Stellar Modelling

Andy Chmilenko

(Dated: Friday April 17, 2015)

I. CODE DISCUSSION & ALGORITHMS

In this section I will discuss some of the algorithmic choices made for our stellar modelling solution as well as talk about some caveats associated with these decisions as well as their possible solutions. This project is done according to the Project Outline[1] and solves the equations outlined therein.

All code is available at (`https://github.com/Lanowen/StellarModelling`).

Preface

When choosing which language to write the code in, we first began using Python since most of the people in the group were comfortable with it. However it quickly became apparent how slow Python was at parsing and executing the code-base we produced, so we decided to transfer the code-base to C++ to give us the added performance we needed. Using C++, the speed of our code increased several fold, allowing us to produce solutions at very high fidelity in a fraction of the time the Python code took to find a solution.

Numerical Integration

A. Theory

For our integration method we decided to use an adaptive step numerical integration method, the Runge-Kutta-Fehlberg[2] method. This method is allows for easy calculation of a 4th-order and 5th-order accurate methods with only one calculation of k-values (the increments), this this method is usually referred to the RKF45 method for brevity.

Using its Butcher tableau, as seen in Table I, once $k_1 - k_6$ have been calculated, getting the 4th-order and 5th-order step solutions is trivially done by multiplying the k-values by the specified coefficients.

	k_1	k_2	k_3	k_4	k_5	k_6
0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	−7200/2197	7296/2197			
1	439/216	−8	3680/513	−845/4104		
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	
O(h^5)	16/135	0	6656/12825	28561/56430	−9/50	2/55
O(h^4)	25/216	0	1408/2565	2197/4104	−1/5	0

TABLE I: RKF45 Butcher tableau

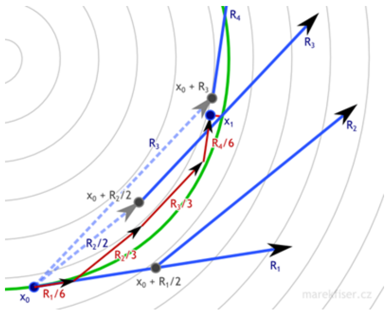

FIG. 1: RK4 step visualization[3]

With this we can find the error associated with the current step-size (h) then adjust the step-size accordingly to achieve the desired precision for the integration step. This gives a huge advantage in calculation time for stellar solutions to reduce the number of integration steps needed drastically while still achieving a satisfactory amount of precision. The step-size can be made more coarse for parts of the integration with small slopes and smaller error, and more fine for parts of integration that change drastically.

The first k value represents a normal Euler integration step as seen in Fig.1 (where R-values are the k-values mentioned in code and theory), while the others are corrections based on y coordinates calculated from each previous k step. When they are all combined they give a well approximated solution from the DE.

B. Coupled Differential Equations and the RK4 class

```
inline bool RK4::updateK() {
   switch (currK) {
   case 1: updatingK = true; intermed_y = y; k1 = f(t, y)*step;   break;
   case 2:  intermed_y = y + k1 / 4.0; break;
   case 3: k2 = f(t + step / 4.0, intermed_y)*step; break;
   case 4:  intermed_y = y + k1 * 3.0 / 32.0 + k2 * 9.0 / 32.0;   break;
   case 5: k3 = f(t + step * 3.0 / 8.0, intermed_y) * step; break;
   case 6: intermed_y = y  + k1*1932.0/2197.0 – k2*7200.0/2197.0 + k3*7296.0/2197.0; break;
   case 7: k4 = f(t + step*12.0/13.0, intermed_y)*step; break;
   case 8:  intermed_y = y + k1*439.0 / 216.0 – k2*8.0 + k3*3680.0 / 513.0 – k4*845.0 / 4104.0; break;
   case 9:  k5 = f(t + step, intermed_y)*step; break;
   case 10: intermed_y = y – k1*8.0 / 27.0 + k2*2.0 – k3*3544.0 / 2565.0 + k4*1859.0 / 4104.0 – k5*11.0 / 40.0; break;
   case 11: k6 = f(t + step / 2.0, intermed_y)*step;break;
   default: updatingK = false; currK = 1; return false;
   }
   currK++;
   return true;
}

inline void RK4::iterate() {
   y += k1*25.0 / 216.0 + k3*1408.0 / 2565.0 + k4*2197.0 / 4104.0 – k5 / 5.0;
   t += step;
}

inline long double RK4::get() {
   if (updatingK)
      return intermed_y;

   return y;
}

inline void Star::step() {
   //calculate k values before iteration
   do {
      temperature.solver.updateK();
      density.solver.updateK();
      luminosity.solver.updateK();
      mass.solver.updateK();
   } while (tau.solver.updateK());

   //iterate rk4
   temperature.iterate();
   density.iterate();
   luminosity.iterate();
   mass.iterate();
   tau.iterate();
}
```

Using classes makes life easier, less code to write, generally faster compile times, and easier debugging. However when working with coupled DE's there are some design choices that need to be made properly to do the integration properly. All the k-values need to be calculated in step with each other, which can be done with this style of loop in `Star::step()`, and the structure of `RK4::updateK()`. The first loop will update k_1 for all the DE's, and the second loop will update the intermediate y-values, subsequent pairs of loops will update the next k and intermediate y values, until it gets to the 12th and last loop in which all the functions will return `false`, and it will move on to the step of iterating the primary y value using all k-values with the proper coefficients from Table I.

Caveats

1. *Intermediate y-values*

It is important to note, you need return the intermediate y-values while the k-values are being calculated instead of the current y-values (the y-value before the `RK4::iterate()` step) which represent that quantity (Mass, Luminosity, etc...), otherwise you lose the accuracy built into the RK4 and end up with a fancy Euler's method. Fig.1 illustrates the purpose of this, but it is not immediately obvious that you also need to do this between coupled DE's.

C. The Adaptive Step

```
vector<double long> relErrors;

while (true) {
   relErrors.clear();
   push();
   step();

   relErrors.push_back(temperature.solver.getRelError());
   relErrors.push_back(density.solver.getRelError());
   relErrors.push_back(luminosity.solver.getRelError());
   relErrors.push_back(mass.solver.getRelError());
   //relErrors.push_back(tau.solver.getRelError());

   long double err_bound = err_sensitivity;
   long double relerr = LDBL_EPSILON;

   for (int i = 0; i < relErrors.size(); i++) {
      if (!isnan(relErrors[i]) && relErrors[i] > relerr) {
         relerr = relErrors[i];
      }
   }

   long double lastStep = RK4::step;
   RK4::step = max(step_min, min(RK4::step*pow(err_bound / 2.0 / relerr, 0.25), step_max));

   if (relerr > err_bound && RK4::step != lastStep && RK4::step != step_min) {
      pop();
      continue;
   }
}

pushValues();
```

After taking a step and before saving your values, you need to collect whatever function errors you want to have your desired error accuracy with and adjust your step accordingly. We took looked for the largest error out of the set of functions we were looking at, in this case we wanted accuracy in our Temperature, Density, Luminosity, and Mass calculations, and using this error and our error bound adjust the step-size. You would make the step-size smaller if your relative error is larger than your desired error bound, giving more accurate steps, or you would make your step-size larger if your relative error was less than your error bound, giving less required steps to find a solution.

One thing to note is if your relative error is above your error bound, you should return your values to what they were before the initial step and recalculate your step with the new, smaller step-size. You should do this until your step-size hits a minimum that you've specified, or until your relative error is within acceptable limits.

Caveats

1. Error

I think it makes more sense to talk about relative error in a function rather than just the truncation error

```
inline long double getError() {
      return abs((k1*16.0/135.0 + k3*6656.0/12825.0 + k4*28561.0/56430.0 – k5*9.0/50.0 + k6*2.0/55.0) – (k1*25.0 / ↩
          216.0 + k3*1408.0 / 2565.0 + k4*2197.0 / 4104.0 – k5 / 5.0));
   }

   inline long double getRelError() {
      long double yi = k1*25.0 / 216.0 + k3*1408.0 / 2565.0 + k4*2197.0 / 4104.0 – k5 / 5.0;
      long double zi = k1*16.0 / 135.0 + k3*6656.0 / 12825.0 + k4*28561.0 / 56430.0 – k5*9.0 / 50.0 + k6*2.0 / 55.0;

      return abs((zi – yi) / yi);
   }
```

Especially when you are dealing with function such as Luminosity which typically has values of $> 10^{26}$, it will grow much faster than the other functions initially, giving you truncation errors of > 1 between 4th-order and 5th-order solutions, when its relative error could be $< 0.001\%$.

2. *Bounded Step-Size*

Some problems we ran into with the adaptive step size is sometimes the step-size would be made so big, that the next iteration step could produce some really strange and unwanted results. For example the step could be so big it may return `INF` values, or produce `NaN` errors. One thing to remember is that variables are in fixed parts of the memory and using a step-size that is too big can produce buffer overflow errors, so it is a good idea to bound your step-size.

