University of Oslo

AST3310

Project 2

Author
Erik Levén

April 27, 2017

Contents

Introduction

The goal of this project is to continue project 1 but to this time include energy transport by convection. Do be able to include convection we need to establish several new functions to calculate needed parameters as well as a test inside the integration loop where we check if the requirement for convection is fulfilled. The code will then be check against a "sanity check" and finally, set to produce a relative realistic star.

I Convection

Convection is the process of which heated plasma inside a star is being transported through a star due to a difference in temperature gradients. Relatively small stars, like our sun, has a convective layer in their outer regions and a radiative core and relatively large stars have convective layers close to their core but radiative outer regions. This in mainly due to the fact that low mass stars produce energy via the P-P chain, which at the core can be transported out by radiation. Large stars operate via the CNO cycle which has a much larger temperature dependency. Thus the energy has to be transported out via convection.

If regions inside a star would produce or otherwise obtain more energy than radiation can they can transport away these regions becomes hotter than their surroundings. A parcel of gas which is hotter than it's surroundings also has lower density which leads to a radial upwards buoyancy force acting on this parcel. If this same parcel does not exchange energy with the other matter inside the star during this radial upwards motion as well as keeping pressure equilibrium with said matter the parcel is said to be convectively unstable. This displacement is called an adiabatic displacement due to the fast that the parcel does not exchange any energy with it's surroundings. This criteria can be generalized by

$$\rho < \rho^\star$$

where $\rho^\star$ is the density outside of the parcel and ρ is the density of the parcel. The inequality can through equations $5.52 \rightarrow 5.60$ in the lecture notes[1] be shown to be equal to

$$\nabla^\star > \nabla_{ad} \tag{1}$$

where $\nabla = (\frac{\partial lnT}{\partial lnP})$, and the subscript ad stands for an adiabatic process. As long as this criteria is fulfilled we will have to calculate and add the convective energy transport throughout the star.

In project 1 we only had to take radiative energy transport into account. When correcting for convective energy transport we need to re-write the equation for the total flux as

$$F_{radiative} + F_{convective} = \frac{L}{4\pi r^2}$$

The convective flux can from first principles be written as

$$F_C = \rho c_P v \Delta T$$

since $\rho c_P \Delta T$ is the energy a parcel with density ρ and velocity v can move.

I.1 The temperature difference

To estimate the temperature difference for a parcel which is not rising adiabatically we write

$$\Delta T = \Big[\Big(\frac{\partial T}{\partial r}\Big)_p - \Big(\frac{\partial T}{\partial r}\Big)^\star\Big]\delta r = (\nabla^\star - \nabla_p)\frac{T\delta r}{H_P} \tag{2}$$

where H_P is the pressure scale height and δr ir the distance the parcel moves. For historical reason this distance is assumed to be half a mixing length, $\delta r = \frac{1}{2} l_m$ with $lm = H_P \alpha$ where

$$\begin{aligned}\alpha &= -\frac{P}{V}\Big(\frac{\partial V}{\partial P}\Big)_T \\ &= \frac{P}{V}\Big(\frac{-NkT}{P^2}\Big) = 1\end{aligned}$$

In equation 2 we have used the definition of the pressure scale height

$$H_P = -P\frac{\partial r}{\partial P}$$

to change the gradient of r to a gradient in pressure as described in equation 5.67 in the lecture notes. The value for the pressure scale height H_P to be used in this project can be found in the solution to exercise 5.9

Exercise 5.9

Assume that we are looking at a static atmosphere where gravity, and pressure are the only two forces present. Assume also that gravity can be represented by the gravitational acceleration and is constant and has the value g. Assume furthermore that gravity is aligned along r. Write down the force balance, and use it to find an equation for P as a function of r. What is the pressure scale height, for an ideal gas, when the pressure scale height is defined as the height over which the pressure drops by a factor $1/e$?

For a static atmosphere the force from gravity must be equal the force from the pressure difference:

$$-g\rho = \nabla P = \frac{\partial P}{\partial x} + \frac{\partial P}{\partial y} + \frac{\partial P}{\partial z}$$

One of our assumptions is that the sun looks the same in all directions except for the radial one, in this case z-direction. From the equation of state for an ideal we can also show that

$$\rho = \frac{P\mu m_\mu}{k_B T}$$

and we know from before that the gravitational force pr. mass is given by

$$g = G\frac{m}{r^2}.$$

We can now set up a separable differential equation on the form

$$\int_{P_0}^{P} \frac{dP}{P} = \int_0^z -\frac{g\mu m_\mu}{k_B T} dz'.$$

Solving this equation gives us

$$P = P_0 e^{-\frac{g\mu m_\mu}{k_B T} z} = P_0 e^{-zH_p}$$

where

$$H_p = \frac{k_B T}{g\mu m_\mu} = \frac{P}{g\rho} = \frac{Pr^2}{\rho G m} \tag{3}$$

We can now re-write the expression for the convective flux as

$$F_C = \rho c_P v \frac{T l_m}{2H_P}(\nabla^\star - \nabla_p) \tag{4}$$

and by introducing the temperature gradient of an adiabatic parcel we can re-write the expression for F_C as

$$F_C = \rho c_P v \frac{T l_m}{2H_P}(\nabla^\star - \nabla_{ad}) + \rho c_P v \frac{T l_m}{2H_P}(\nabla_{ad} - \nabla_p). \tag{5}$$

The first term is now the convective flux if the parcel rises adiabatically and the second term corrects for the loss of energy due to the radiation. Since the second term stands for the radiative loss of the parcel we can by assuming that the temperature gradient from outside and inside the parcel is twice the temperature difference divided by the diameter of the parcel use the definition of radiative flux:

$$f_R = -\frac{16}{3}\frac{\sigma T^3}{\kappa\rho}\frac{2\Delta T}{d} = -\frac{16}{3}\frac{\sigma T^4}{\kappa\rho H_P}\frac{l_m}{d}(\nabla^\star - \nabla_p) \tag{6}$$

To get the total energy lost by the parcel we need to multiply it with the surface area S of the parcel since flux is defined as energy pr. area pr. second. The second term in equation 5 is a difference in flux in a gas, and since this gas can only move upwards we have to multiply this term with the surface Q normal to the velocity v. Applying these correction, setting equation 6 equal to the second term in equation 5 and solving for $(\nabla_p - \nabla_{ad})$ we get

$$(\nabla_p - \nabla_{ad}) = \frac{32\sigma T^3}{3\kappa\rho^2 c_P v}\frac{S}{dQ}(\nabla^\star - \nabla_p) \tag{7}$$

The expression for the geometric factor $\frac{S}{dQ}$ is described in the solution of exercise 5.10.

