UNIVERSITY OF WATERLOO
Faculty of Science

PHYS 375 Final Project

PHYS 375
Waterloo, Ontario

Prepared by

Robert Burnet
3B Physics and Astronomy
ID 20465122
April 5, 2016

Table of Contents

I Algorithmic Choices

A Adaptive Runge-Kutta-Fehlberg and Bisection Method

To solve the stellar structure equations, our group decided to adapt a 4th-5th order adaptive step-sized Runge-Kutta-Fehlberg (RKF45) method (see rkf.py in the **Appendix**). RKF45 is a numerical method to solve differential equations of order O(h^4), using the O(h^5) order to estimate error for implementing adaptive step sizes. It is identified by its Butcher Tableau which details the coefficients used for the integration method (see table 1 below).

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	
	16/135	0	6656/12825	28561/56430	-9/50	2/55
	25/216	0	1408/2565	2197/4104	-1/5	0

Table 1: Butcher tableau of RKF45

The numbers to the right of the vertical line and above the horizontal line correspond to elements of the Runge-Kutta matrix, the numbers to the left of the vertical line correspond to the nodes, and the numbers below the horizontal line correspond to the weights. The first row below the horizontal line gives the O(h^5) order accurate method where as the second row gives the O(h^4) order accurate method [1]. We use the 5th order to find the error with our step size and adaptively change its size to accommodate for the error. This saves time for solving the differential equations over just carrying out an RK4 method to solve it by allowing us to adaptively change the step sizes as we won't needlessly be using small steps when not much is changing; we can make the step sizes bigger when the solutions to the differential equations don't change as much, and decrease the step sizes only when we need to using this method.

Along with using RKF45 to solve the differential equations, we also employed a type of adaptive bisection method to find the root to the equation:

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

This bisection method is similar to the one suggested in the project description [2], except that it adaptively changes its precision to save time (see adaptive_bisection.py in the **Appendix**). At first, all one needs to start the bisection method is a value greater than the desired value, and a value less than the desired value. The precision of the chosen values is not important. The precision only becomes important as you approach the desired value. This method adaptively increases the precision as you approach the desired value, while starting off at a very low precision, all to save time.

We start with a ρ_c that gives a value of $f(\rho_c)$ greater than 0, and a ρ_c that gives a value of $f(\rho_c)$ less than zero, to a low precision. We then employ bisection method, increase the precision, and carry it out again until we reach a ρ_c that gives $f(\rho_c) = 0$ to an acceptable precision.

B Creating and Plotting Stars

stellar_generator.py details all the differential equations and parameters we needed to solve to create a star. In it, there is a Star class which contains all the parameters, functions, and differential equations needed to solve for the stellar structure equations. It is in here we employ the RKF45 method and adaptive bisection method to solve for the stellar structure. One can call the desired parameters of a solved star by calling its specific attribute from its Star class. For instance, if you wish to attain the central density of the star, you simply call [solved star's variable name].density_c after solving the star using [solved star's variable name].solve .

stellar_plotter.py calls stellar_generator.py with an input star's free parameters (central temperature and X, Y, and Z composition values) to solve the stellar structure equations for that star, and then plots them into the various plots we desire, such as the normalized ρ, T, M, L, P, dL/dr, and κ as functions of r/R_* plots, the partial pressures, luminosities, opacities, and the dlogP/dlogT plots. stellar_plotter.py can also call main_sequence.py to generate the various main sequence plots.

main_sequence.py details a MainSequence class which solves the stellar structure equations of various stars, given the range of central temperature, the compositions, and the number of stars desired to generate by calling stellar_generator.py. The stars can be plotted on an HR diagram, a $L/L_\odot$ as a function of $M/M_\odot$ plot, and a $R/R_\odot$ as a function of $M/M_\odot$ plot through stellar_plotter.py.

There were other scripts as well, such as constants.py which simply held all the physical constants needed, composition.py which made sure Z = 1 + X + Y, progress.py which simply printed a progress bar to screen for the bisection method, timing_profiler.py which timed the execution of stellar_plotter.py, dot_dict.py which created more class attributes, and where_positive.py which returns intervals where a given value is positive which is used for stellar_plotter.py to plot the convective regions.

II Main Sequence

The following figure 1 is the HR diagram of 100 main sequence stars with central temperatures ranging from 5×10^6 K to 3.5×10^7 K.

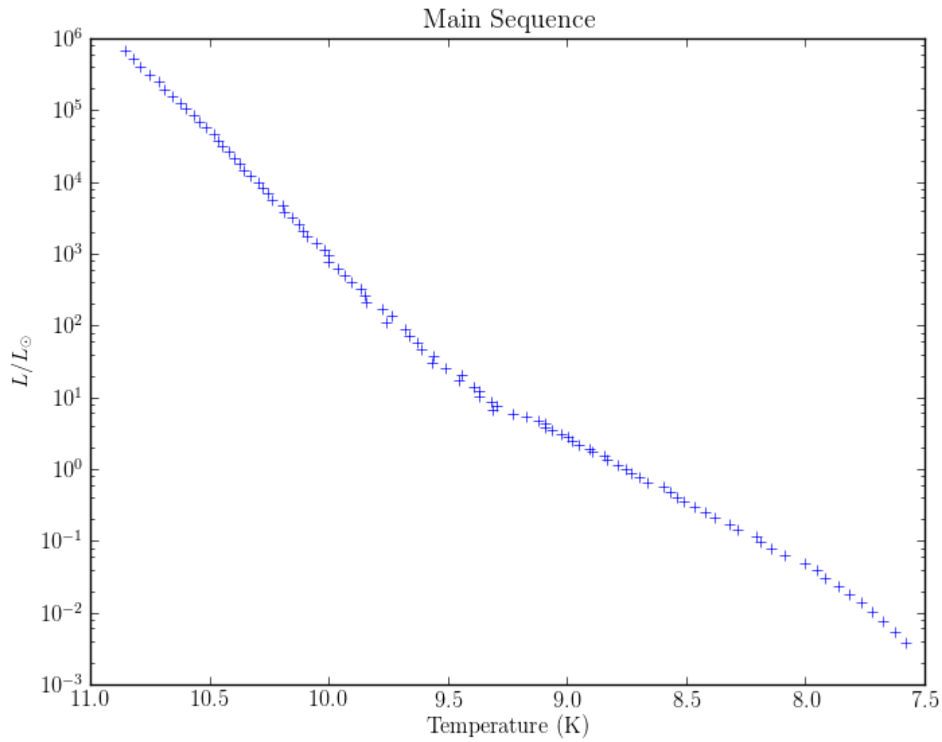


Figure 1: HR diagram of 100 main sequence stars with central temperatures ranging from 5×10^6 K to 3.5×10^7 K

As expected, an increase in surface temperature increases luminosity. This plot agrees with the theory.

The following two figures, figures 2 and 3, are the $L/L_\odot$ vs $M/M_\odot$ and $R/R_\odot$ vs $M/M_\odot$ plots of 100 main sequence stars with central temperatures ranging from 5×10^6 K to 3.5×10^7 K respectively. Also shown in each plot are the empirical expressions found in the text, p. 330 [3].

The calculated plots agree with the empirical expressions well. There is an interesting feature in the $R/R_\odot$ vs $M/M_\odot$ plot where the stars' radius seem to remain constant as you increase mass in a region near where the empirical expression changes, but other than this feature, the plots seem to agree.