Similarly, you may run into an unattainable error accuracy, which can make your step-size so small it could become truncated to 0, or you solution would iterate endlessly until you run out of available memory. So having a minimum step-size is also a good idea.

Finding Solutions

We parametrized the solutions we were looking for in terms of variable ρ_c and a fixed T_c and by varying ρ_c we were able to find a solution where the structure of the star converges. By comparing the calculated Luminosity from the Stefan-Boltzmann Law (Eq.4) using the solved T_{surf} of the star as seen in Eq.1, with the integrated solution for Luminosity we knew how we needed to vary ρ_c in order to converge.

$$f(\rho_c) = \frac{L_* - 4\pi\sigma R_*^2 T_*^4}{\sqrt{4\pi\sigma R_*^2 T_*^4 L_*}} \tag{1}$$

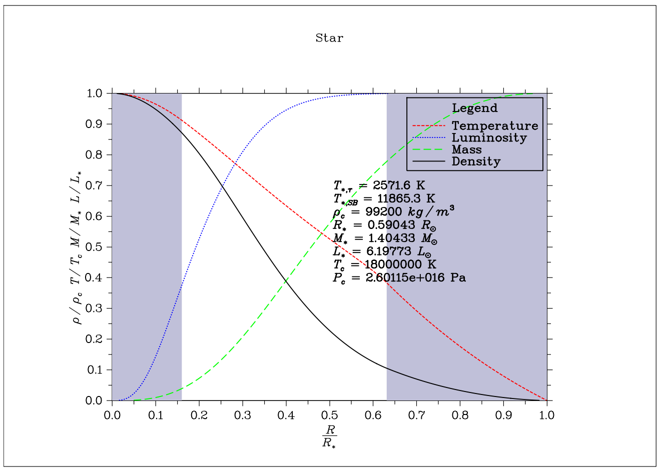

FIG. 2: Solution for star with $T_c = 1.8 \times 10^7 K$. $\rho_{c,test}$ is larger than $\rho_{c,sol}$, we are above target solution. Temperature rapidly converges. Convection zones shown in grey.

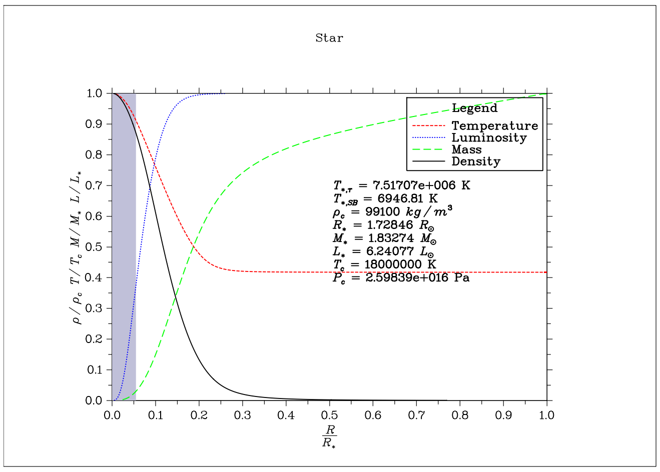

FIG. 3: Solution for star with $T_c = 1.8 \times 10^7 K$. $\rho_{c,test}$ is smaller than $\rho_{c,sol}$, we are below target solution. Temperature doesn't converge. Convection zones shown in grey.

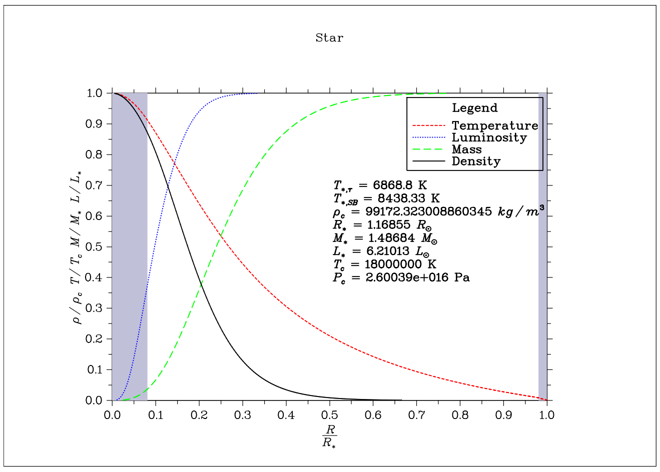

FIG. 4: Solution for star with $T_c = 1.8 \times 10^7 K$. Solution converged on proper ρ_c. Convection zones shown in grey.

When looking for a solution we will have two solutions, one that is a ρ_c above our solution (Fig.2), and one that is ρ_c below our solution (Fig.3). When we have a solution like in Fig.2, the Temperature converges rapidly giving a lower T_{surf} and a radius R_* smaller than the radius would be for the proper solution (Fig.4), we get a solution for the Stefan-Boltzmann Law that has a luminosity far less than the integrated luminosity L_* giving a *positive* fractional difference for Eq.1. When we have a solution like in Fig.3, the temperature doesn't converge, leading to a high temperature for T_{surf}. In this case we get a solution for the Stefan-Boltzmann Law that has a luminosity far higher than the integrated luminosity L_* giving a *negative* fractional difference for Eq.1.

Using this information we can use the Shooting Method to find a set of bounds to be able to do our Bisection Method to zero in on the proper ρ_c solution quickly and efficiently.

D. Shooting method

Not knowing where the solution for ρ_c for any particular T_c condition can make finding the solution difficult, but by using the Shooting Method we can coarse adjust our $\rho_{c,test}$ by some large amount. If we are above the solution we adjust our new $\rho_{c,test}$ down, and vice-versa if we are below the solution.

```
double frac = star->frac_diff();

if (frac > 0 && !isinf(frac)) { //positive, go lower
   R_lim = star->R_star / Rsun;
   if (frac_test == 1 && !isnan(frac) && !isinf(frac)) {
      bisect = true;
      cout << endl << endl << "===Beginning bisection===" << endl;
      break;
   }
   else {
      if (!isnan(frac) && !isinf(frac))
         frac_test = 2;
      rho_c_1 = rho_c;
      if (rho_c - shoot_delta_density < 0)
         rho_c /= 2;
      else
         rho_c -= shoot_delta_density;
   }
}
else{ //negative, go higher
   if (frac_test == 2 && !isnan(frac) && !isinf(frac)) {
      bisect = true;
      cout << endl << endl << "===Beginning bisection===" << endl;
      break;
   }
   else {
      if (!isnan(frac) && !isinf(frac))
         frac_test = 1;
      rho_c_1 = rho_c;
      rho_c += shoot_delta_density;
   }
}
```

As we can see, after we solve a star, we can calculated the fractional difference (Eq.1) and we can keep track of where are are relative to the solution. If we are above the solution and the next "shot" we are below the solution, we know we passed over the solution and we can begin bisection between our two last $\rho_{c,test}$'s. Like-wise we can do the same if we go from below the solution to above it. Some speed enhancements we can do, is with the introduction of a limiting radius, `R_lim`.

E. Integration Stopping Conditions

We implemented three checks to stop the integration process when solving a star.

```
while (kappa.get()*pow(density.get(), 2) / abs(density.dRho_dr(density.arr[0].back(), density.get())) > LDBL_EPSILON &&↩
      temperature.arr[0].back() < int_R_stop*Rsun && mass.get() < 1E3*Msun) {
   this->iterate();
}
```

Where `*.arr[0].back()` is the last integrated radius value in metres.

The first way, is by using an opacity proxy, defined by Eq.2.

$$\tau(\infty) - \tau \approx \delta\tau \equiv \frac{\kappa\rho^2}{\mid d\rho/dr \mid} \tag{2}$$

We know that the structure has converged (or is nearly converged) to some solution when the $\delta\tau \ll 1$. This is a convenient and is easily calculated.

Other way is a simple Mass limit, for our solutions to be physical solution of the main sequence the mass should be limited by a reasonable amount $M < 10^3 M_\odot$. In practice this rarely, if ever, is the terminating condition of the integration (with regular values).

The third way we've implemented the stopping condition is with a limiting radius. Generally for MS stars, they have a maximum radius of $\approx 10R_\odot$. To begin with, we can set `R_lim` to be about 10. This becomes helpful for solutions in the regime below the proper solution. $\delta\tau$ will stay large, and this radius limit becomes the primary stopping condition. If we are in the regime above the proper solution, the structure converges to a radius that is smaller than the proper radius, but we can infer that the proper radius is close to the calculated R_*, we can set `R_lim = star->R_star / Rsun` as seen in the Shooting Method code. However to make sure you don't stop your integration prematurely, we implemented `R_lim` of the star to be set to `R_lim_star = 1.2L*R_lim + 1.0L` when defining the new star to test. Heuristically this worked well and improved our calculation times immensely.