Exercise 5.10

Calculate the geometric factor $S/(dQ)$ for a spherical parcel with radius r_p.

The radius of the parcel is simply half of the mean free path, or $\frac{l_m}{2}$. When assuming a spherical parcel we can calculate the surface area as $S = 4\pi r^2 = \pi l_m^2$. The area of the surface Q can be described as $Q = \pi r^2 = \frac{\pi l m^2}{4}$. Combining these values we get the following expression for the geometric factor:

$$\frac{S}{dQ} = \frac{4}{lm} \tag{8}$$

The only thing left now to have an expression for the convective flux is to find an expression for the velocity of the parcel.

I.2 The convective velocity

To find the convective velocity we need the kinetic energy of the parcel as it travels upwards. We know that the force pushing the parcel up is buoyancy, f_B. If we assume pressure equilibrium we can estimate the kinetic energy of the parcel by calculating the work done by the force on the parcel and subtract the work done by the parcel on the surrounding. As an estimation we will in this project assume that 50% of the total work done by the force on the parcel is used to by the parcel to push the gas away as it travels upwards. We can then define the kinetic energy pr. mass as

$$\begin{aligned}\frac{1}{2}v^2 &= \frac{1}{2}f_B\delta r \\ &= \frac{1}{2}g\delta\frac{\Delta T}{T}\delta r \\ &= \frac{g\delta}{2T}\frac{T}{H_P}(\nabla^\star - \nabla_p)(\delta r)^2 \\ &= \frac{g\delta}{2H_P}(\nabla^\star - \nabla_p)(\delta r)^2.\end{aligned} \tag{9}$$

Solving for the convective velocity gives us the expression

$$v = \sqrt{\frac{g\delta l_m^2}{4H_P}}(\nabla^\star - \nabla_p)^{1/2} \tag{10}$$

where variable δ is defined as

$$\begin{aligned}\delta &= \frac{T}{V}\Big(\frac{\partial V}{\partial T}\Big)_P \\ &= \frac{T}{V}\frac{Nk}{P} = 1\end{aligned}$$

when assuming an ideal gas.

I.3 Find the convective flux value

Combining the expression for the velocity and equation 7 with equation 4 we finally end up with an expression for the convective flux:

$$F_C = \rho c_P T \sqrt{g\delta} H_P^{-3/2} \Big(\frac{l_m}{2}\Big)^2 (\nabla^\star - \nabla_p)^{3/2} \tag{11}$$

Now we only need to find the value of $(\nabla^\star$ to be able tu update the temperature when convection in included. This is done by solving exercises 5.11 to 5.13 in the lecture notes. And results in equation

Exercise 5.11

Insert Eq. 5.78 and Eq. 5.80 into Eq. 5.79 to get an expression that contains an expression for $(\nabla^\star - \nabla_p)$ as a function of $\nabla^\star$ and ∇_{stable}.

Equation 5.79:

$$F_R + F_C = \frac{16\sigma T^4}{3\kappa\rho H_p}\nabla_{stable}. \tag{12}$$

Equation 5.78:

$$F_C = \rho C_p T \sqrt{\rho\delta} H_p^{-\frac{3}{2}} (\frac{lm}{2})^2 (\nabla^\star - \nabla_p)^{\frac{3}{2}} \tag{13}$$

Equation 5.80:

$$F_R = \frac{16\sigma T^4}{3\kappa\rho H_p}\nabla^\star \tag{14}$$

Inserting equation 5.78 and equation 5.80 into equation 5.79 results in

$$\frac{16\sigma T^4}{3\kappa\rho H_p}\nabla^\star + \rho C_p T \sqrt{\rho\delta} H_p^{-\frac{3}{2}} (\frac{lm}{2})^2 (\nabla^\star - \nabla_p)^{\frac{3}{2}} = \frac{16\sigma T^4}{3\kappa\rho H_p}\nabla_{stable} \tag{15}$$

which can re-arrange in the following manner:

$$\begin{aligned}(\nabla^\star - \nabla_p)^{\frac{3}{2}} &= \frac{16\sigma T^4}{3\kappa\rho H_p}(\nabla_{stable} - \nabla^\star)\frac{4H_p^{\frac{3}{2}}}{\rho C_p \sqrt{g\delta} l_m^2} \\ &= \frac{U}{l_m^2}(\nabla_{stable} - \nabla^\star)\end{aligned} \tag{16}$$

I have here used the substitution $U = \frac{64\sigma T^3}{3\kappa\rho^2 C_p}\sqrt{\frac{H_p}{g\delta}}$.

Exercise 5.12

Insert Eq. 5.71 into

$$(\nabla_p - \nabla_{ad}) = (\nabla^\star - \nabla_{ad}) - (\nabla^\star - \nabla_p) \tag{17}$$

and use Eq. 5.76, to get a second order equation for $(\nabla^\star - \nabla_p)^{1/2}$. Express the solution ξ to the resulting equation as a function of $\nabla^\star$, ∇_{ad}, U and the geometrical constant in front of U in the equation above, where

$$U = \frac{64\sigma T^3}{3\kappa\rho^2 C_p}\sqrt{\frac{H_p}{g\delta}}$$

Argue why this equation has only one viable solution.