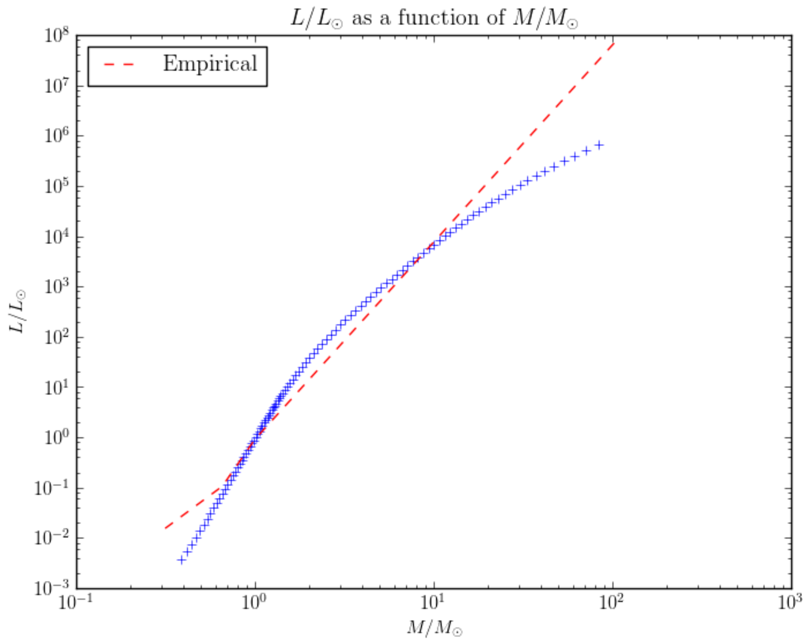


Figure 2: luminosity vs mass plot of 100 main sequence stars with central temperatures ranging from $5{\times}10^6$ K to $3.5{\times}10^7$ K

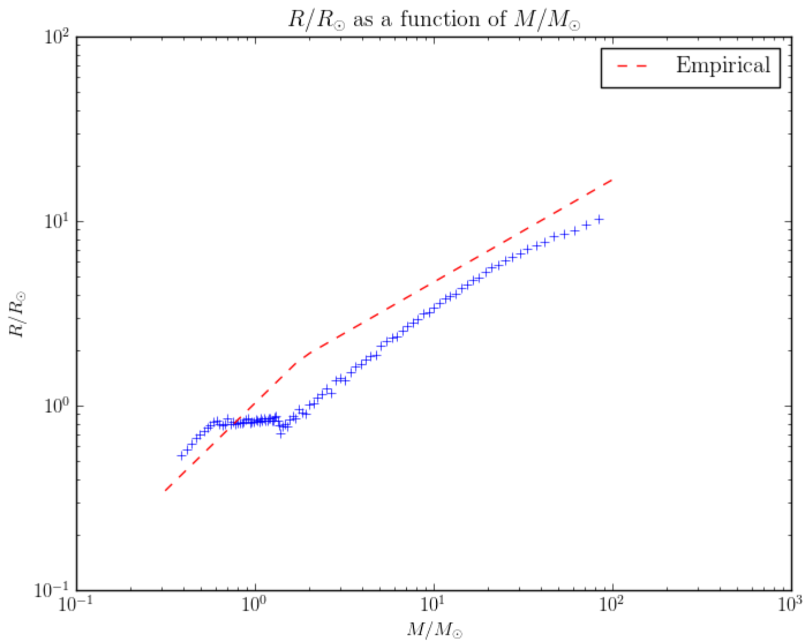


Figure 3: radius vs mass plot of 100 main sequence stars with central temperatures ranging from $5{\times}10^6$ K to $3.5{\times}10^7$ K

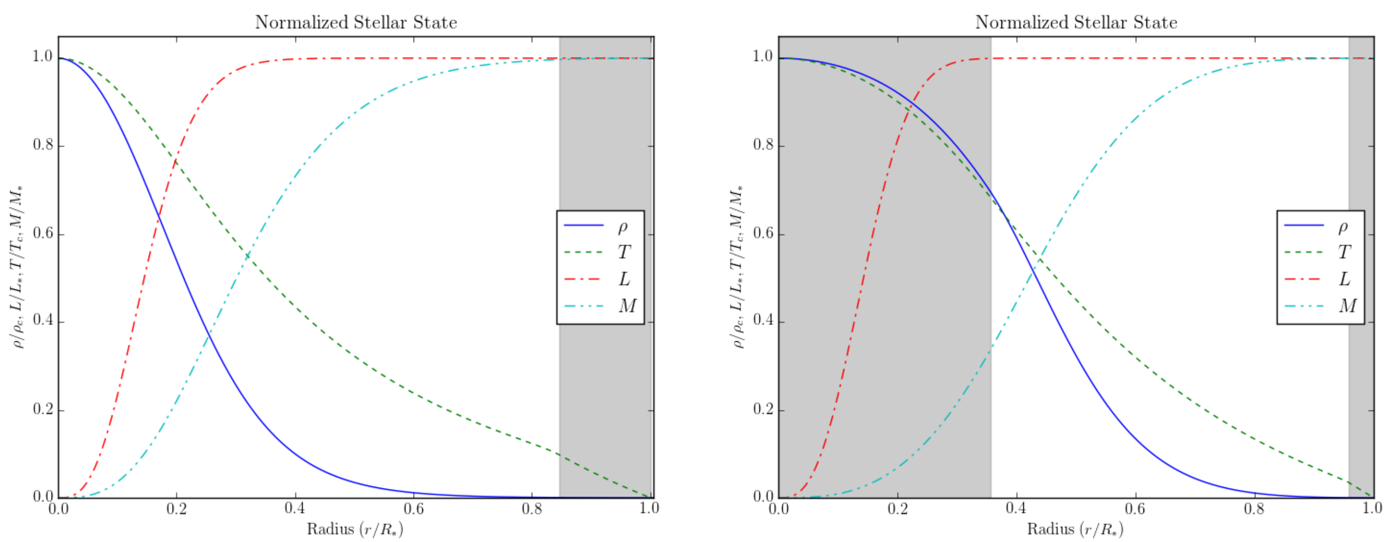
III Stellar Structure

Two stars, one of mass $0.709M_{\odot}$ and another of mass $35M_{\odot}$, were used to compare stellar structure. The following table 2 details each stars' surface temperature, central density, radius, mass, and luminosity.

	Star 1	Star 2
Central Temperature	9×10^{6} K	3.5×10^{7} K
Surface Temperature	3795 K	41885 K
Central Density	74238 kg/m^3	1237 kg/m^3
Radius	$0.81R_{\odot}$	$7.08R_{\odot}$
Mass	$0.709M_{\odot}$	$35M_{\odot}$
Luminosity	$0.122L_{\odot}$	$1.4\times10^{5}L_{\odot}$

Table 2: Butcher tableau of RKF45

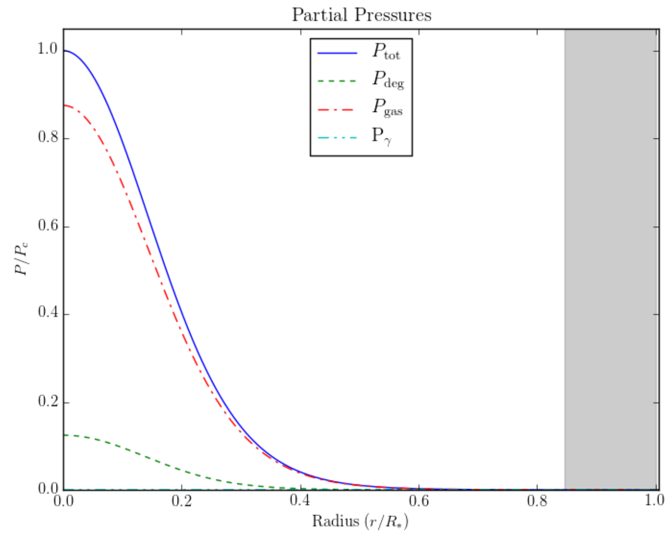
Their features such as ρ, T, M, L, P, dL/dr, and κ were plotted as functions of r/R_*, as well as their partial pressures, luminosities, opacities, and the dlogP/dlogT, as shown in the below figures.



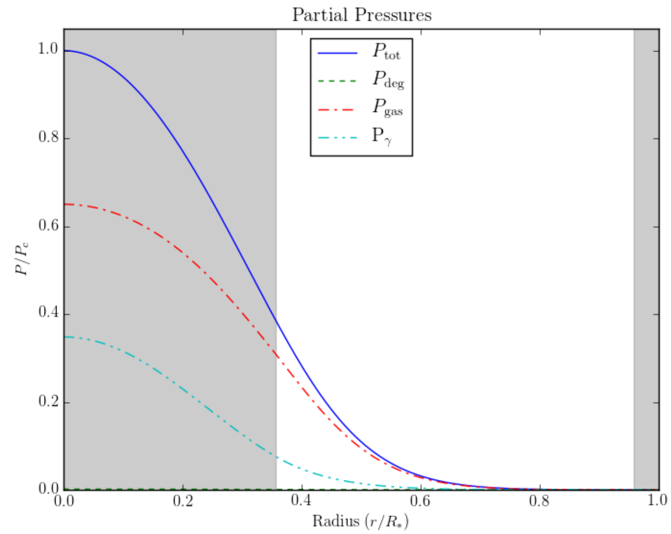
(a) Stellar structure plots of star with mass $0.709M_{\odot}$