F. Bisection Method

```
if (frac > 0) { //positive, go lower
down:
    lastDir = 2;
    R_lim = R_lim;
    cout << endl << endl << "Bisecting down." << endl;
    rho_c_1 = min(rho_c_1, rho_c_2);
    rho_c_2 = rho_c;
}
else{ //negative, go higher
up:
    lastDir = 1;
    cout << endl << endl << "Bisecting up." << endl;
    rho_c_1 = max(rho_c_1, rho_c_2);
    rho_c_2 = rho_c;
}
```

The Bisection Method is remarkably similar to the shooting method, except it deals with the average between two ρ_c, and therefore converges closer to the proper solution exponentially. Every step the $\rho_{c,ave}$ is tested, `rho_c = (↩ rho_c_1 + rho_c_2)/ 2.0;`. Then the fractional difference is compared, if $\rho_{c,ave}$ is above the solution, the bounds become $\rho_{c,ave}$ and the *minimum* of the current bounding ρ_c, and vise-versa it if it below the solution, the bounds become $\rho_{c,ave}$ and the *maximum* of the current bounding ρ_c as seen in the code. Again, we can continue setting `R_lim` as we get closer to the solution for faster results.

Caveats

1. *Floating Point Precision*

One thing you need to be constantly aware of is the precision of data types in programming. The data occupies a finite spot in memory and thus has a limited precision. `long double` typically has an accuracy on the order of 10^{-16} (`LDBL_EPSILON` ≈ `2.2045e-016`), and star solutions become increasingly more sensitive w.r.t. the initial conditions. Assuming your initial ρ_c bounds are within 100, you will hit the precision limit after 56 bisections ($100/2^{56} = 6.94 \times 10^{-16}$). The higher T_c you try to test, the further off your solutions will be (generally). We also ran into the problem where the bisections gets stuck in a regime *below* the proper solution, as in Fig.5, temperature will never converge, so after 56 bisections you may not actually end up with the best solution, so it is good to keep track of the best solution as you bisect, then output the quantities you want from `Star last_pos_frac;` as it is your best solution.

```
if (last_pos_frac == 0 || abs(last_pos_frac->frac_diff()↩
    ) > abs(frac)) {
    if (last_pos_frac != 0)
        delete last_pos_frac;
    last_pos_frac = star;
    star = 0;
}
```

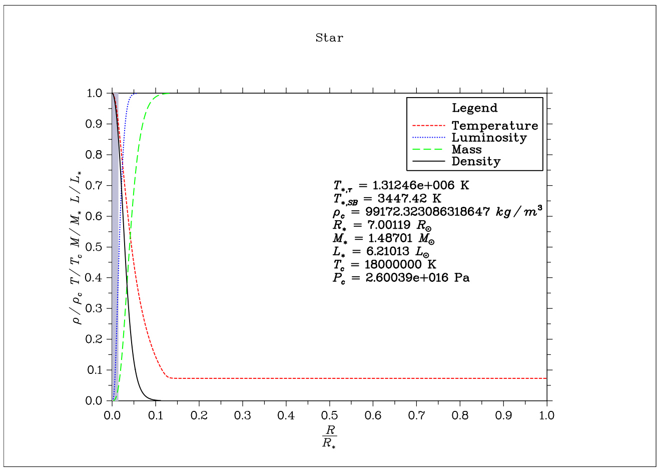

FIG. 5: Solution that is $\rho_c < 10^{-12} kg/m^2$ below the proper ρ_c, where Temperature doesn't converge. Convection zones shown in grey.

G. τ & Finding Surface Solutions

```
long double Tau_inf = this->tau.arr[1].back();
for (int i = this->tau.arr[1].size() - 1; i >= 0; i--) {
    if (Tau_inf - this->tau.arr[1][i] > 2.0 / 3.0) {
        T_star = this->temperature.arr[1][i + 1];
        R_star = this->temperature.arr[0][i + 1];
        break;
    }
}
```

$$\tau(\infty) - \tau(R_*) = \frac{2}{3} \tag{3}$$

$$L(R_*) = 4\pi\sigma R_*^2 T_*^4 \tag{4}$$

The surface boundary conditions are satisfied by Eq.3 and Eq.4, and it is pretty trivial to find it with a simple loop. Similarly using Eq.4 with integrated values for L_* and R_*, a solution can be found for T_*. Looking at the MS plot in Fig.11, both values for T_* are plotted against each other, the values for T_τ and T_{SB} agree quite well for lower temperature stars, then starts to deviate around 5000 K. However, there is one outlier around 28000 K where they both agree. That is why earlier, I said it is *generally* not possible to converge to a solution at higher temperatures with the precision in `long double`, but it so happens that we were lucky for this one case. Otherwise, the structure of the star converges, and R_* and L_* can be used to find the T_{surf} with the Stefan-Boltzmann Law and it correlates well to expected MS values.

H. κ & Opal Tables

Opal tables are a good way to calculate opacities for your stars, they are computed with many different sets of equations to take into account electromagnetic and quantum effects that give more fidelity than our approximate functions do.

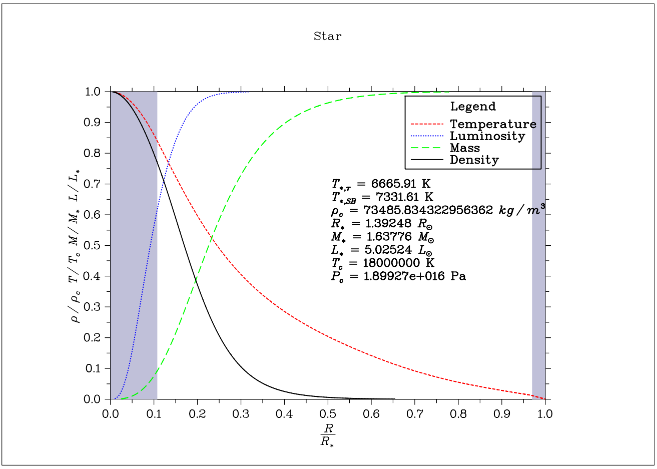

FIG. 6: Solution for star with $T_c = 1.8 \times 10^7 K$ using Opacity generated from Opal Tables.

The solution using Opal Tables converges for a ρ_c a bit smaller (Fig.6) than when using the fitted κ functions (Fig.4). You can see the variations that appear in κ from the Opal Tables comparing Fig.7 and Fig.9, as well as the variations in $\ln P/d\ln T$ comparing Fig.8 and Fig.10.

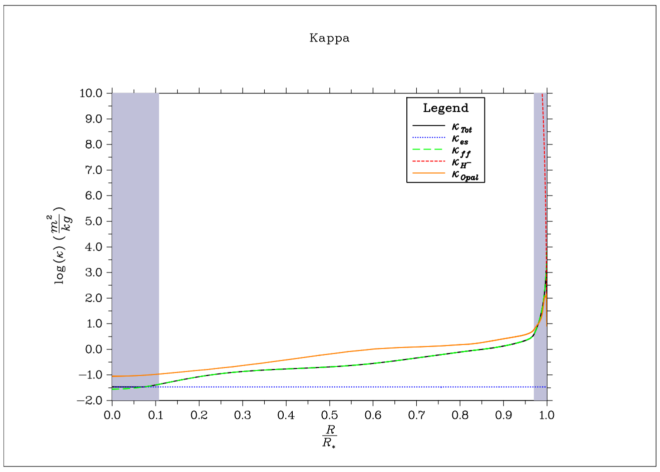

FIG. 7: Solution for star with $T_c = 1.8 \times 10^7 K$. κ vs R/R_* showing κ_{opal} as well as the fitted κ function

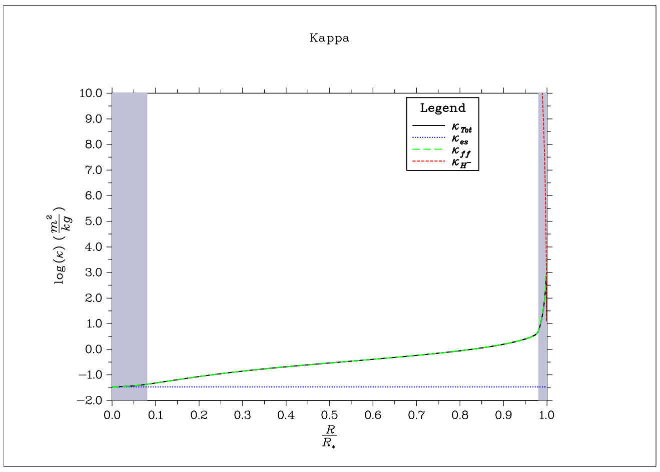

FIG. 9: Solution for star with $T_c = 1.8 \times 10^7 K$. κ vs R/R_* the fitted κ function

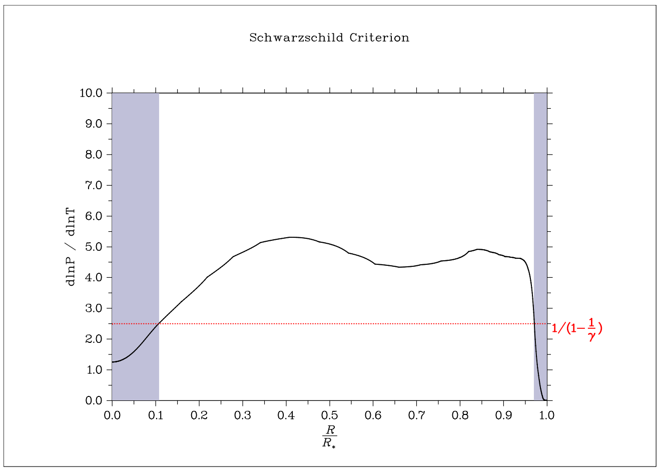

FIG. 8: Solution for star with $T_c = 1.8 \times 10^7 K$. $d \ln P / d \ln T$ variations from Opal Table values.