Equation 5.71:

$$(\nabla_p - \nabla_{ad}) = \frac{32\sigma T^3}{3\kappa\rho^2 C_p v}\frac{S}{dQ}(\nabla^\star - \nabla_p) \tag{18}$$

Equation 5.76:

$$v = \sqrt{\frac{g\delta l_m^2}{4H_p}}(\nabla^\star - \nabla_p)^{1/2} \tag{19}$$

Inserting these relations into equation (17) we get the following:

$$\frac{32\sigma T^3}{3\kappa\rho^2 C_p}\sqrt{\frac{4H_p}{g\delta l_m^2}}(\nabla^\star - \nabla_p)^{-1/2}\frac{S}{dQ}(\nabla^\star - \nabla_p) = (\nabla^\star - \nabla_{ad}) - (\nabla^\star - \nabla_p) \tag{20}$$

This equation can be re-written as

$$(\nabla^\star - \nabla_p)^{1/2}(\frac{US}{dQ} + (\nabla^\star - \nabla_p)^{1/2}) - (\nabla^\star - \nabla_{ad}) = 0. \tag{21}$$

We now substitute $(\nabla^\star - \nabla_p)^{1/2} = \xi$ and $\frac{US}{dQ} = K$ and end up with the expression

$$\xi^2 + \xi K - (\nabla^\star - \nabla_{ad}) = 0 \tag{22}$$

This is a second order polynomial on the form $ax^2 + bx + c$ which can be solved by using the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Performing this on equation (22) we and up with

$$\xi = \frac{-K \pm \sqrt{K^2 + 4(\nabla^\star - \nabla_{ad})}}{2} \tag{23}$$

We know that for convection to take place we need to fulfill the relationship $\nabla^\star > \nabla_p$. For this to be true then we must demand that $\xi > 0$. Since our constant K is always positive the only solution that can fulfill $\nabla^\star > \nabla_p$ is

$$\xi = \frac{-K + \sqrt{K^2 + 4(\nabla^\star - \nabla_{ad})}}{2} \tag{24}$$

Exercise 5.13

Use your solution to Exerc. 5.12 into your solution of Exerc. 5.11 to eliminate ∇ except in the form of ξ used in Exerc. 5.12.

From Exercise 5.12 we have $\nabla^\star = \xi^2 + \xi K + \nabla_{ad}$. When inserting this into equation (16) and substituting $\frac{U}{l_m^2} = B$ we get

$$\frac{\xi^3}{B} + \xi^2 + \xi K + (\nabla_{ad} - \nabla_{stable}) = 0 \tag{25}$$

This is a third order polynomial with up to three real roots. To solve for ξ I start with examining the discriminant to see how many real roots ξ has. The discriminant for a general third degree polynomial takes the form

$$\Delta = 18abc - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2 \tag{26}$$

where

- $\Delta > 0 \rightarrow$ Three real roots
- $\Delta = 0 \rightarrow$ A multiple root and all it's roots are real
- $\Delta < 0 \rightarrow$ One real root

Inserting our own variables in equation(26) results in

$$\Delta = 68(\nabla_{ad} - \nabla_{stable}) - 240R^2 - \frac{27(\nabla_{ad} - \nabla_{stable})^2}{R^2} \tag{27}$$

Since $(\nabla_{ad} - \nabla_{stable})$ is negative for the case of convention (which is the only case where we are interested to solve this equation) we can clearly see that the discriminant is negative. Since we now know that there is only one real root to ξ we can solve for this root. The general solution for one of the roots is given by

$$x_k = -\frac{1}{3a}(b + \zeta^k C + \frac{\Delta_0}{C}) \tag{28}$$

for $k = 0, 1, 2$. The ζ is the variable used when searching for complex roots. Since $k = 0$ for the real root this variable disappears. Furthermore

$$C = \sqrt[3]{\frac{\Delta_1 \pm \sqrt{\Delta_1^2 - 4\Delta_0^3}}{2}}, \tag{29}$$

$$\Delta_0 = b^2 - 3ac, \tag{30}$$

and

$$\Delta_1 = 2b^3 - 9abc + 27a^2d. \tag{31}$$

Starting to solve for Δ_0 and Δ_1 results in

$$\begin{aligned} \Delta_0 &= 1 - 3\frac{K}{B} \\ &= 1 - 12 \\ &= -11 \end{aligned}$$

$$\begin{aligned} \Delta_1 &= 2 - 9\frac{B}{K} + 27\frac{(\nabla_{ad} - \nabla_{stable})}{B^2} \\ &= -34 + 27\frac{(\nabla_{ad} - \nabla_{stable})}{B^2} \end{aligned} \tag{32}$$

Inserting these values into equation (29) results in

$$\begin{aligned} C &= \sqrt[3]{\frac{-34 + 27\frac{(\nabla_{ad} - \nabla_{stable})}{B^2} \pm \sqrt{(-34 + 27\frac{(\nabla_{ad} - \nabla_{stable})}{B^2})^2 + 5324}}{2}} \\ &= \sqrt[3]{\frac{-34 + 27\frac{(\nabla_{ad} - \nabla_{stable})}{B^2} \pm \sqrt{6480 - 1836\frac{(\nabla_{ad} - \nabla_{stable})}{B^2} + 729\frac{(\nabla_{ad} - \nabla_{stable})^2}{B^4}}}{2}} \end{aligned} \tag{33}$$

We can now finally find a solution for the real root of ξ using equation (28). The final result is given by

$$\xi = \frac{B}{3}(\frac{11}{C} - C - 11) \tag{34}$$

Unfortunately, this solution gives me numerical difficulties when applying it in the Python programme. I therefore used Numpy's "poly1d" and "roots" functions to define the polynomial and to find it's roots. The code that solved the polynomial then took the following form:

```
def xi(self, U, lm, nabla_ad, nabla_st):
        B = U/(lm**2)
        K = 4*B
        p = np.poly1d([1./B, 1.0, K, nabla_ad-nabla_st])
        r = np.roots(p)
        r = r[np.isreal(r)]
        return np.linalg.norm(r)
```

The line "$r = r[np.isreal(r)]$" comes from the fact that the discriminant is less than zero, which means I can exclude all imaginary solutions.

We now have all we need to apply convective energy transport into our code.

II Coding the convective energy transport

In the beginning of the integration loop there has to be a check to see if the temperature gradient of the star is larger than that of the adiabatic parcel. If so, then we must take convection into account. To start with I have to write in methods which calculates all three temperature gradients, ∇_{stable}, $\nabla^{\star}$ and ∇_{ad}.

∇_{ad} is given by equation 5.44 in the lecture notes.

To find $\nabla^{\star}$ I first need to solve the third degree polynomial as shown in the solution to exercise 5.13. When the solution is found I can use the third degree polynomial with the correct value of ξ and solve for $\nabla^{\star}$.