(b) Stellar structure plots of star with mass $35M_{\odot}$

Figure 4: Stellar structure plots of both stars

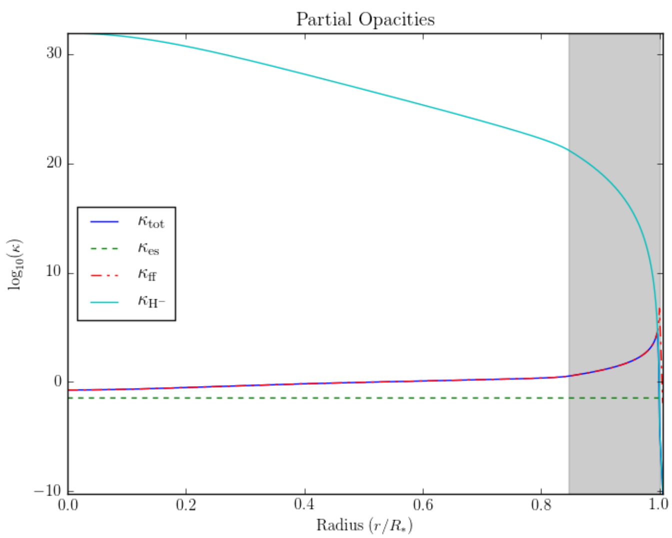


(a) Partial pressure plot of star with mass $0.709M_\odot$

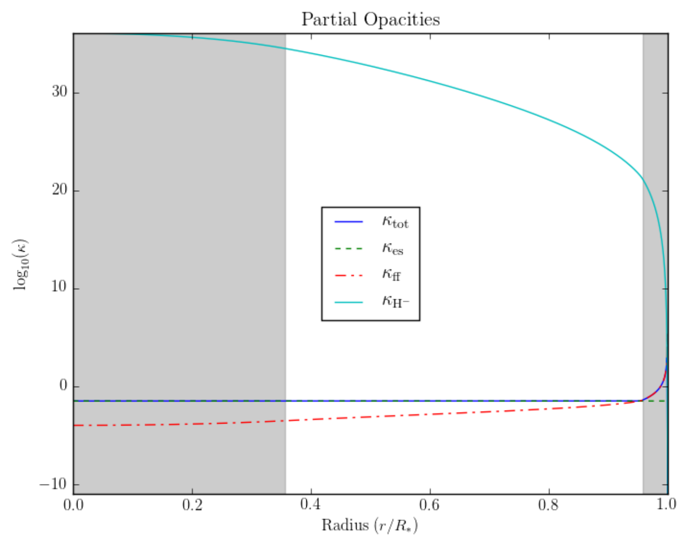


(b) Partial pressure plot of star with mass $35M_\odot$

Figure 5: Partial pressure plots of both stars

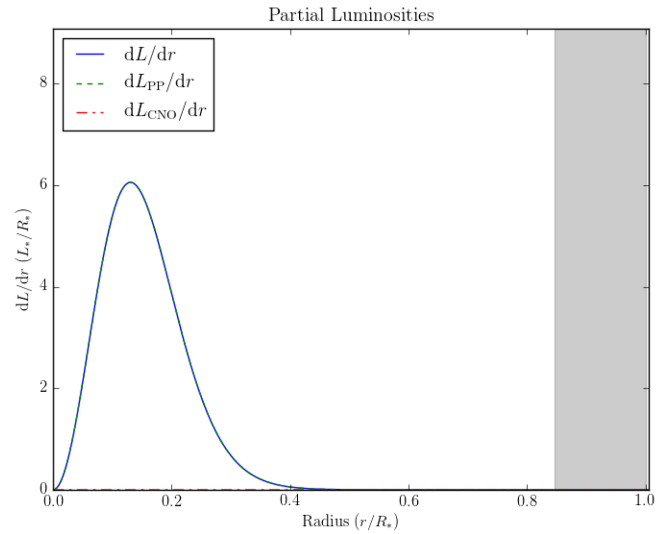


(a) Partial opacity plot of star with mass $0.709M_\odot$

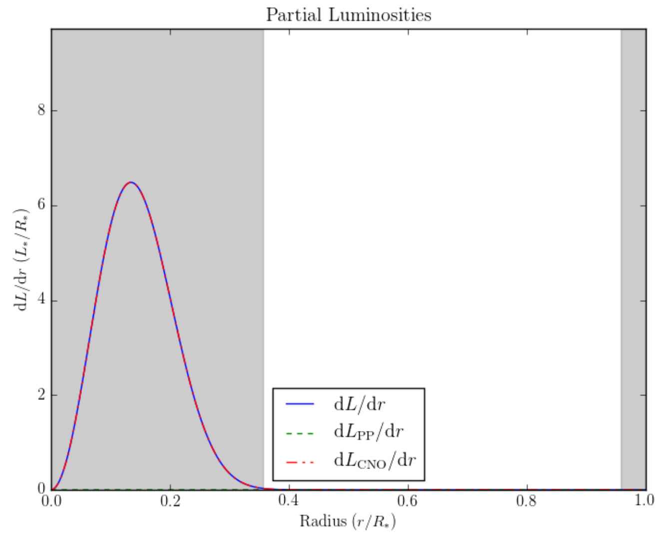


(b) Partial opacity plot of star with mass $35M_\odot$

Figure 6: Partial opacity plots of both stars

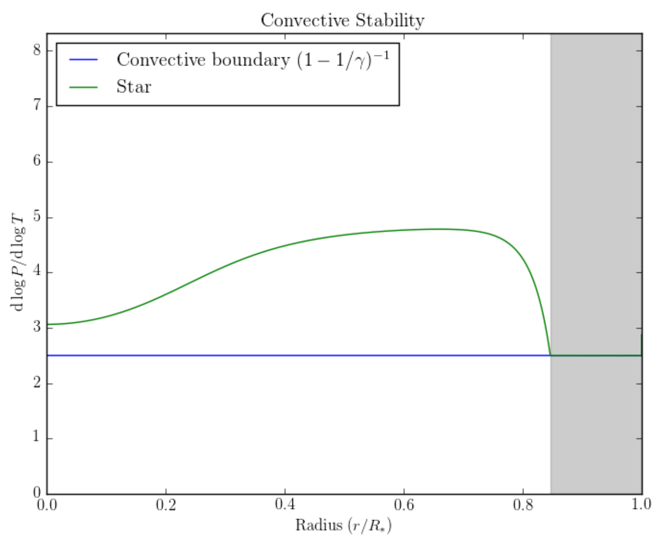


(a) Partial luminosity plot of star with mass $0.709 M_\odot$

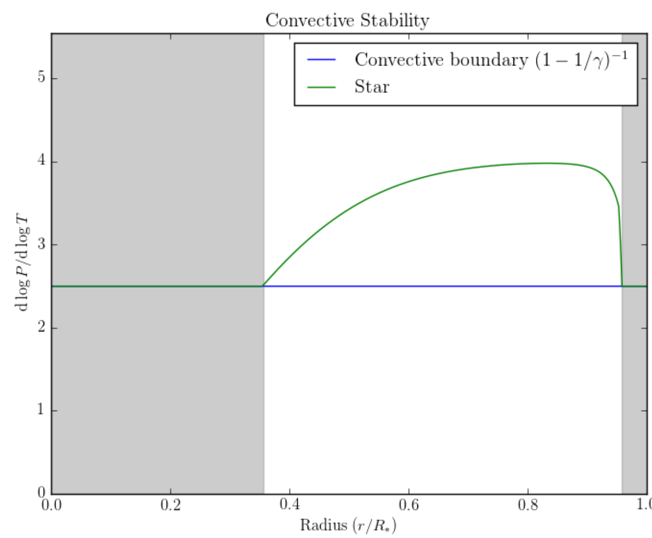


(b) Partial luminosity plot of star with mass $35 M_\odot$

Figure 7: Partial luminosity plots of both stars



(a) Convective stability plot of star with mass $0.709 M_\odot$



(b) Convective stability plot of star with mass $35 M_\odot$

Figure 8: Convective stability plots of both stars

An interesting feature to note is that the convective zone in the 0.709 solar mass star is entirely at the surface of the star, taking about 15% of the total radius of the star, while the rest of the star is radiative. However, the 34 solar mass star has a large convective zone at the center of the star that takes up about 35% of the radius, and another convective zone at the surface taking about 5% of the radius, with a radiative zone in between. This large convective zone at the center appears due to the fact that photon gas pressure is contributory to the total pressure, and it scales with T^4, forcing both terms in the temperature gradient to scale with T^3, allowing the convective term to become the flatter of the two and drive convection. The photon gas pressure in the 0.709 mass star is not as contributory until about the last 15% of the star's radius, which is why it doesn't have a convective zone until then.