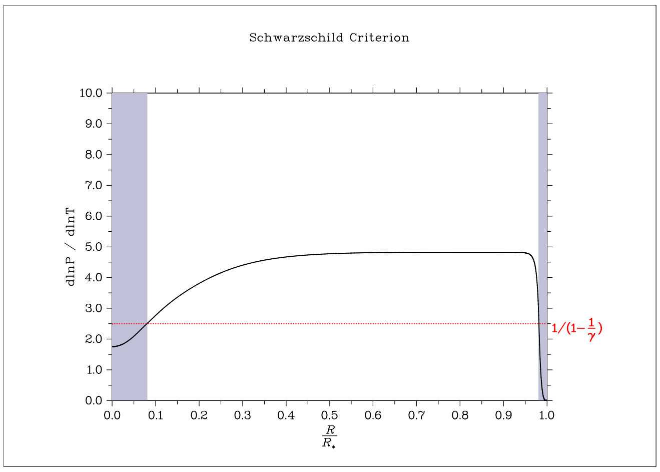

FIG. 10: Solution for star with $T_c = 1.8 \times 10^7 K$. Smooth $d \ln P / d \ln T$ from fitted κ function.

A simple and naïve approach to implementing the Opal Tables is using linear interpolation. The tables are organized with respect to two variables, with rows of constant $\log T$ and columns of constant $\log R$, where $R \equiv density[g/cm^3]/T_6^3$ and $T_6 \equiv T/10^6$. Using this you can find a space in the table in-between two columns and in-between two rows. First you start of by finding the fractional difference between your values of $\log T$ and $\log R$, and their respective neighbours.

			$\leftarrow$ `fR` $\longrightarrow$		
		$\log R_1$	$\leftarrow \log R_{calc} \longrightarrow$	$\log R_2$	$\cdots$
	$\log T_1$	`table[x][y]`	$\leftarrow$ `r1` $\longrightarrow$	`table[x][y+1]`	$\cdots$
$\uparrow$ `fT` $\downarrow$	$\uparrow \log T_{calc} \downarrow$		$\uparrow \kappa_{opal} \downarrow$		
	$\log T_2$	`table[x+1][y]`	$\leftarrow$ `r2` $\longrightarrow$	`table[x+1][y+1]`	$\cdots$
	$\vdots$	$\vdots$		$\vdots$	$\ddots$

TABLE II: Opal Table Linear Interpolation Visualization

```
fT = (T – table[x][0]) / (table[x + 1][0] – table[x][0]);
fR = (R – table[0][y]) / (table[0][y + 1] – table[0][y]);

r1 = table[x][y] + fR*(table[x][y + 1] – table[x][y]);
r2 = table[x + 1][y] + fR*(table[x + 1][y + 1] – table[x + 1][y]);

return pow(10.0, (r1 + fT*(r2 – r1)))/10.0; //divide by 10 to convert from cm^2/g to m^2/kg
```

Where `x` is the index of the column containing the greatest value of $\log T$ such that $\log T_{calc} > \log T_{table}$, and similarly `y` is the index of the row containing the greatest value of $\log R$ such that $\log R_{calc} > \log R_{table}$ and `table` is a array of rows, or in different words a matrix of [rows][colums]. Once you find the fractional differences `fT` and `fR`, you can use `fR` to calculate the two 'in-between' values `r1` and `r2` as shown in Table.II. Then the desired k_{opal} is one last linear interpolation between `r1` and `r2` using the fractional separation `fT`.

Caveats

1. Better Interpolation Methods

The Opal Opacity group has code written in Fortran which uses cubic spline interpolation and smoothing functions when looking up values in tables. As said before, the linear interpolation is a good start however cubic spline interpolation would be better, albeit harder to implement.

2. Edge-Cases and Extrapolation

You can run into problems when you are at the edge of the opal table, such that table[x+1] or table[x][y+1] will give you a sub-script out of range error. Similarly, the table isn't perfectly square, the bottom right corner has some empty values which would give you problems as well.

```
if ((x < table.size() – 1 && y > table[x + 1].size() – 1) || x == table.size() – 1) {
   fR = (R – table[0][y]) / (table[0][y + 1] – table[0][y]);
   fT = (T – table[x][0]) / (table[x – 1][0] – table[x][0]);
}
else if (y == table[x].size() – 1) {
   fR = (R – table[0][y]) / (table[0][y – 1] – table[0][y]);
   fT = (T – table[x][0]) / (table[x + 1][0] – table[x][0]);
}
else {
   fT = (T – table[x][0]) / (table[x + 1][0] – table[x][0]);
   fR = (R – table[0][y]) / (table[0][y + 1] – table[0][y]);
}

if (y == table[x].size() – 2 && y == table[x + 1].size() – 1) {
   r1 = table[x][y] + fR*(table[x][y + 1] – table[x][y]);
   r2 = table[x + 1][y] + fR*(table[x + 1][y] – table[x + 1][y–1]);

   }
else {
   r1 = table[x][y] + fR*(table[x][y + 1] – table[x][y]);
   r2 = table[x + 1][y] + fR*(table[x + 1][y + 1] – table[x + 1][y]);
}

return pow(10.0, (r1 + fT*(r2 – r1)))/10.0; //divide by 10 to convert from cm^2/g to m^2/kg
```

This code handles some of the edge cases, and allows for some rudimentary extrapolation (linear). This is far from ideal, perhaps something is wrong with either the extrapolation code (most likely), or with the star solutions that are driving the Opal values out of range of the pre-generated values. In either case, using the fitted κ functions proved to be more robust and less prone to error.

II. THE MAIN SEQUENCE

Some notes on the main sequence graph Fig.11, as mentioned in G, about finding surface solutions, the boundary conditions become increasingly more sensitive as the temperature increases; `long double` doesn't have enough precision to resolve solutions. T_{surf} solutions can be solved using either Eq.3, or Eq.4, the T_τ solutions line up with the T_{SB} solutions at lower temperatures almost perfectly, then start to deviate around 5000 K surface temperatures. However, it is interesting to note the one outlier at about 28,000 K, where the T_τ solution line up with the T_{SB} solution again, this is probably just a stroke of luck where ρ_c parameter was well approximated in the 10^{-16} precision of `long double`.

Plotted also is the Mass-Luminosity (Fig.12) and Mass-Radius (Fig.13) relationships along with the relationships from the text[4]. The calculated values are in good agreement with the fitted relationships from the text, with some slight deviation at the lower mass stars for the Mass-Radius relationship (Fig.13).