To find ∇_{stable} I can use equation 5.79 in the lecture notes and set it equal to the total flux.

$$\begin{aligned} \frac{L}{4\pi r^2} &= \frac{16\sigma T^4}{3\kappa\rho H_P}\nabla_{stable} \\ \nabla_{stable} &= \frac{3\kappa L\rho H_P}{64\pi r^2\sigma T^4} \end{aligned} \tag{35}$$

The code for these methods look like the following:

```
    def nabla_ad(self, P, delta, T, rho):
        """Methods which returns the temperature gradient for the
           adiabatic parcel """
        return P*delta/(T*rho*self.Cp)

    def nabla_st(self, L, K, rho, Hp, r, T):
        """Methods which returns the temperature gradient for the
           parcel at stable conditions"""
        return 3*L*K*rho*Hp/(64*np.pi*r**2*self.sigma*T**4)

    def xi(self, U, lm, nabla_ad, nabla_st):
        """Methods which returns the solution to the third degree polynomial
        as described in the solution to exercise 5.13"""
        B = U/(lm**2)
        K = 4*B
        p = np.poly1d([1./B, 1.0, K, nabla_ad-nabla_st])
        r = np.roots(p)
        r = r[np.isreal(r)]
        return np.linalg.norm(r)

    def nabla_star(self, U, lm, nabla_ad, nabla_st):
        """Methods which returns the temperature gradient for the
           star """
        xi = self.xi(U, lm, nabla_ad, nabla_st)
        return xi**2 + xi*(4*U/lm**2) + nabla_ad
```

A simple if-else check in the beginning of the loop will check if the conditions demands convective energy transport.

```
    if nabla_st[i] > nabla_ad[i]:
        """If true then we take convection into account """
        Fc, Fr = self.calculate_convection(U, lm, nabla_ad[i], nabla_st[i],
        nabla_star[i], rho[i], T[i], g, Hp, K, M[i], P[i], r[i])
        dTdm = -T[i]*self.G*M[i]/(4*P[i]*r[i]**4*np.pi)*nabla_star[i]

        Fc_frac[i] = Fc/(Fr + Fc)
        Fr_frac[i] = Fr/(Fr + Fc)
    else:
        """Continue with the same algorythms as in project 1 """
        nabla_star[i] = nabla_st[i]
        dTdm = 0
        Fc_frac[i] = 0
        Fr_frac[i] = 1
```

Finally I have to change the temperature calculation if I'm currently in a convective area:

```
    M[i+1] = M[i] + dm
    r[i+1] = r[i] + dm*1.0/(4*np.pi*r[i]**2*rho[i])
    P[i+1] = P[i] + dm*((-self.G*M[i])/(4*np.pi*r[i]**4))
    L[i+1] = L[i] + E[i]*dm
    T[i+1] = T[i] + dm*((-3*K*L[i])/(256*np.pi**2*self.sigma*r[i]**4*T[i]**3))
    if nabla_st[i] > nabla_ad[i]:
        T[i+1] = T[i] + dm*dTdm
    rho[i+1] = self.find_rho(P[i+1], T[i+1])
```

This last temperature correction dTdm comes from the general expression of $\frac{\partial T}{\partial m}$ but now the luminosity is replaced by luminosity when taking convection into account.

$$\begin{aligned}\frac{\partial T}{\partial m} &= \frac{-3\kappa L}{256\pi^2\sigma r^4 T^3} \\ &= \frac{-3\kappa}{256\pi^2\sigma r^4 T^3}\frac{4\pi r^2 16\sigma T^4}{3\kappa\rho H_P}\nabla^\star \\ &= \frac{-GmT}{4\pi r^4 P}\nabla^\star\end{aligned}$$

III Sanity checks, plots

We were given sanity checks both in terms of values and in terms of plots for this project. I have chosen only to account for the plots since they them selves if correct validates the sanity check for the values shown in example 5.1 in the lecture notes. The plot over the three temperature gradients with the same initial conditions as in the sanity check yielded the following result:

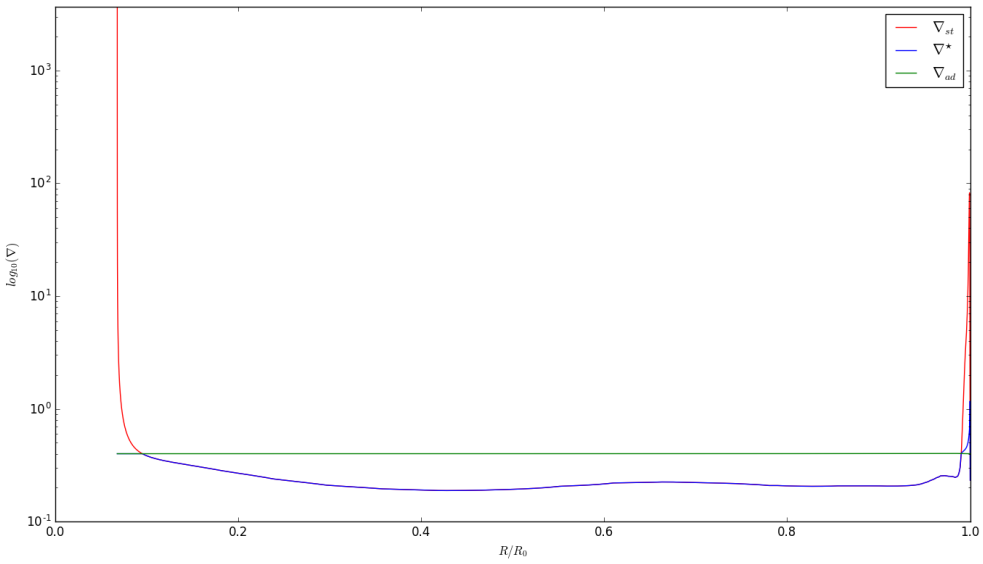

Figure 1: The temperature gradients of the star with initial conditions as described in the sanity check.

When comparing with the plot made available in they lecture notes the look more or less exactly the same. The curve of the temperature gradient of the star in the lecture notes looks more sporadic in the mid section than mine does. This I believe is due to the fact that the code for the variable step length used to produce the sanity check in the lecture notes allows for higher step sizes than mine does.