Looking at figure 7, in the 0.709 solar mass star, the PP chain dominates over the CNO cycle. However, for the 34 solar mass star, the CNO cycle dominates over the PP chain. This is because the CNO cycle only kicks in at approximately 1.7×10^7 K, and since the central temperature of the lower mass star is lower than that at 9×10^6

K, its temperature is too low for the CNO cycle to kick in. The 34 solar mass star has a temperature of 3.5×10^7 K, which is higher than the requirement for the CNO cycle to kick in, and maintains a high enough temperature throughout its central convective region for the CNO cycle to dominate. The CNO cycle dominates over the PP chain for the large mass star because the CNO cycle energy generation rate scales with $T^19.9$, whereas the PP chain energy generation rate scales with only T^4, so a higher temperature means the CNO cycle energy generation rate increases faster than the PP chain, resulting in its domination.

Looking at figure 6, in the 0.709 solar mass star, the free-free scattering opacity dominates over the electron scattering opacity until it reaches the surface of the star where the H^- opacity takes over and drives the opacity down. However, for the 34 solar mass star, the electron scattering opacity dominates over the free-free scattering opacity throughout most of the star until it reaches the surface convective zone where the free-free scattering opacity takes over, followed by the H^- opacity near the surface of the star. The larger mass star is mostly dominated by electron scattering opacity because the equation for total opacity is:

$$\kappa(\rho, T) = \left[\frac{1}{\kappa_{H^-}} + \frac{1}{\max(\kappa_{es}, \kappa_{ff})}\right]^{-1} \tag{2}$$

with each components being:

$$\kappa_{es} = 0.02(1 + X)\mathrm{m}^2/\mathrm{kg} \tag{3}$$

$$\kappa_{ff} = 1.0 \times 10^{24}(Z + 0.0001)\rho_3^{0.7}T^{-3.5}\mathrm{m}^2/\mathrm{kg} \tag{4}$$

$$\kappa_{H^-} = 2.5 \times 10^{-32}(Z/0.02)\rho_3^{0.5}T^9\mathrm{m}^2/\mathrm{kg} \tag{5}$$

[2]Note that free-free scattering opacity scales with $T^-3.5$ (the higher the temperature, the smaller it's opacity) whereas electron scattering opacity only depends on X (the mass fraction of hydrogen), and so remains flat throughout the star no matter the temperature or density. Also note that the total opacity is driven by the maximum of either electron scattering or free-free scattering, and is the sum of the inverse of either of those two and the H^- opacity. For the large mass star, the temperature is high enough for the free-free scattering opacity to fall below the electron scattering opacity, allowing the electron scattering opacity to dominate throughout most of the star, until the temperature decreases low enough for the free-free scattering opacity to rise above the electron scattering opacity and take over. The smaller mass star doesn't have a high enough temperature for the free-free scattering opacity to fall below the electron scattering opacity, and so it remains dominated by the free-free scattering opacity throughout most of the star. H^- opacity scales with T^9, and so remains high throughout most of both of the stars, contributing not much to the total opacity, until the temperature drops enough for it to drop below the free-free scattering opacity near the surface of the star and take over all the way to the surface.