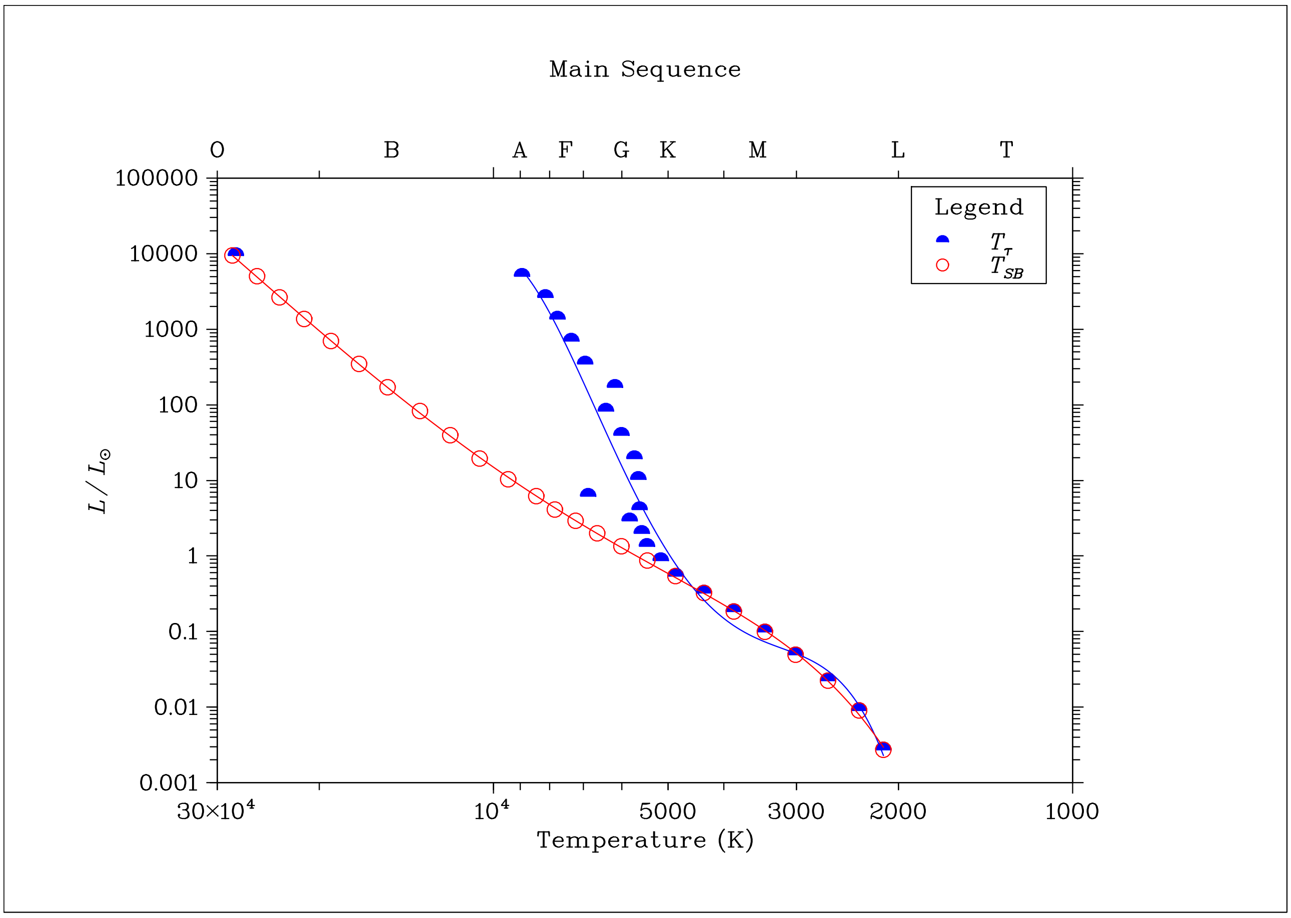

FIG. 11: Main Sequence Plots containing surface temperatures: T_τ calculated from τ, as well as T_{SB} calculated from the Stefan-Boltzmann Law (Eq.4).

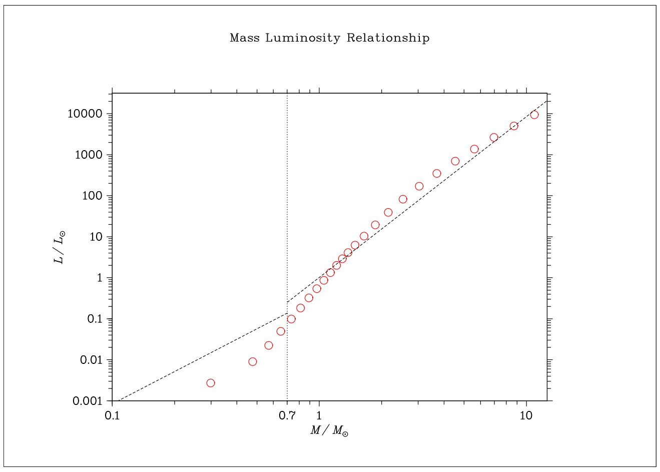

FIG. 12: Mass-Luminosity relationships from text[4] plotted against MS solutions.

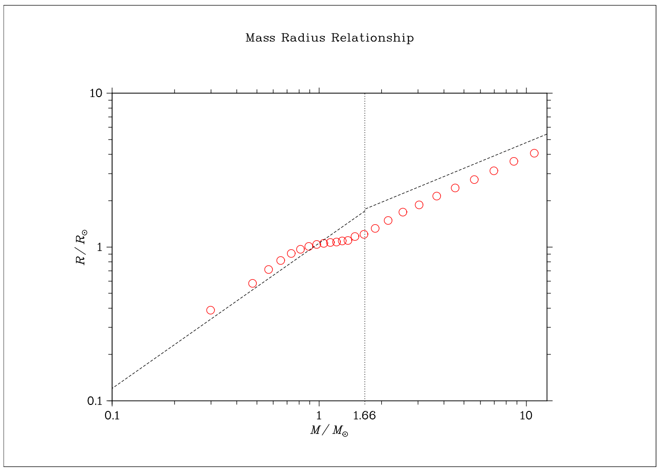

FIG. 13: Mass-Radius relationships from text[4] plotted against MS solutions.

III. A CLOSER LOOK AT TWO STARS

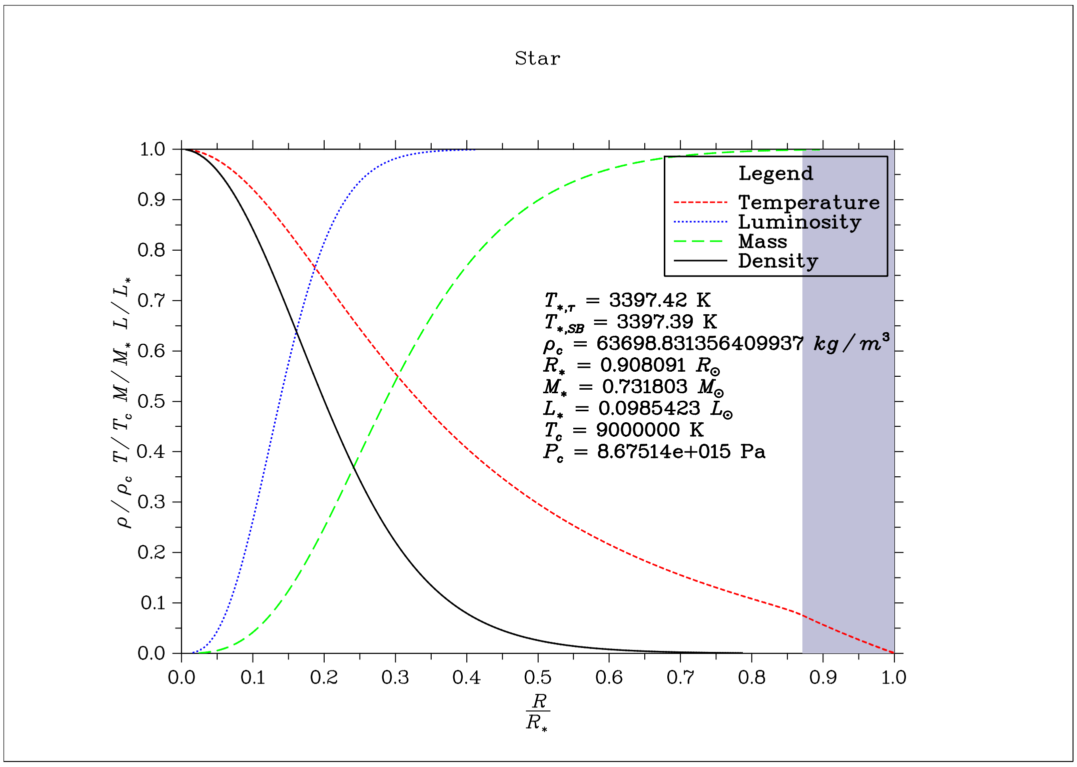

FIG. 14: Star with T_c of 9.0×10^6 K. ρ, T, M, L plotted as a function of $R/R_\odot$. Convection zones shown in grey.