The circular plot over the convective zones with the same initial conditions as in the sanity check yielded the following result:

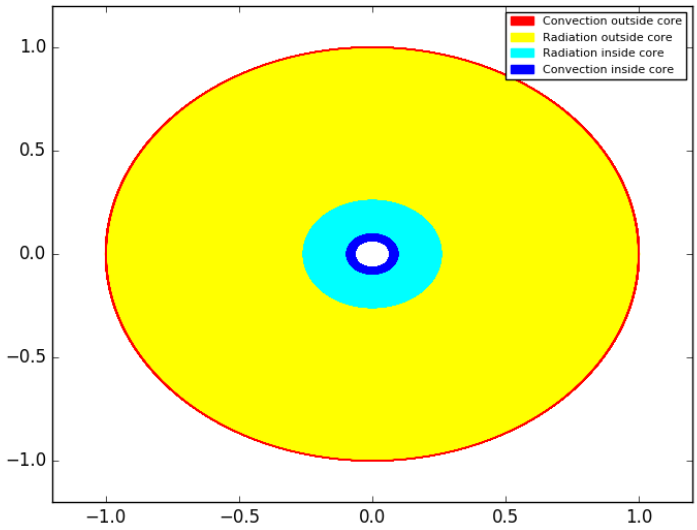

Figure 2: Convective and radiative zones shown in a cross section of the star.

When comparing with the results from the sanity check in the lecture notes I think it's safe to say that the results are the same.

We can now confirm that the code works as it should after convection has been applied.

IV Initial condition vs convection zone

As stated in the beginning of the convection section of this project, in general, larger stars have convective cores and radiative outer layers and small stars have radiative cores and convective outer layers. However, we have only included the P-P chain in this project. In addition we have maid a lot of other simplifications such as constant number densities through out the star, ideal gas and no heat conduction to mention some, so a correct physical interpretation of the models behaviour might be hard to achieve. We can however make some assumptions based of the mathematical formulas used.

IV.1 Changing the initial density

We only allow convection when $\nabla_{stable} > \nabla_{ad}$. ∇_{stable} is proportional to the density while ∇_{ad} is inverse proportional to it. This should mean that the higher the initial density, the larger the convection zone outside the core should be. When applying the different initial densities ($5\rho_0$, $10\rho_0$ and $50\rho_0$ where ρ_0 is the initial density given in project 2) to the code the following cross sections were generated:

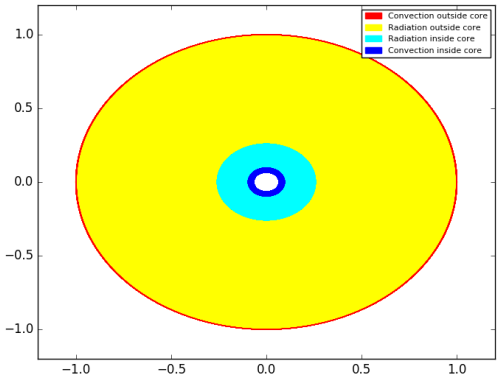

Figure 3: Cross section of the star with initial parameters as described in project 2.

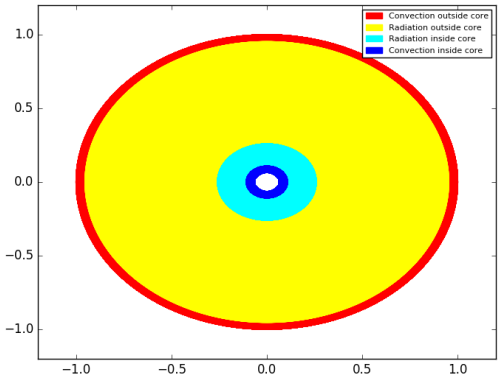

Figure 4: Cross section of the star with initial parameters as described in project 2 except the initial density which is now $5\rho_0$.

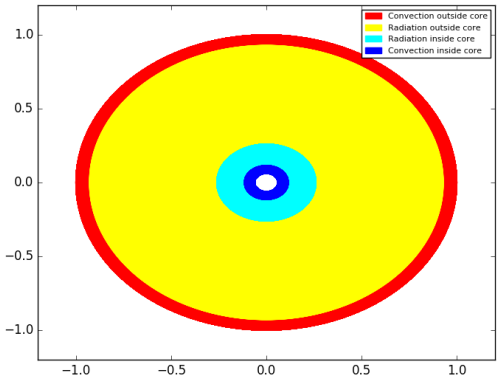

Figure 5: Cross section of the star with initial parameters as described in project 2 except the initial density which is now $10\rho_0$.

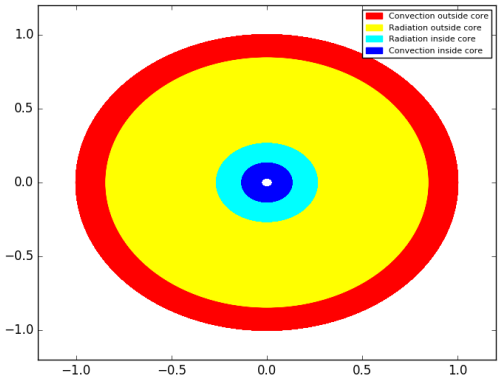

Figure 6: Cross section of the star with initial parameters as described in project 2 except the initial density which is now $50\rho_0$.

As we can see by figures 3-5 the convection zones outside the core does indeed become larger with larger initial density. It could also seem as if the convection zone inside the core becomes larger as well, but this is only due to the fact that with higher initial densities but constant total mass the radius

converge to a value closer to zero. Thus the convention zone at the core is by some means larger, but only since the code manages to show display it closer to $r/r_0 = 0$.

IV.2 Changing the initial temperature

Since ∇_{stable} inverse proportional to T^4 and ∇_{ad} is inverse proportional to T I would expect that for high temperatures the convection zone outside the core would get smaller since high temperatures would give us a small ∇_{stable} which means that the convection test $\nabla_{stable} > \nabla_{ad}$ will not get fulfilled. When applying the different initial temperatures ($0.1 * T_0$, $5 * T_0$ and $10T_0$ where T_0 is the initial temperature given in project 2) to the code the following cross sections were generated:

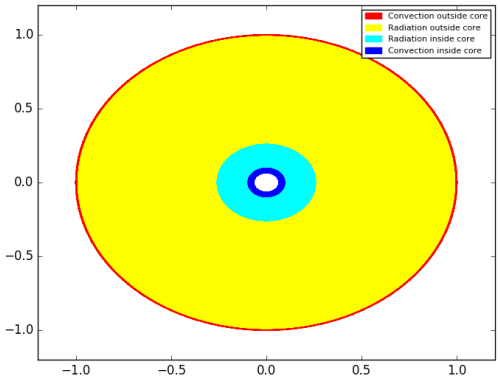

Figure 7: Cross section of the star with initial parameters as described in project 2 except the initial temperature which is now $0.1\rho_0$.

