature as a function of entropy and pressure [T(S, P)] program, as that is necessary to determine if convection is occurring at any given time.

B. <u>Creating the T(S, P) function & looking into the Ledoux criterion:</u>

1. In order to check when the Ledoux criterion is satisfied, we need to be able to find what the temperature is given certain entropy and pressure values.

2. In order to fully understand the goal of this segment of the project, I researched the [Ledoux Criterion], finding that it is a principle which determines the "stability of a stellar layer against convection." Here, I am applying it for a core of a mixture of liquid and solid iron, but the principle remains the same. Specifically, it applies for regions of varying chemical composition, which works here due to the fact that I will ultimately combine the work with the iron EoS tables that I have been doing with a model of different phases of other rocks. The regions between those different rocks and the iron core could potentially convect, changing the magnet

3. Thus, I wrote the following T(S, P) function:

   a) def T_of_SP(P_target, S_target, s_interp, T_bounds=(1000, 50000)):
   b) """
   c) Compute temperature T given pressure P and entropy S using the EOS interpolator.
   d)
   e) Parameters
   f) ----------
   g) P_target : float

h) Pressure in GPa.
i) S_target : float
j) Entropy (kB/baryon).
k) s_interp : RegularGridInterpolator
l) Interpolator giving S(P, T).
m) T_bounds : tuple, optional
n) Lower and upper temperature bounds (K) for the root finder.
o)
p) Returns
q) -------
r) float
s) Temperature (K) where S(P, T) = S_target.
t) Returns np.nan if no valid solution found in bounds.
u) """
v) T_min, T_max = T_bounds
w)
x) # Define the residual function
y) def f_T(T):
z) return s_interp((P_target, T)) - S_target
aa)
bb) # Check that the entropy actually crosses the target within the bounds
cc) try:
dd) f_lo, f_hi = f_T(T_min), f_T(T_max)
ee) if np.isnan(f_lo) or np.isnan(f_hi) or f_lo * f_hi > 0:
ff) # No sign change or invalid range
gg) return np.nan
hh)
ii) T_sol = brentq(f_T, T_min, T_max, xtol=1e-3)
jj) return T_sol
kk)
ll) except ValueError:
mm) return np.nan

4. When I tested this, I encountered some small errors with syntax of functions and of variables, as well as one issue with how the brentq function is implemented within the above code.