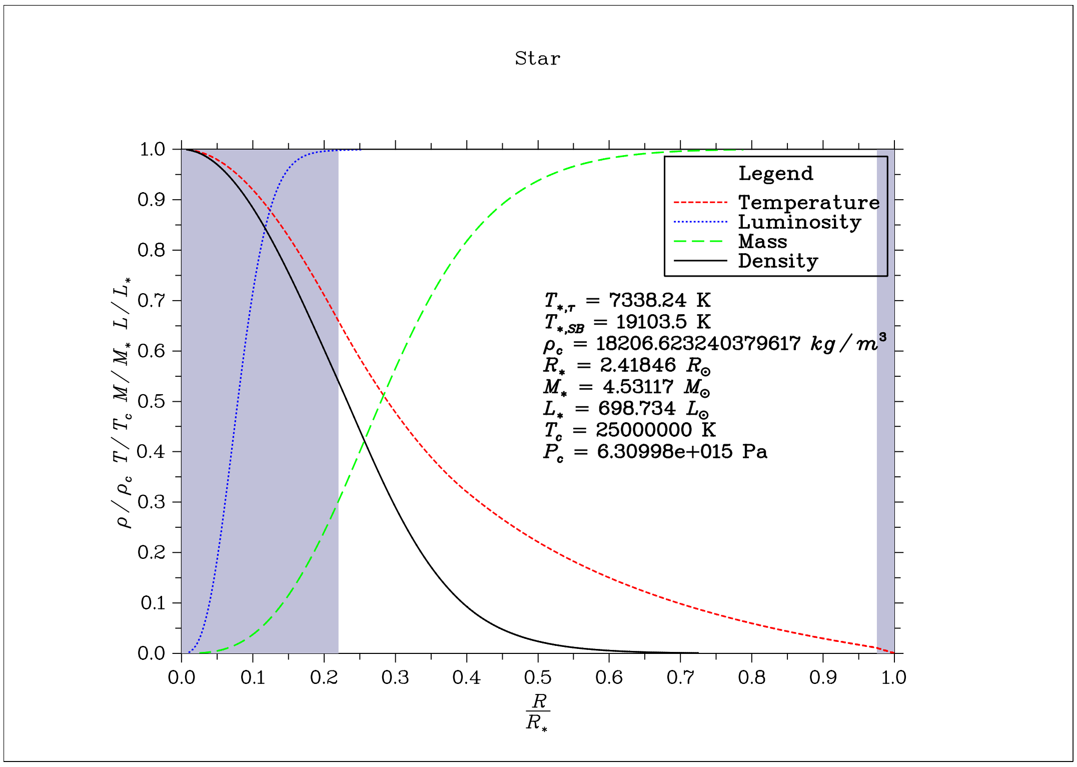

FIG. 15: Star with T_c of 2.5×10^7 K. ρ, T, M, L plotted as a function of $R/R_\odot$. Convection zones shown in grey.

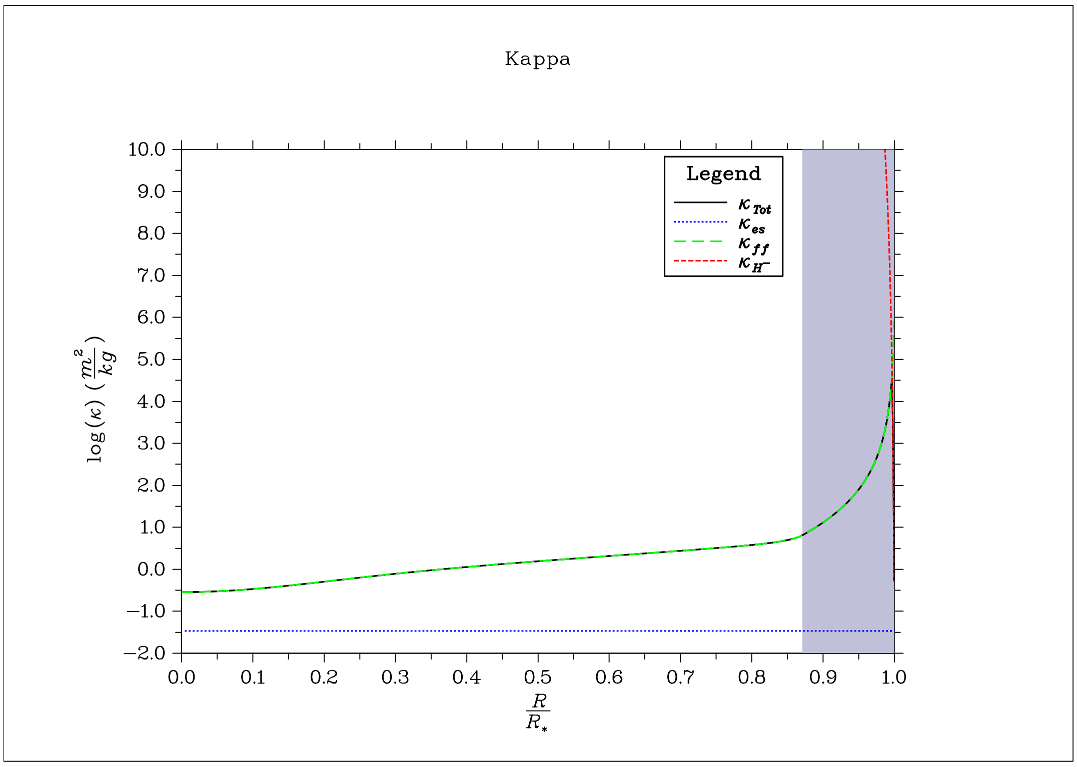

FIG. 16: Star with T_c of $9.0{\times}10^6$ K. κ plotted as a function of $R/R_\odot$. Convection zones shown in grey.

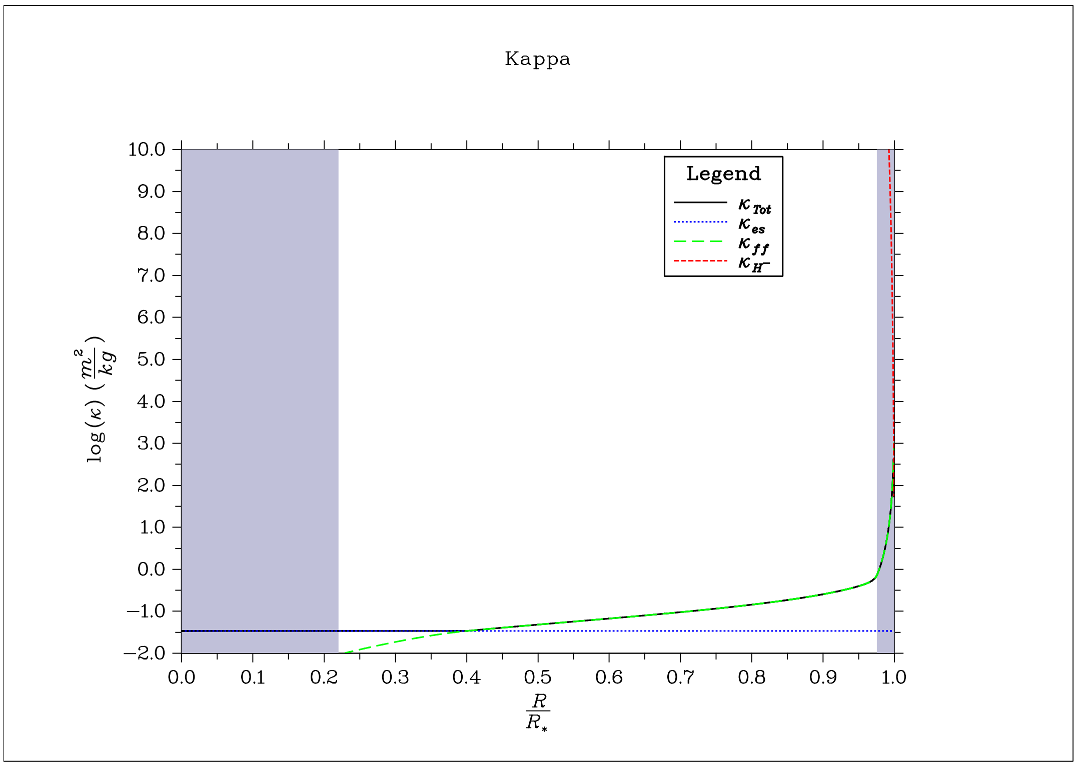

FIG. 17: Star with T_c of $2.5{\times}10^7$ K. κ plotted as a function of $R/R_\odot$. Convection zones shown in grey.

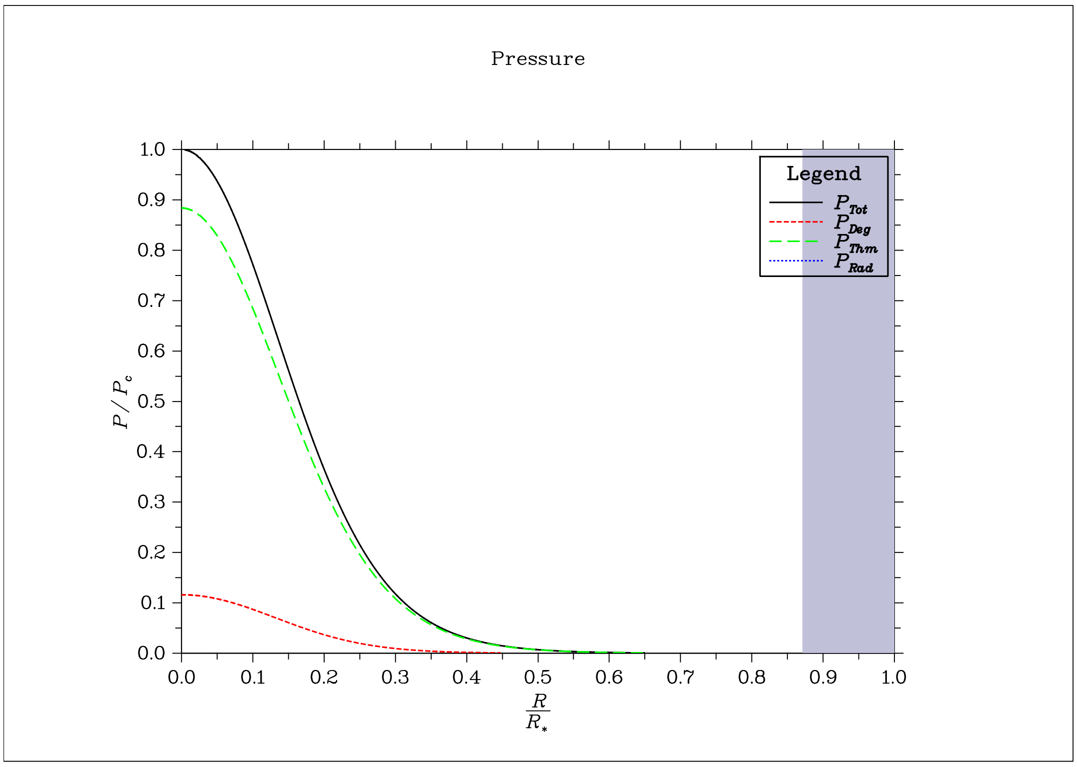

FIG. 18: Star with T_c of 9.0×10^6 K. Pressure of various sources plotted as a function of $R/R_\odot$. Convection zones shown in grey.