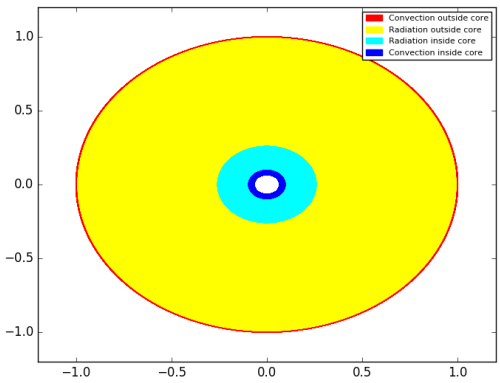

Figure 8: Cross section of the star with initial parameters as described in project 2.

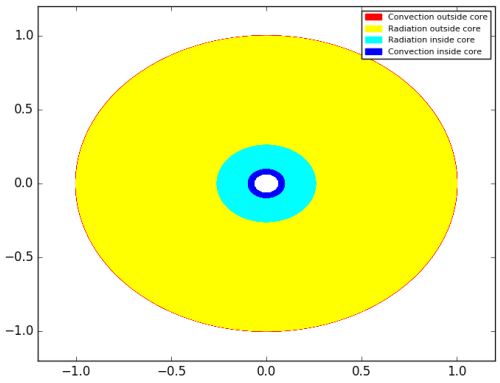

Figure 9: Cross section of the star with initial parameters as described in project 2 except the initial temperature which is now $5\rho_0$.

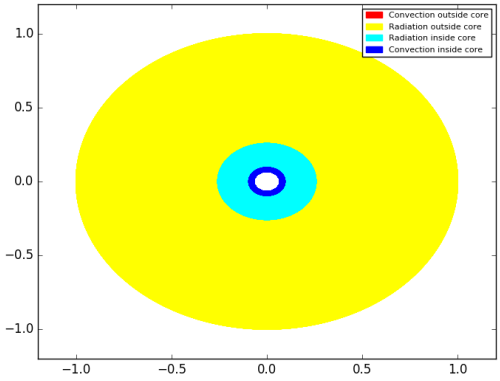

Figure 10: Cross section of the star with initial paramteters as described in project 2 except the initial temperature which is now $10\rho_0$.

We can clearly see from figures 7-10 that as the initial temperature value increases the convection zone outside the core diminishes, as expected. It would also seem like changing the initial temperature has no, or at least very little effect in the range chosen, in weather or not the radius converges closer to zero or not.

IV.3 Changing the initial radius

If we start with a relatively small radius, I would expect a large temperature gradient as we move in closer to the star's core. If the star's temperature gradient is large then we would expect a large

convection zone for these initial radius values, specially when considering that ∇_{ad} is not directly dependent of radius. When applying the different initial radius ($0.1 * R_0$, $5 * R_0$ and $10R_0$ where R_0 is the initial radius given in project 2) to the code the following cross sections were generated:

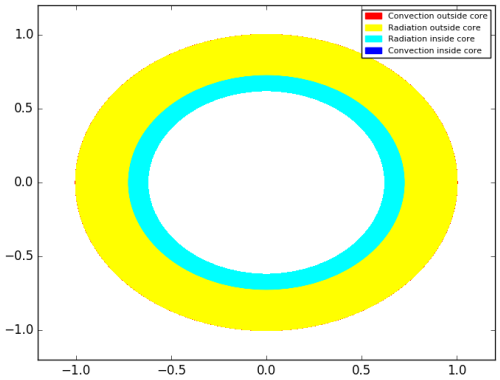

Figure 11: Cross section of the star with initial parameters as described in project 2 except the initial radius which is now $0.1T0$.

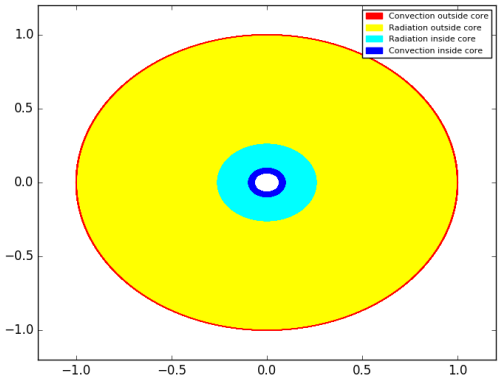

Figure 12: Cross section of the star with initial parameters as described in project 2.

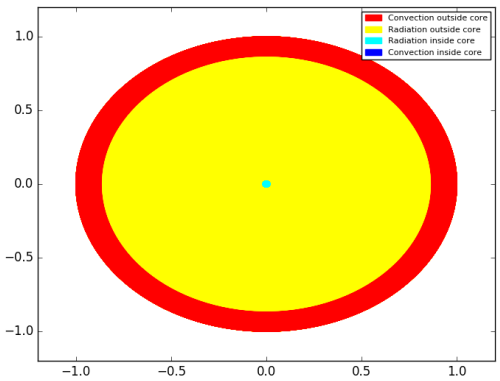

Figure 13: Cross section of the star with initial parameters as described in project 2 except the initial radius which is now $5T_0$.

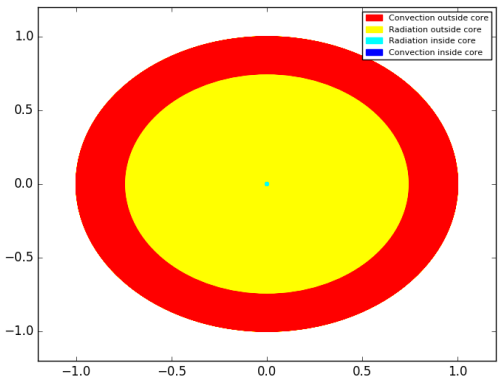

Figure 14: Cross section of the star with initial parameters as described in project 2 except the initial radius which is now $10T_0$.

Again we can see that our predictions came true. When the initial radius of the star increases, so does the size of the convection zone outside the core. We can also see that for small initial radius we do not get very close to $r/r_0 = 0$

IV.4 Creating a larger convection zone

Should we want to create a large outer convection zone in our final star model, we could do one or more of three things:

1. Increase the initial density as shown in section "Changing the initial density
2. Decrease the initial temperature as shown in section "Changing the initial temperature"
3. Increase the initial radius as shown in section "Changing the initial radius"

Bare in mind that these parameters are linked, ie. increasing the radius might not increase the outer convection zone if you at the same time decrease the initial radius and so on.