IV Appendix

stellar_generatory.py:

```python
from __future__ import division, print_function

from constants import *
from composition import Composition
from rkf import rkf
from adaptive_bisection import adaptive_bisection

# --- Stellar State Enum ---
ss_size = 4
# ---------------------------
density = 0
temp = 1
lumin = 2
mass = 3
# opt_depth = 4
readable_strings = ("Radius", "Density", "Temperature", "Luminosity", "Mass") #, "Opt Depth")

class Star():

    def __init__(self, temp_c, composition):
        self.composition = composition
        self.temp_c = temp_c
        self.delta_tau_thres = 1e-20
        self.stellar_structure_eqns = [self.drho_dr, self.dT_dr, self.dL_dr, self.dM_dr] # Needs to match stellar state enum order above
        self.stellar_structure_size = ss_size
        self.is_solved = False

    def diP_diT(self, ss, r):
        """
        The partial pressure gradient with respect to temperature
        """
        ideal_gas = ss[density] * k / (self.composition.mu * m_p)
        photon_gas = 4/3 * a * ss[temp]**3
        return ideal_gas + photon_gas

    def diP_dirho(self, ss, r):
        """
        The partial pressure gradient with respect to pressure
        """
        ideal_gas = k * ss[temp] / (self.composition.mu * m_p)
        nonrel_degenerate = (5/3) * nonrelgenpress * ss[density]**(2/3)
        return ideal_gas + nonrel_degenerate

    def dP_dr(self, ss, r):
        return - (G * ss[mass] * ss[density])/(r**2)

    def partial_opacity(self, ss, r):
        # Electron Scattering Opacity
        kes = 0.02 * (1+self.composition.X)
        # Free-free Opacity
        kff = 1e24 * (self.composition.Z+0.0001) * ((ss[density] * 1e-3)**0.7) * (ss[temp]**(-3.5))
        # Hydrogen opacity
        kH = 2.5e-32 * ((self.composition.Z+tiny_float)/0.02) * ((ss[density] * 1e-3)**0.5) * (ss[temp]**9)
        # Avoid division by zero (kes, kff cant be zero) added tiny_float

        return (kes, kff, kH)

    def opacity(self, ss, r):
        """
        Opacity of the star as it depends on composition, density and temperature
        """
        (kes, kff, kH) = self.partial_opacity(ss,r)

        kappa = 1/((1/kH)+(1/max(kff,kes)))
        return kappa

    def partial_pressure(self, ss, r):
        """
        Each of the three pressure sources at different points in th star
        """
        nonrel_degenerate = nonrelgenpress * ss[density]**(5/3)
        ideal_gas = k * ss[density] * ss[temp] / (self.composition.mu * m_p)
        photon_gas = 1/3 * a * ss[temp]**4
        return (nonrel_degenerate, ideal_gas, photon_gas)

    def pressure(self, ss, r):
        """
        Pressure at a given state in the star due to three competing effects
        """
        return sum(self.partial_pressure(ss,r))

    def partial_energy_prod(self, ss, r):
        X = self.composition.X
        ep_pp = ep_pp_coeff * ss[density] * X**2 * ss[temp]**4
        ep_cno = ep_cno_coeff * ss[density] * X * (X * 0.03) * ss[temp]**19.9 # X_CNO = X * 0.03?
        return (ep_pp, ep_cno)

    def energy_prod(self, ss, r):
        """
        Total energy production for a given X, density and temperature
        """
        return sum(self.partial_energy_prod(ss,r))

    def delta_tau(self, ss, r):
```

```python
        """
        Delta tau stopping condition used to determine when r is well outside the star
        """
        return (self.opacity(ss, r) * ss[density]**2) / abs(self.drho_dr(ss, r))

    def partial_dT_dr(self, ss, r):
        """
        The two methods of energy transport
        """
        radiative = (3 * self.opacity(ss, r) * ss[density] * ss[lumin]) / ( 16 * pi * a * c * ss[temp]**3 * r**2)
        convective = ( 1 - 1 / gamma) * ( ss[temp] / self.pressure(ss, r) ) * ( G * ss[mass] * ss[density] ) / ( r**2 )

        return (radiative, convective)

    # Stellar Structure
    def dT_dr(self, ss, r):
        """
        Temperature gradient with respect to radius
        """
        return -min(*self.partial_dT_dr(ss, r))

    # Stellar Structure
    def drho_dr(self, ss, r):
        """
        Density gradient with respect to radius
        """
        return - ((G * ss[mass] * ss[density])/(r**2) + self.diP_diT(ss,r) * self.dT_dr(ss, r))/(self.diP_dirho(ss,r))

    # Stellar Structure
    def dM_dr(self, ss, r):
        """
        Mass gradient with respect to radius
        """
        return 4 * pi * r**2 * ss[density]

    def partial_dL_dr(self, ss, r):
        """
        Luminosity gradient with respect to radius
        """
        partial_energy_production = self.partial_energy_prod(ss,r)
        mass_grad = self.dM_dr(ss,r)
        dL_dr_pp = mass_grad * partial_energy_production[0]
        dL_dr_cno = mass_grad * partial_energy_production[1]
        return (dL_dr_pp, dL_dr_cno)

    # Stellar Structure
    def dL_dr(self, ss, r):
      """
      Luminosity gradient with respect to radius
      """
      return sum(self.partial_dL_dr(ss, r))

    # Stellar Structure
    def dtau_dr(self, ss, r):
        """
        Optical depth gradient
        """
        return - self.opacity(ss, r) * ss[density]

    def solve(self):
        if self.is_solved:
            return

        # (i_surf, ss, r, delta_tau_surf) = self.solve_density_c(1.5e3)
        # raise Exception("fds")
        # density_c = adaptive_bisection(self.solve_density_c_error, 1e-1, 1e14, 0.01)
        # density_c = adaptive_bisection(self.solve_density_c_error, 1e4, 1e9, 0.01)
        density_c, tol = adaptive_bisection(self.solve_density_c_error, 1, 1e10)
        # density_c = adaptive_bisection(self.solve_density_c_error, 0.03, 500, 1)

        # print("---- Solving Star With Correct Central Density ---")
        (i_surf, ss, r, delta_tau_surf) = self.solve_density_c(density_c, tol)
        # print("--------------------- Solved ---------------------")

        self.i_surf = i_surf
        self.ss_profile = ss
        self.r_profile = r
        self.ss_surf = ss[:,i_surf]
        self.r_surf = r[i_surf]
        self.density_c = density_c
        self.delta_tau_surf = delta_tau_surf
        self.lumin_surf_bb, self.lumin_surf_rkf = self.relative_surface_lumin(i_surf, ss, r)
        self.temp_surf = (self.lumin_surf_rkf / (4 * pi * sigma * self.r_surf**2))**(1/4)
        self.lumin_surf = self.ss_surf[lumin]
        self.mass_surf = self.ss_surf[mass]
        self.density_surf = self.ss_surf[density]
        self.data_size = len(r)

        self.is_solved = True

    def relative_surface_lumin(self, i, ss, r):
        lumin_surf_bb = 4 * pi * sigma * r[i]**2 * ss[temp, i]**4
        lumin_surf_rkf = ss[lumin, i]

        return lumin_surf_bb, lumin_surf_rkf

    def solve_density_c_error(self, density_c, tol):
        lumin_surf_bb, lumin_surf_rkf = self.relative_surface_lumin(*self.solve_density_c(density_c, tol)[0:3])

        error = (lumin_surf_rkf - lumin_surf_bb)/np.sqrt(lumin_surf_rkf * lumin_surf_bb)
        return error
```

```python
    def system_DE(self, ss, r):
        if np.min(ss) < 0:
            raise ValueError("Stellar state values are negative.")
        # elif np.isinf(self.opacity(ss, r)) or np.isnan(self.opacity(ss, r)):
        #     raise ValueError("Opacity too big.")
        else:
            return np.array([f(ss,r) for f in self.stellar_structure_eqns])

    def solve_density_c(self, density_c, tol):
        if self.is_solved:
            return

        temp_c = self.temp_c

        r_0 = 1 # Doesn't matter really as long as this is less than scale height

        ic = np.empty(self.stellar_structure_size)

        ic[density] = density_c
        ic[temp] = temp_c
        ic[mass] = 4 * pi / 3 * r_0**3 * density_c
        ic[lumin] = ic[mass] * self.energy_prod(ic, r_0)

        # print(self.delta_tau_thres)
        tau_surf = 2/3

        r, ss = rkf(self.system_DE, r_0, ic, tol, self.stop_condition)

        surfaced = False

        for i in reversed(xrange(1, r.shape[0]-1)):
            i_delta_tau_inner = self.delta_tau(ss[:, i-1], r[i-1])
            i_delta_tau_outer = self.delta_tau(ss[:, i], r[i])
            if (i_delta_tau_inner > tau_surf):
                if abs(i_delta_tau_inner - tau_surf) > abs(i_delta_tau_outer - tau_surf):
                    i_surf = i # Take outer
                else:
                    i_surf = i-1 # Take inner

                i_surf = i_surf
                ss_surf = ss[:, i_surf]
                r_surf = r[i_surf]
                delta_tau_surf = self.delta_tau(ss[:, i_surf], r[i_surf])
                surfaced = True
                break

        assert surfaced, "Surface wasn't reached"

        return (i_surf, ss, r, delta_tau_surf)

    def stop_condition(self, i, ss, r):
        delta_tau = self.delta_tau(ss, r)
        mass_limit = 1e3 * M_s
        # if ss[mass] > mass_limit:
            # print(delta_tau)
            # print("Terminating star due to mass limit.")
            # return True
        if delta_tau < self.delta_tau_thres:
            # print(i,ss,r)
            # print(delta_tau)
            # print("Terminating star due to optical depth.")
            return True
        return False

    def log_solved_properties(self):
        print("------- Solved Variables -------")
        log_format = "{0:40} {1:10.10E}"
        print(log_format.format("Delta tau surface (2/3 = 6.7E-1)", self.delta_tau_surf))
        print(log_format.format("Central density", self.density_c))
        print(log_format.format("Luminosity Blackbody", self.lumin_surf_bb))
        print(log_format.format("Luminosity RKF", self.lumin_surf_rkf))
        print(log_format.format("Inferred Surface Temperature", self.temp_surf))
        print(log_format.format("Data Size", self.data_size))

        self.log_ss()
        self.log_ss(self.ss_surf, self.r_surf)

    def log_ss(self, ss=None, r=None):
        if ss is not None:
            log_format = "{0:20.10E} | {1:20.10E} | {2:20.10E} | {3:20.10E} | {4:20.10E}"
            print(log_format.format(r, *ss))
        else:
            print("{0:20} | {1:20} | {2:20} | {3:20} | {4:20}".format(*readable_strings))

    def log_raw(self, a=0, b=0):
        size = self.ss_profile.shape[1]
        if a>0:
            print("Printing first {0} data entires.".format(a))
            self.log_ss()
            for i in np.arange(0, min(size, a), 1):
                self.log_ss(self.ss_profile[:, i], self.r_profile[i])
        if b>0:
            print("Printing last {0} data entires.".format(b))
            self.log_ss()
            for i in np.arange(max(min(size, size-b), 0), size, 1):
                self.log_ss(self.ss_profile[:, i], self.r_profile[i])
```