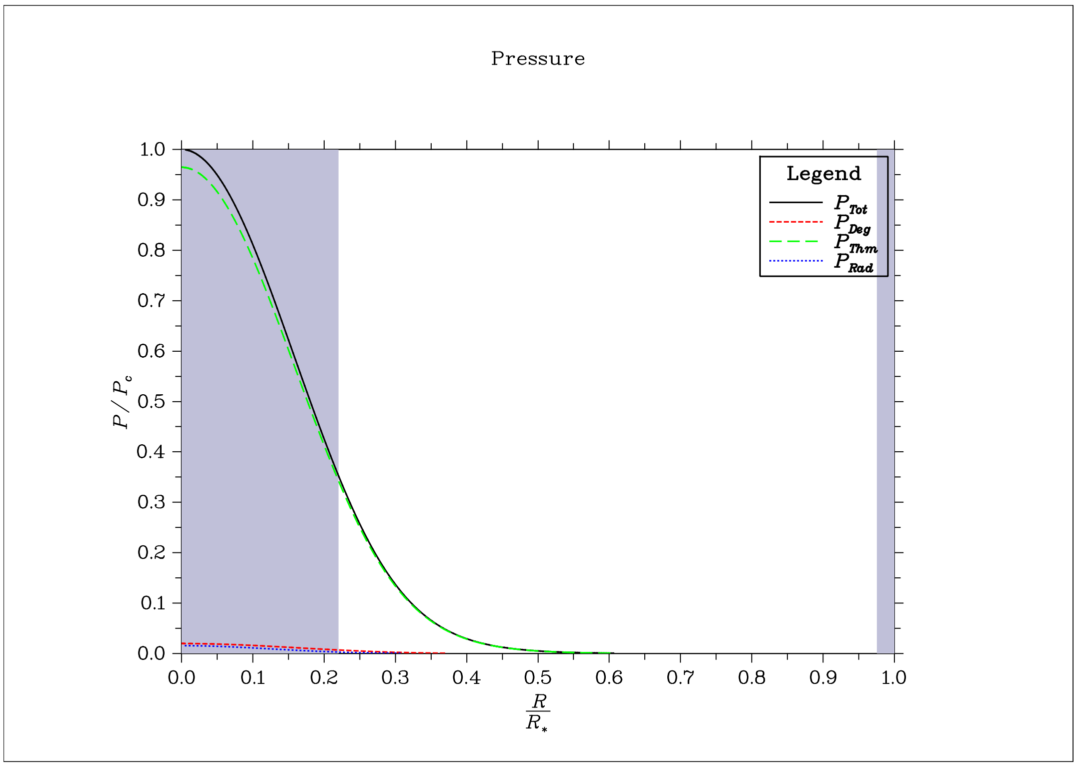

FIG. 19: Star with T_c of 2.5×10^7 K. Pressure of various sources plotted as a function of $R/R_\odot$. Convection zones shown in grey.

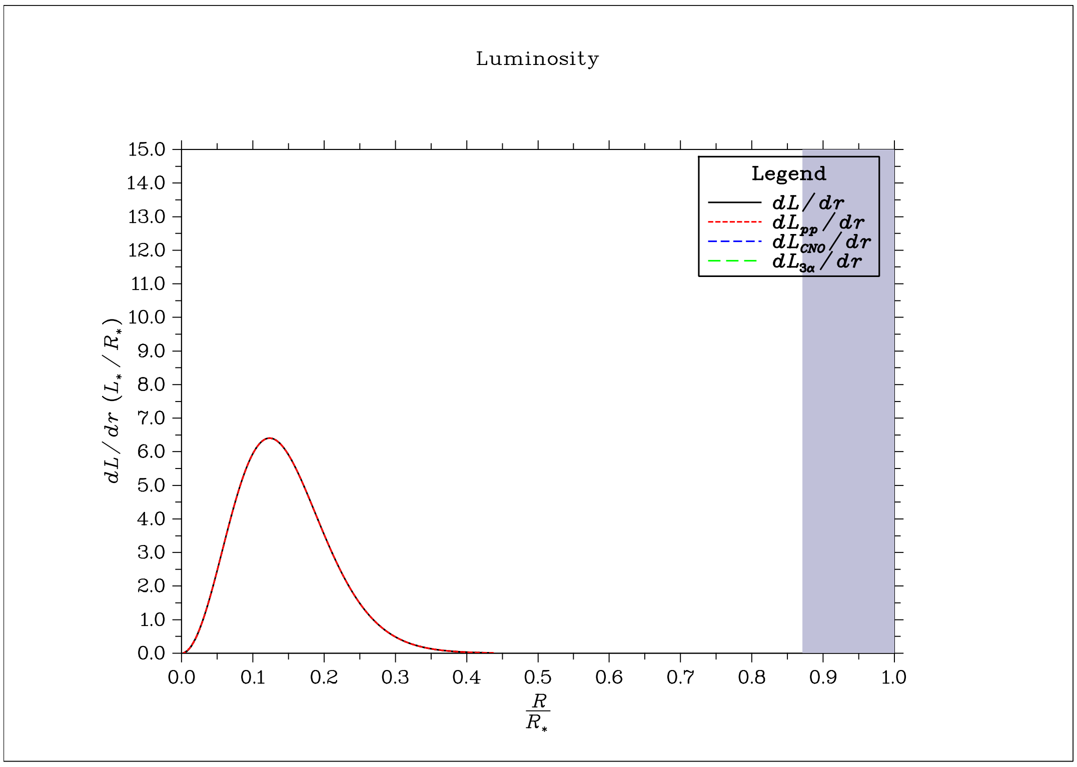

FIG. 20: Star with T_c of $9.0{\times}10^6$ K. dL/dr of various sources plotted as a function of $R/R_\odot$. Convection zones shown in grey.

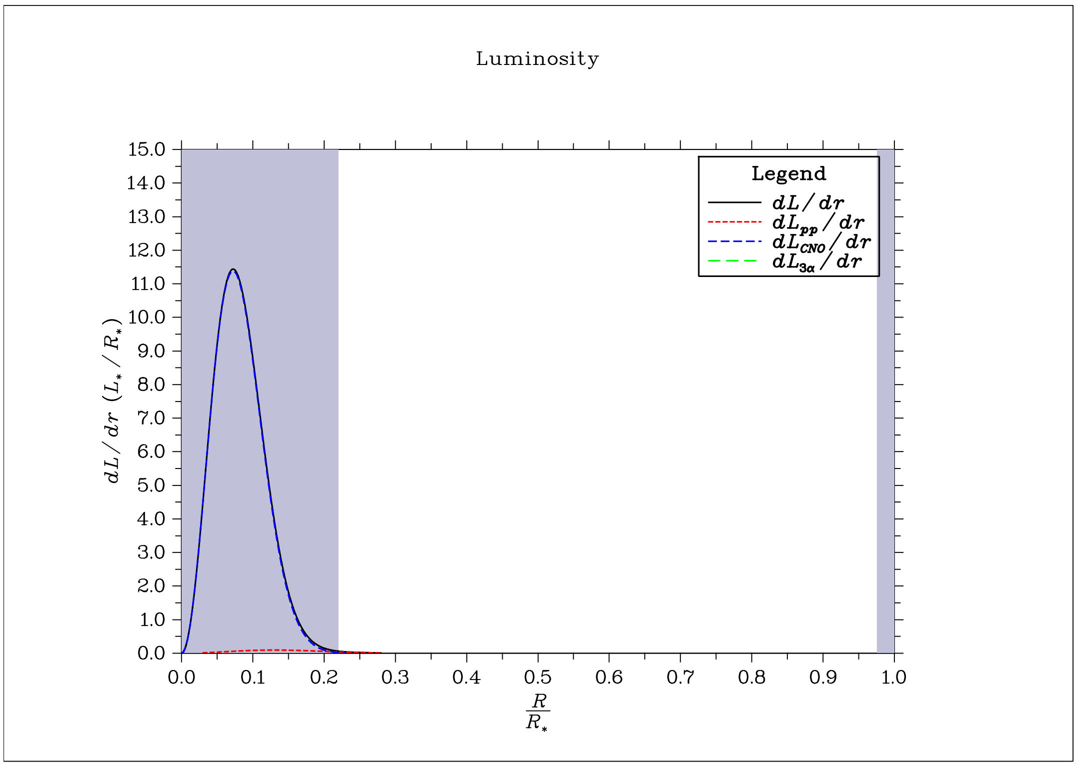

FIG. 21: Star with T_c of $2.5{\times}10^7$ K. dL/dr of various sources plotted as a function of $R/R_\odot$. Convection zones shown in grey.