V Final model

As in project 1, I chose to implement a "least squares model" to find a final model which had L/L_0, M/M_0 and R/R_0 all ending up less than 5%. I quickly ran into trouble in this part though as it was not easy to find a solution that had both values going towards zero and had an outer convection zone stretching to 15% of the total radius of the star. I ran the program maybe 50 times with different parameter choices until I realized that it was pointless. To be able to create a star model with relatively realistic behaviour, like the temperature rising steadily as the radius got smaller I needed to make the code more efficient. One problem I encountered was the energy production rising radically as I approached the center (this turned out to be important to be able to plot $\epsilon(r)/\epsilon_{max}$). This is of course to be expected as the temperature grows closer to the stars centre, but in terms of plotting the normalized total energy produced I wanted a model that never got closer than the area of $R/R_0 \approx 4\%$. In addition I noticed that the code went through a lot of parameter settings that could be disregarded earlier without having to calculate every radius layer. I therefore implemented the following additions to my code:

1. 15% **restriction of the outer convection layer .** If the outer convection layer does not cover at least 15% of the total radius of the star, I disregarded the chosen parameters and continued.

2. **Small variable value restrictions.** If all of the values M/M_0, L/L_0 or R/R_0 are smaller than 0.05 I ended the program. This was to make sure that none of the variables got small enough to "mess up" the plots as well as saving computational time.

3. **Radius vs temperature restriction.** I wanted a star that had a relative even rise in temperature as the radius grew smaller. Of course, I would expect the temperature to follow a type of exponential curve where it grew more and more as the radius became smaller, but I did not want to end up with a more or less constant temperature until the radius was less than 2% either. I therefore ended up with implementing an if-test where I disregarded all parameters that had $T/T_0 < 0.1$ when $R/R_0 < 0.3$. This might not be the most accurate restriction, but is successfully disregarded many of the parameter combinations I was not interested in as well as reduced the computational time drastically.

4. 50% **restriction of the outer convection layer.** I was more unsure about this condition. In general I suppose that there is no problem with a large outer convection zone, but I seemed that a large convection zone did not work well with the 5% restrictions on mass, density and luminosity. I ended up removing this restriction after a while since it was hard to determine whether it was beneficiary or not in term of processing time.

These restriction allowed me to run through thousands of parameter combination in a relative short time, but still I did not find any solutions where the plot of the relative energy production from each of the two PP-chains as a function of radius combined with $\epsilon(r)/\epsilon_{max}$ made any sense. The value of ϵ_{max} grew very large as I approached the center, which meant that you could not tell anything about the how much each off the PP-chain branches contributed to the overall luminosity. It seemed as if I had 2 choices, either restrict the radius goal to $0.03 < R/R_0 < 0.05$ (or something in that area) to avoid too high energy productions close to the core or to simply ignore this specific plot addition. After numerous attempts of trying the first choice I realized one thing. The reason to include $\epsilon(r)/\epsilon_{max}$ in the plot is to be able to estimate how the contribution of each specific PP-chain effect the overall luminosity. This however could be done in other ways, faster ways. The easiest way of finding out how much each branch contributes is to simply calculate the area under the curves. Since both axis (r and energy produced) are normalized to 1 I integrated the curves using the trapezoidal function in the Numpy library. The results are explained in the caption to figure 20.

My final model resulted in the following parameters:

- $R_0 = 1.65 * R_{innit}$
- $\rho_0 = 40 * rho_{innit}$
- $T_0 = 1.10175 * T_{innit}$

where the "innit" index symbolizes the initial parameters given by project 2 and used in the sanity check. All other variables were the same as in the sanity check.

This parameters produced a star with the following core values:

- $L/L_0 \approx 0$
- $M/M_0 \approx 0.048$
- $R/R_0 \approx 0.003$

These values were produced with 25000 time steps, an initial mass step length of $dm = -10^{26}$ with variable step length activated and p (function value fraction for variable step length) $= 10^{-2}$ for speed. This star has a core ($L/L_0 < 0.995$) stretching out to about 13.8% of R_0, well withing the 10% limit stated in project 2.

When the calculations were finished, the following plots were produced:

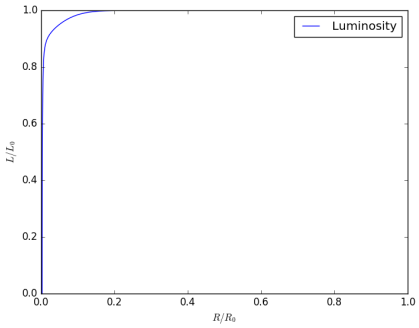

Figure 15: The luminosity as a function of radius. We can see how the luminosity rapidly drops in the region very close to the core as expected.

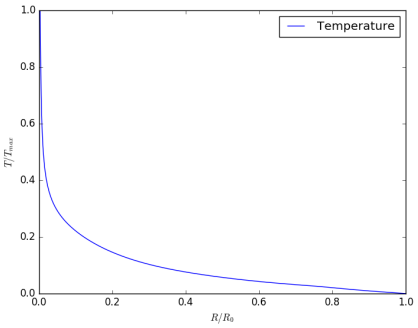

Figure 16: The temperature as a function of radius. We can see how the temperature steadily grows as the radius decreases. It would seem that the radius vs temperature restriction worked as planned.

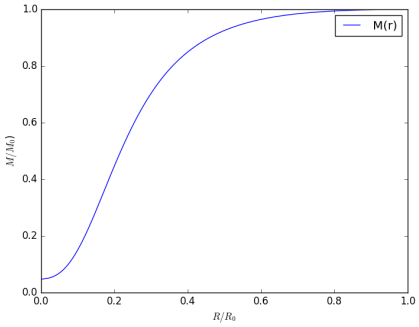

Figure 17: The mass as a function of radius. We can see how the mass steadily decreases as the radius decreases.

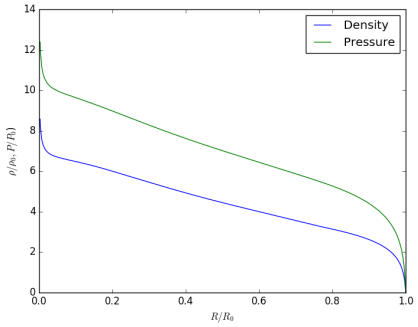

Figure 18: The density and pressure as a function of radius. We can see how the both density and pressure steadily grows as the radius decreases.