StellarModellingMetallicity/code/stellar_generator.py stellar_plotter.py:

```
from __future__ import division, print_function

from matplotlib import rc
import os
import matplotlib.pyplot as plt
from stellar_generator import *
from constants import gamma,mach_ep
from composition import Composition
from main_sequence import MainSequence
from where_positive import where_positive
import math
import os

# Computer modern fonts
rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)

def plot_star(star):
    if not star.is_solved:
        star.solve()

    star_dir_name = "../figures/star_comp-{comp}_Tc-{temp_c}".format(comp = star.composition.file_string,temp_c=star.temp_c)
    # Previous file name
    #star_file_name = "../figures/{prefix}_star_comp-{comp}_Tc-{temp_c}.pdf".format(prefix = "{prefix}", comp = star.composition.file_string, temp_c = star.temp_c)
    star_file_name = (star_dir_name+"/{prefix}.png").format(prefix="{prefix}")
    if not os.path.exists(os.path.dirname(star_file_name)):
        try:
            os.makedirs(os.path.dirname(star_file_name))
        except OSError as exc: # Guard against race condition
            if exc.errno != errno.EEXIST:
                raise
    i_surf = star.i_surf
    i_count = len(star.r_profile)
    lumin_surf = star.lumin_surf
    temp_surf = star.temp_surf
    r_surf = star.r_surf
    mass_surf = star.mass_surf
    lumin_p = star.ss_profile[lumin, :]
    mass_p = star.ss_profile[mass, :]
    density_p = star.ss_profile[density, :]
    density_c = star.density_c
    temp_p = star.ss_profile[temp, :]
    central = star.ss_profile[:, 0]
    r = star.r_profile
    r_0 = star.r_profile[0]
    pressure_c = star.pressure(central, r_0)
    pressure_p = np.zeros([4, i_count])
    opacity_p = np.zeros([4, i_count])
    dL_dr_p = np.zeros([3, i_count])
    is_convective = np.zeros(i_count)
    radiative = np.zeros(i_count)
    convective = np.zeros(i_count)
    pressure_grad = np.zeros(i_count)
    for i in xrange(i_count):
        ss_i = star.ss_profile[:, i]
        r_i = star.r_profile[i]
        partial_pressure = star.partial_pressure(ss_i, r_i)
        pressure_grad[i] = star.dP_dr(ss_i, r_i)
        pressure_p[0, i] = sum(partial_pressure)
        pressure_p[1:4, i] = [p for p in partial_pressure]

        partial_opacity = star.partial_opacity(ss_i, r_i)
        opacity_p[0, i] = star.opacity(ss_i, r_i)
        opacity_p[1:4, i] = [k for k in partial_opacity]

        partial_dL_dr = star.partial_dL_dr(ss_i, r_i)
        dL_dr_p[0, i] = star.dL_dr(ss_i, r_i)
        dL_dr_p[1:3, i] = [k for k in partial_dL_dr]

        partital_energy_trans = star.partial_dT_dr(ss_i, r_i)
        radiative[i], convective[i] = partital_energy_trans
        if (radiative[i] > convective[i]):
            is_convective[i] = 1

    convective_regions = where_positive(is_convective)

    n_r = r / r_surf
    n_lumin = lumin_p / lumin_surf
    n_mass = mass_p / mass_surf
    n_temp = temp_p / star.temp_c
    n_density = density_p / star.density_c
    n_pressure = pressure_p / pressure_c
    n_opacity = np.log10(opacity_p)
    n_dL_dr = dL_dr_p * r_surf / lumin_surf
    n_ss = [n_density, n_temp, n_lumin, n_mass]

    # dlogP_dlogT = - (temp_p / pressure_p[0]) * pressure_grad / np.minimum(radiative, convective)
    # dlogP_dlogT = - (temp_p / pressure_p[0]) * pressure_grad / np.minimum(radiative, convective)
    logP = np.log(pressure_p[0])[:i_surf]
    logT = np.log(temp_p)[:i_surf]
    dlogP = logP[1:] - logP[:-1]
    dlogT = logT[1:] - logT[:-1]

    dlogP_dlogT = dlogP/(dlogT+mach_ep)

    # Plot for stellar state values
    plt.figure()
    plt.title(r"Normalized Stellar State")
```

```python
    plt.xlabel(r"Radius ($r/R_*$)")
    plt.ylabel(r"$\rho/\rho_c, L/L_*, T/T_c, M/M_*$")
    n_ss_labels = [r"$\rho$", r"$T$", r"$L$", r"$M$"]
    plots = [plt.plot(n_r, n_ss_i)[0] for n_ss_i in n_ss]
    plots[1].set_dashes([4,4])
    plots[2].set_dashes([8,4,2,4])
    plots[3].set_dashes([8,4,2,4,2,4])
    #l4.set_dashes([8,4,2,4,2,4,2,4])
    #l5.set_dashes([8,4,2,4,2,4,2,4,2,4])
    plt.axis([0,n_r[-1],0, 1.05])
    plt.legend(plots, n_ss_labels, loc="best", numpoints = 1)
    plt.gca().set_autoscale_on(False)
    for region in convective_regions:
        plt.axvspan(n_r[region[0]], n_r[region[1]], color='gray', alpha=0.4)
    plt.savefig(star_file_name.format(prefix="stellar_state"), format="png")
    # plt.show()

    # Plotting pressure decomposition
    plt.figure()
    plt.title(r"Partial Pressures")
    plt.xlabel(r"Radius ($r/R_*$)")
    plt.ylabel(r"$P/P_c$")
    n_pp_labels = [r"$P_{\mathrm{tot}}$", r"$P_{\mathrm{deg}}$", r"$P_{\mathrm{gas}}$", r"P_{\gamma}"]
    plots = [plt.plot(n_r, n_pressure_i)[0] for n_pressure_i in n_pressure]
    plots[1].set_dashes([4,4])
    plots[2].set_dashes([8,4,2,4])
    plots[3].set_dashes([8,4,2,4,2,4])
    plt.axis([0,n_r[-1],0, 1.05])
    plt.legend(plots, n_pp_labels, loc="best", numpoints = 1)
    plt.gca().set_autoscale_on(False)
    for region in convective_regions:
        plt.axvspan(n_r[region[0]], n_r[region[1]], color='gray', alpha=0.4)
    plt.savefig(star_file_name.format(prefix="partial_pressure"), format="png")
    # plt.show()

    # Plotting luminosity decomposition
    plt.figure()
    plt.title(r"Partial Luminosities")
    plt.xlabel(r"Radius ($r/R_*$)")
    plt.ylabel(r"$\mathrm{d}L/\mathrm{d}r$ ($L_*/R_*$)")
    n_pl_labels = [r"$\mathrm{d}L/\mathrm{d}r$", r"$\mathrm{d}L_{\mathrm{PP}}/\mathrm{d}r$", r"$\mathrm{d}L_{\mathrm{CNO}}/\mathrm{d}r$"]
    plots = [plt.plot(n_r, n_dL_dr_i)[0] for n_dL_dr_i in n_dL_dr]
    plots[1].set_dashes([4,4])
    plots[2].set_dashes([8,4,2,4])
    plt.axis([0,n_r[-1],0,max(n_dL_dr[0])*1.5])
    plt.legend(plots, n_pl_labels, loc="best")
    plt.gca().set_autoscale_on(False)
    for region in convective_regions:
        plt.axvspan(n_r[region[0]], n_r[region[1]], color='gray', alpha=0.4)
    plt.savefig(star_file_name.format(prefix="partial_lumin"), format="png")
    # plt.show()

    # Plotting opacity decomposition
    plt.figure()
    plt.title(r"Partial Opacities")
    plt.xlabel(r"Radius ($r/R_*$)")
    plt.ylabel(r"$\log_{10}(\kappa)$")
    n_pk_labels = [r"$\kappa_{\mathrm{tot}}$", r"$\kappa_{\mathrm{es}}$", r"$\kappa_{\mathrm{ff}}$", r"$\kappa_{\mathrm{H}^-}$"]
    plots = [plt.plot(n_r, n_opacity_i)[0] for n_opacity_i in n_opacity]
    plots[1].set_dashes([4,4])
    plots[2].set_dashes([8,4,2,4])
    plt.axis([0,n_r[-1], np.min(n_opacity), np.max(n_opacity)])
    plt.legend(plots, n_pk_labels, loc="best")
    plt.gca().set_autoscale_on(False)
    for region in convective_regions:
        plt.axvspan(n_r[region[0]], n_r[region[1]], color='gray', alpha=0.4)
    plt.savefig(star_file_name.format(prefix="partial_opacity"), format="png")
    # plt.show()

    # Plotting logPlogT
    plt.figure()
    plt.title(r"Convective Stability")
    plt.xlabel(r"Radius ($r/R_*$)")
    plt.ylabel(r"$\mathrm{d}\log P/\mathrm{d}\log T$")
    n_lPlT_labels = [r"Convective boundary $(1 - 1/\gamma)^{-1}$", r"Star"]
    plots = []
    log_points = len(dlogP_dlogT)
    boundary = np.zeros(log_points)
    boundary.fill(1/(1-1/gamma))
    plots.append(plt.plot(n_r[:log_points], boundary)[0])
    plots.append(plt.plot(n_r[:log_points], dlogP_dlogT)[0])
    plt.axis([0,n_r[log_points], 0, np.max(dlogP_dlogT) * 1.3])
    plt.legend(plots, n_lPlT_labels, loc="best")
    plt.gca().set_autoscale_on(False)
    for region in convective_regions:
        plt.axvspan(n_r[region[0]], n_r[region[1]], color='gray', alpha=0.4)
    plt.savefig(star_file_name.format(prefix="dlogP_dlogT"), format="png")
    # plt.show()
    ## Saving the star specifics
    ## We are targetting :
    ## 1. Surface Temp
    ## 2. Central density
    ## 3. Central Temperature
    ## 4. Radius
    ## 5. Mass
    ## 6. Luminosity
    f = open(star_dir_name+'/profile.txt', 'w')
    f.write('Surface Temperature = '+ repr(temp_surf) + '\n')
    f.write('Central Density = '+repr(density_c)+'\n')
    f.write('Radius = '+repr(r_surf)+'\n')
```

```
    f.write('Mass = '+repr(mass_surf/M_s)+'\n')
    f.write('Luminosity = '+ repr(lumin_surf) +'\n')
    f.close()


def plot_step_sizes(star):
    if not star.is_solved:
        star.solve()

    steps = star.r_profile[1:] - star.r_profile[:-1]
    plt.figure()
    plt.title("Step Size Required")
    plt.gca().set_yscale("log")
    plt.plot(range(len(steps)), steps)
    plt.show()

def lumin_mass_exact(m):
    if m < 0.7:
        return 0.35*m**2.62
    else:
        return 1.02*m**3.92
vlme = np.vectorize(lumin_mass_exact)

def radius_mass_exact(m):
    if m < 1.66:
        return 1.06*m**0.945
    else:
        return 1.33*m**0.555
vrme = np.vectorize(radius_mass_exact)

def plot_main_sequence(v_main_seq):
    for main_seq in v_main_seq:
        if not main_seq.solved:
            main_seq.solve()


    labels = [r"$Z = {Z}$".format(Z=main_seq.composition.Z) for main_seq in v_main_seq]
    num_stars = sum([main_seq.num_stars for main_seq in v_main_seq])
    num_seqs = len(v_main_seq)

    main_seq_folder = "../figures/main_sequence_{0}_stars_{1}_seq/".format(num_stars, num_seqs)
    if not os.path.exists(main_seq_folder):
        os.makedirs(main_seq_folder)

    padding = 0.2
    # Main Sequence
    plt.figure()
    plt.title(r"Main Sequence")
    plt.xlabel(r"Temperature (K)")
    plt.ylabel(r"$L/L_{\odot}$")
    #eps = 1e-20
    #blah = log()
    blah = [math.log(x) for x in main_seq.temp_surf]
    plots = [plt.plot(blah, main_seq.n_lumin_surf, "+")[0] for main_seq in v_main_seq]
    plt.legend(plots, labels, loc="best")
    plt.gca().invert_xaxis()
    plt.gca().set_yscale("log")
    #plt.gca().set_xscale("log")
    plt.savefig(main_seq_folder + "ms.pdf", format="pdf")
    plt.show()

    merge_mass_surf = [main_seq.n_mass_surf for main_seq in v_main_seq]
    mass_space = np.linspace(np.min(merge_mass_surf)*(1 - padding), np.max(merge_mass_surf)*(1 + padding), 300)

    # L/L_sun as a function of M/M_sun
    plt.figure()
    plt.title("$L/L_{\odot}$ as a function of $M/M_{\odot}$")
    plt.xlabel(r"$M/M_{\odot}$")
    plt.ylabel(r"$L/L_{\odot}$")
    plots = [plt.plot(main_seq.n_mass_surf,main_seq.n_lumin_surf,"+")[0] for main_seq in v_main_seq]
    plots.append(plt.plot(mass_space,vlme(mass_space),"r--")[0])
    plt.legend(plots, labels + ["Empirical"], loc="best")
    plt.gca().set_yscale("log")
    plt.gca().set_xscale("log")
    plt.savefig(main_seq_folder + "LvM.pdf", format="pdf")
    plt.show()

    ## R/R_sun as a function of M/M_sun
    plt.figure()
    plt.title("$R/R_{\odot}$ as a function of $M/M_{\odot}$")
    plt.xlabel(r"$M/M_{\odot}$")
    plt.ylabel(r"$R/R_{\odot}$")
    plots = [plt.plot(main_seq.n_mass_surf,main_seq.n_r_surf,"+")[0] for main_seq in v_main_seq]
    plots.append(plt.plot(mass_space,vrme(mass_space),"r--")[0])
    plt.legend(plots, labels + ["Empirical"], loc="best")
    plt.gca().set_yscale("log")
    plt.gca().set_xscale("log")
    plt.savefig(main_seq_folder + "RvM.pdf", format="pdf")
    plt.show()

if __name__ == "__main__":
    # Remember to turn off logging in adaptive_bisection.py
#    composition = [Composition.fromZX(Z, 0.73) for Z in [0.00, 0.01, 0.015, 0.02, 0.03]]
#    v_main_seq = [MainSequence(min_core_temp=5e6, max_core_temp=3.5e7, composition=comp, num_stars=100) for comp in composition]
#    for main_seq in v_main_seq:
#        main_seq.solve_stars()

#    plot_main_sequence(v_main_seq)

    # test_star = Star(temp_c = 1.5e7, density_c=1.6e5, composition=Composition.fromXY(0.69, 0.29))
    # test_star = Star(temp_c = 3e7, composition=Composition.fromXY(0.73, 0.25))
```

```
    # test_star = Star(temp_c = 1.2e10, composition=Composition.fromXY(0.73, 0.25))
    # test_star = Star(temp_c = 1e6, composition=Composition.fromXY(0.73, 0.25))
    # test_star = Star(temp_c = 3.5e7, composition=Composition.fromXY(0.5, 0.1))
    # test_star = Star(temp_c = 1e8, composition=Composition.fromXY(0.73, 0.25))
    # test_star = Star(temp_c = 3.5e7, composition=Composition.fromXY(0.73, 0.25))
     test_star1 = Star(temp_c = 3.3e7, composition=Composition.fromXY(0.73, 0.25))
    # test_star2 = Star(temp_c = 9e6, composition=Composition.fromXY(0.73, 0.25))
    # test_star2 = Star(temp_c = 8.23e6, composition=Composition.fromXY(0.73, 0.27-0.00001))
    # test_star3 = Star(temp_c = 8.23e6, composition=Composition.fromXY(0.73, 0.17))

    # test_star.solve()
    # # # test_star.log_raw(b=20)
    # test_star.log_solved_properties()

   # # # plot_step_sizes(test_star)
    # plot_star(test_star1)
     plot_star(test_star1)
```