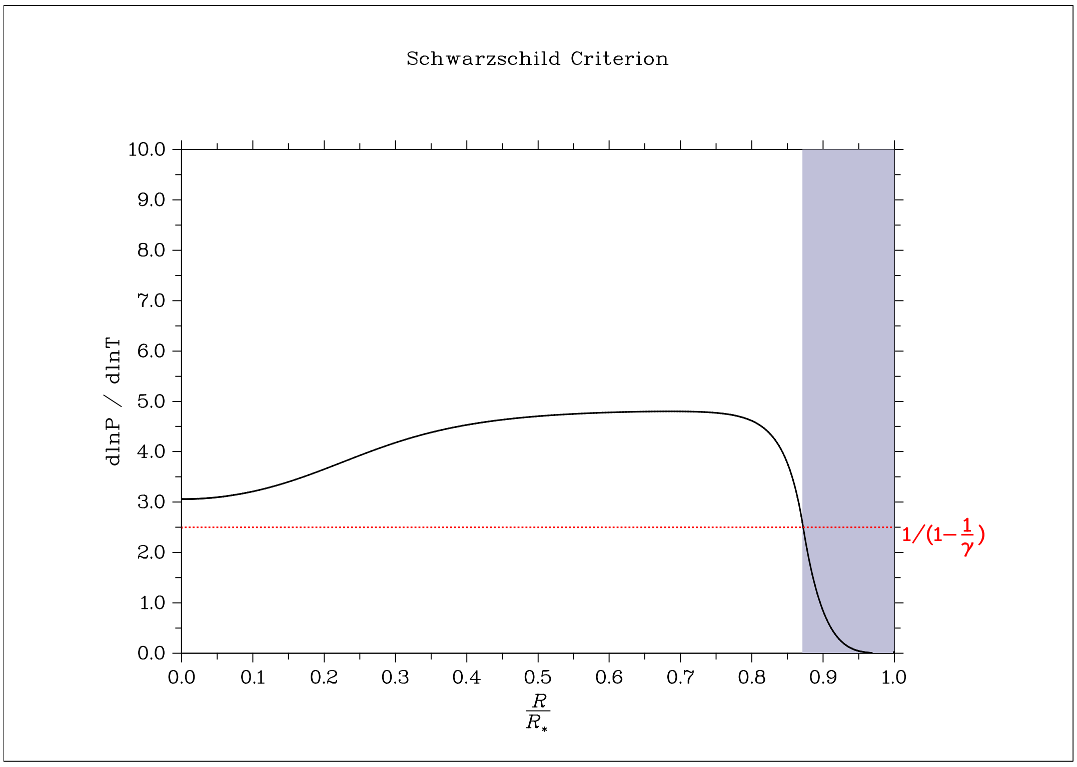

FIG. 22: Star with T_c of $9.0{\times}10^6$ K. $dlnP/dlnT$ plotted as a function of $R/R_\odot$. Convection zones shown in grey.

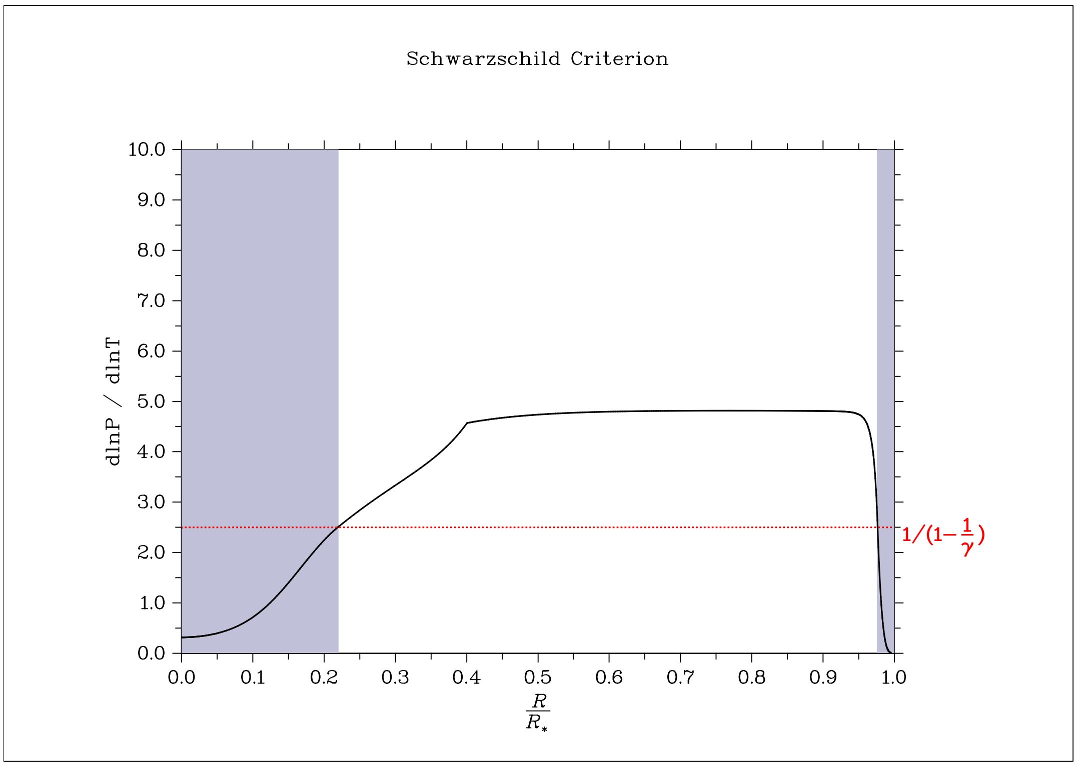

FIG. 23: Star with T_c of $2.5{\times}10^7$ K. $dlnP/dlnT$ plotted as a function of $R/R_\odot$. Convection zones shown in grey.

The convection zones in the above stars are labelled in blue/grey. In the low mass star (with T_c of $9.0{\times}10^6$ K) and higher mass star (with T_c of $2.5{\times}10^7$ K), there is convective region envelope at the surface of the star. The pressure gradient is low in this region and the temperature gradient is high in this region. The radiative temperature gradient is high due to a large increase in opacity near the surface of the star, because of the lower temperatures the H$^-$ contribution to opacity drives up, as seen in Fig.16 and Fig.17.

The larger mass star has a convective core where the smaller star does not. The larger star has a convective core because the core temperature, T_c is sufficiently high such that the CNO cycle begins to dominate in energy production as seen inFig.21, while the smaller star is dominated by the PP-chain since its T_c isn't high enough to start CNO burning. This CNO burning creates high temperatures in the core that falls off rapidly, creating a high temperature gradient that causes convection in the core.

As mentioned, the core temperature of the smaller star is not hot enough to start CNO burning (Fig.20), so it is dominated by the PP-chain, while the larger star has begun CNO burning (Fig.21). The larger star is dominated by CNO burning but the PP-chain is still active in the star, but to a much lesser extent. Both stars however, do not have sufficient core temperatures to because the 3α-process, this only becomes relevant at temperatures at $1{\times}10^8$ K.

In the core of the larger star, Thomson electron-scattering is the dominant source of opacity, this is again because of the higher core temperature where free-free absorption increases with temperature. At high temperatures the free-free absorption affects the κ_{ff} inversely with temperature, driving it down as seen in Fig.17. The temperature is not high enough in the core of the low mass star so κ_{ff} dominates over k_{es} for the majority of the structure of the star as seen in Fig.16. In the latter parts of both stars, k_{ff} is the dominant source of opacity until the temperature near their surfaces drop sufficiently for H$^-$ ions to contribute significantly; at this point near the surface the other sources of opacity drop off, κ_{H^-} dominates and falls of as the star converges at their surface.

IV. REFERENCES

[1] Broderick, Avery E. PHY 375 Final Project. University of Waterloo, 2015.
[2] `http://en.wikipedia.org/wiki/Runge-Kutta-Fehlberg_method`
[3] `http://www.marekfiser.com/Projects/Real-time-visualization-of-3D-vector-field-with-CUDA/4-Vector-field-integrators-for-stream-line-visualization`
[4] Ryden, Barbara Sue., and B. M. Peterson. Foundations of Astrophysics. San Francisco: Addison-Wesley, 2010. 330-31. Print.