StellarModellingMetallicity/code/stellar_plotter.py

main_sequence.py:

```python
from __future__ import division, print_function
from multiprocessing import Process, Queue
import multiprocessing
from constants import *
from composition import Composition
import matplotlib.pyplot as plt
from matplotlib import rc
from stellar_generator import Star
from dot_dict import DotDict
from progress import printProgress
from timing_profiler import timing
# Computer modern fonts
rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)

N_CORES = multiprocessing.cpu_count()
PROCESSES_PER_CORE = 1
LOG = True

star_attributes_to_pickle = [ # Must be serializable attributes
    'i_surf',
    'ss_profile',
    'r_profile',
    'ss_surf',
    'r_surf',
    'density_c',
    'temp_c',
    'composition',
    'delta_tau_surf',
    'lumin_surf_bb',
    'lumin_surf_rkf',
    'temp_surf',
    'lumin_surf',
    'mass_surf',
    'density_surf',
    'data_size',
]

class MainSequence():

    def __init__(self, min_core_temp, max_core_temp,composition,num_stars):
        self.min_core_temp = min_core_temp
        self.max_core_temp = max_core_temp
        self.num_stars = num_stars
        self.composition = composition
        self.solved = False

    def star_worker(self, temp_c_vals, out_queue):
        for temp_c in temp_c_vals:
            star = Star(temp_c = temp_c, composition=self.composition)
            star.solve()
            star_pickle = {}
            for attribute in star_attributes_to_pickle:
                star_pickle[attribute] = getattr(star, attribute)
            out_queue.put(star_pickle)
            if LOG: printProgress(out_queue.qsize(), self.num_stars, "Workers")

    @timing
    def solve_stars(self):
        stars = []
        out_queue = Queue()
        num_processes = N_CORES * PROCESSES_PER_CORE
        stars_per_process = np.ceil(self.num_stars / num_processes)
        temp_c_space = np.linspace(start=self.min_core_temp,stop=self.max_core_temp,num=self.num_stars)
        # print("Solving {n} stars with temp_c_space:".format(n=self.num_stars))
        processes = []
        i = 0
        if LOG: printProgress(0, self.num_stars, "Workers")
        while i < self.num_stars:
            remaining_stars = self.num_stars - i
            batch = min(remaining_stars, stars_per_process)
            process = Process(target=self.star_worker, args=(temp_c_space[i:i + batch], out_queue))
            processes.append(process)
            process.start()
            i += batch

        for i in xrange(self.num_stars):
```

```
            star = DotDict(out_queue.get())
            stars.append(star)

        for process in processes:
            process.join()

        if LOG: printProgress(self.num_stars, self.num_stars, "Workers")

        self.stars = stars
        self.temp_surf = np.array([star.temp_surf for star in self.stars])
        self.lumin_surf = np.array([star.lumin_surf for star in self.stars])
        self.n_lumin_surf = self.lumin_surf / L_s
        self.mass_surf = np.array([star.mass_surf for star in self.stars])
        self.n_mass_surf = self.mass_surf / M_s
        self.r_surf = np.array([star.r_surf for star in self.stars])
        self.n_r_surf = self.r_surf / R_s
        self.solved = True
```

StellarModellingMetallicity/code/main_sequence.py

rkf.py:

```
from __future__ import division, print_function
import numpy as np
import matplotlib.pyplot as plt

# Coefficients used to compute the independent variable argument of f

a2  =   2.500000000000000e-01  #  1/4
a3  =   3.750000000000000e-01  #  3/8
a4  =   9.230769230769231e-01  #  12/13
a5  =   1.000000000000000e+00  #  1
a6  =   5.000000000000000e-01  #  1/2

# Coefficients used to compute the dependent variable argument of f

b21 =   2.500000000000000e-01  #  1/4
b31 =   9.375000000000000e-02  #  3/32
b32 =   2.812500000000000e-01  #  9/32
b41 =   8.793809740555303e-01  #  1932/2197
b42 =  -3.277196176604461e+00  # -7200/2197
b43 =   3.320892125625853e+00  #  7296/2197
b51 =   2.032407407407407e+00  #  439/216
b52 =  -8.000000000000000e+00  # -8
b53 =   7.173489278752436e+00  #  3680/513
b54 =  -2.058966861598441e-01  # -845/4104
b61 =  -2.962962962962963e-01  # -8/27
b62 =   2.000000000000000e+00  #  2
b63 =  -1.381676413255361e+00  # -3544/2565
b64 =   4.529727095516569e-01  #  1859/4104
b65 =  -2.750000000000000e-01  # -11/40

# Coefficients used to compute local truncation error estimate.  These
# come from subtracting a 4th order RK estimate from a 5th order RK
# estimate.

r1  =   2.777777777777778e-03  #  1/360
r3  =  -2.994152046783626e-02  # -128/4275
r4  =  -2.919989367357789e-02  # -2197/75240
r5  =   2.000000000000000e-02  #  1/50
r6  =   3.636363636363636e-02  #  2/55

# Coefficients used to compute 4th order RK estimate

c1  =   1.157407407407407e-01  #  25/216
c3  =   5.489278752436647e-01  #  1408/2565
c4  =   5.353313840155945e-01  #  2197/4104
c5  =  -2.000000000000000e-01  # -1/5

BUFFER = 2**14

def rkf( f, a, x0, tol, stop ):
    """Vectorized Runge-Kutta-Fehlberg method to solve x' = f(x,t) with x(t[0]) = x0.

    USAGE:
        T, X = rkf(f, a, b, x0, tol)

    INPUT:
        f     - an array that is the system of equations dvx/dt = vf(vx,t)
        a     - left-hand endpoint of interval (initial condition is here)
        x0    - initial x value: x0 = x(a)
        tol   - maximum value of local truncation error estimate
        stop  - stopping condition func(i, vx, t)

    OUTPUT:
        T     - NumPy array of independent variable values
        X     - NumPy array of corresponding solution function values

    NOTES:
        This function implements 4th-5th order Runge-Kutta-Fehlberg Method
        to solve the initial value problem

           dx
           -- = f(x,t),     x(a) = x0
           dt

        on the interval [a,...).

        Based on pseudocode presented in "Numerical Analysis", 6th Edition,
```

```python
        by Burden and Faires, Brooks-Cole, 1997.
    """

    s = len(x0)

    # Set t and x according to initial condition and assume that h begins with resonable value

    t = a
    x = x0
    # h = hmin
    h = 1

    # max_samples = np.ceil(abs((b-a)/hmin))

    # print("Sampling with initial buffer {0}".format(BUFFER))

    # Initialize arrays that will be returned

    T = np.empty( BUFFER )
    X = np.empty( [s, BUFFER] )

    T[0] = t
    # print(x.shape)
    # print(X.shape)
    X[:,0] = x

    i = 0

    while not stop(i,x,t):
        if i + 1 >= len(T):
            # print("Increasing buffer by {0}".format(BUFFER))
            T = np.hstack((T, np.empty(BUFFER)))
            X = np.hstack((X, np.empty( [s, BUFFER] )))

        # Compute values needed to compute truncation error estimate and
        # the 4th order RK estimate.

        try:
            k1 = h * f( x, t )
            k2 = h * f( x + b21 * k1, t + a2 * h )
            k3 = h * f( x + b31 * k1 + b32 * k2, t + a3 * h )
            k4 = h * f( x + b41 * k1 + b42 * k2 + b43 * k3, t + a4 * h )
            k5 = h * f( x + b51 * k1 + b52 * k2 + b53 * k3 + b54 * k4, t + a5 * h )
            k6 = h * f( x + b61 * k1 + b62 * k2 + b63 * k3 + b64 * k4 + b65 * k5, t + a6 * h )
        except ValueError:
            # The step size must be too large and it is interpolating to negative values
            h = h * 0.5
            continue

        fifth_order = x + r1 * k1 + r3 * k3 + r4 * k4 + r5 * k5 + r6 * k6
        fourth_order = x + c1 * k1 + c3 * k3 + c4 * k4 + c5 * k5

        error = np.max(abs((fifth_order - fourth_order)/(abs(fifth_order) + np.finfo(float).eps)))

        if error <= tol:
            t = t + h
            x = fourth_order
            T[i] = t
            X[:,i] = x
            i += 1

        # Now compute next step size, and make sure that it is not too big or
        # too small.
        # print(h)
        h = h * min( max( 0.84 * ( tol / (error + np.finfo(float).eps) )**0.25, 0.5 ), 2 )
        # h = h * min( 0.84 * ( tol / (r + np.finfo(float).eps) )**0.25, 4.0 )
        # h = h * 0.84 * ( tol / (error + np.finfo(float).eps) )**0.25

    # endwhile

    T = T[0:i]
    X = X[:,0:i]

    # print("Truncating buffer to {0} filled samples.".format(i-1))
    return ( T, X )


# Tests
def test_rkf():
    f0 = lambda x, t: x[1]
    f1 = lambda x, t: -x[0]
    f2 = lambda x, t: x[3]
    f3 = lambda x, t: -x[2]+x[1]/(x[4]**2+1)
    f4 = lambda x, t: x[1]

    stop = lambda i, x, t: t > 10*np.pi
    f = lambda x, t: np.array([f(x,t) for f in [f0, f1, f2, f3, f4]])

    T, X = rkf(f, 0, [1, 0, 0, 0, 0], 1, stop)

    plt.figure()
    for i in range(X.shape[0]):
        plt.plot(T, X[i,:])
    plt.show()

# test_rkf()
```

StellarModellingMetallicity/code/rkf.py

adaptive_bisection.py:

```python
from __future__ import division, print_function
import numpy as np
from progress import printProgress
import matplotlib.pyplot as plt
from matplotlib import rc
# Computer modern fonts
rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)

eval_tol_max = 0.5
eval_tol_min = 0.02

LOG = True

def tween(i, a, b):
    assert (0 <= i <= 1), "i needs to be normalized"

    return a + ((1-i) * i**(2) + (i) * i**(1/6)) * (b - a)

# i_space = np.linspace(0,1,1000)
# it = np.vectorize(tween)
# plt.figure()
# plt.title(r"Adaptive Precision $a=0.01, b=0.5$")
# plt.xlabel(r"Bisection Step")
# plt.ylabel(r"RKF Precision")
# plt.plot(i_space, it(i_space, 0.5, 0.01))
# plt.savefig("../figures/bisection_tween.png", format="png")
# plt.show()

def adaptive_bisection(f, a, b, precision=0.001):
    n_max = np.ceil(np.log2(abs(b-a) / precision))

    n = 1
    if LOG: printProgress(0, n_max, "Bisection")
    f_a = f(a, eval_tol_max)
    f_b = f(b, eval_tol_max)
    if f_a * f_b > 0:
        print(f_a, f_b)
        raise Exception("No root in range")
    best_c = None
    best_f = None
    best_tol = None

    cs = [a, b]
    fs = [f_a, f_b]
    while n <= n_max:
        c = (a + b)/2
        tol = tween(n/n_max, eval_tol_max, eval_tol_min)
        # print(tol)
        f_c = f(c, tol)
        # print(c,f_c)

        cs.append(c)
        fs.append(f_c)

        if best_c is None or (abs(f_c) < abs(best_f)):
            best_c = c
            best_f = f_c
            best_tol = tol

        if (f_c == 0 or (b-a)/2 < precision):
            if LOG: printProgress(n_max, n_max, "Complete")
            # print("Error", best_f, best_c)
            # plt.figure()
            # plt.plot(cs, fs, 'ro')
            # plt.axis([-0.1, 0.1, -100, 100])
            # plt.gca().set_autoscale_on(False)
            # plt.show()
            return (best_c, best_tol)
        if LOG: printProgress(n, n_max, "Bisection")
        n = n+1
        if (f_c * f_a > 0):
            a, f_a = c, f_c
        else:
            b, f_b = c, f_c
    raise Exception("n_max was not large enough.")
```

StellarModellingMetallicity/code/adaptive_bisection.py

References

[1] Hairer, E., Nørsett, S. P., & Wanner, G. (1987). *Solving ordinary differential equations.* Berlin: Springer-Verlag.

[2] *Physics 375 Final Project.* Waterloo (ON): University of Waterloo.

[3] Ryden, B. S., & Peterson, B. M. (2010). *Foundations of Astrophysics.* San Francisco: Addison-Wesley.