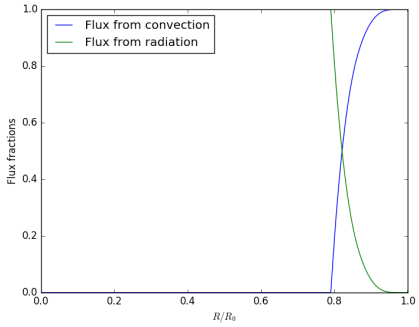

Figure 19: The fractions of the energy being transported by convection and radiation as a function of radius. I the outermost regions of the star convection is dominant. As the radius drops the radiation fraction increases until around $R/R_0 \approx 20\%$ where the convection stops.

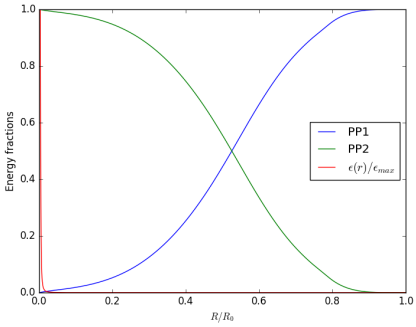

Figure 20: The energy fractions of PP-1 and PP-2 as a function of radius. When close to the edge of the star (low temperatures) PP-1 is dominant while close to the core (high temperatures) PP-2 is dominant. We can also see the $\epsilon(r)/\epsilon_{max}$ plot. As stated above, the energy production increases rapidly close to the core due to the high temperature. This is why $\epsilon(r)/\epsilon_{max}$ peaks so much at $R/R_0 \approx 0$. When using the trapezoidal function I found the PP1 chain being greater than PP2 49% of the "time". Looking at $\epsilon(r)/\epsilon_{max}$ however we can see that PP2 constitutes for almost all of the total energy produced throughout the star.

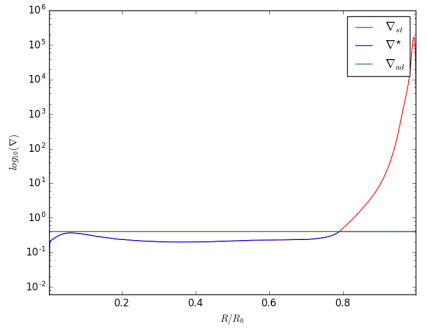

Figure 21: The different temperature gradients (∇-functions) as a function of radius. As shown in the figure, ∇_{stable} is bigger than ∇_{ad} in the convection region as expected, as well as ∇_{ad} being constant through out the star.

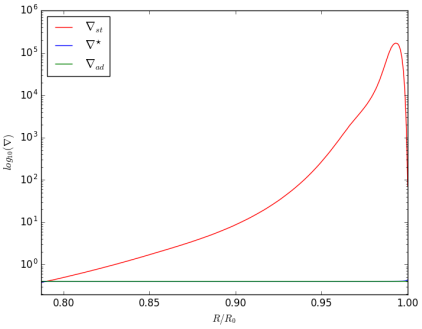

Figure 22: Figure 21 zoomed in at the convection zone.

VI Discussion

The main problem in this project for me was to find parameters fitting the demands stated by the assignment in addition to achieve a relative physical result. As previously stated in section III, "Final model", I spent a fair amount of time trying to produce a star according to my wishes. The restrictions I implemented helped reduce the processing time dramatically, but even then I ran the "least squares model" over night to iterate over thousands of possible parameter combinations. The results from section IV, "Initial condition vs convection zone", helped in the beginning before I implemented the restrictions but yielded no acceptable results. When I finally realized that the effects of the $\epsilon(r)/\epsilon_{max}$-plot could be disregarded (at least I hope it could) I only had to run the program once more to find the parameter combination I ended up with. I am quite satisfied with the final model though. Not

only does it have "well behaving" curves for mass and temperature, it also lacks an inner convection zone, which stated in section I, "Convection", is a hallmark for relative small stars.
I had some trouble distinguishing between the different ∇s, since in the lecture notes both non-indexed and indexed variations were used. I think I got them all right though.
I also had an error in the code from project 1 where I calculate the energy. I had forgotten to scale some of the rates which led to the usage of too many elements. This has been corrected in this project, all though the effect of this correction were so small I did not even notice them in the final model.

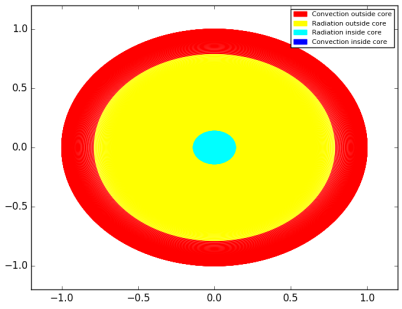

Figure 23: Cross section of the star showing the different energy transportation zones. Note that there is no visible inner core convection zone in this model.

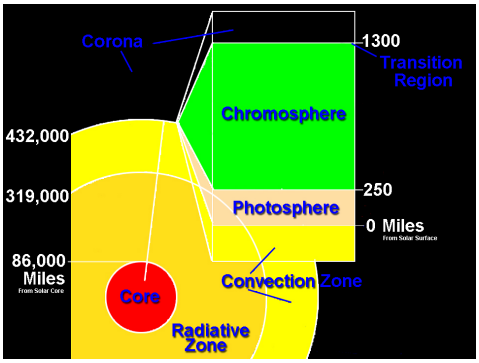

Figure 24: Cross section of the sun to compare with figure 23. Picture is taken from NASA[2]

When comparing the cross section of my model as seen in figure 23 to that of the sun as seen in figure 24 there are actually quite a few resemblances. It would seem that the fraction of the outer convection zone, the fraction of the radiation zone outside of the core and the core itself relative to the radius of the star matches the sun's. That being said, we have not paid any mind the the effects of the star's corona, chromosphere or photosphere in this project, and the similarities might just be a fluke.

In the caption to figure 20 I wrote that the PP1 chain was greater than the contribution from the PP2 chain approximately 49% of the "time". In the lecture notes it says that the PP1 chain should be around 69%, all though I assume this is only valid for the sun, not necessarily other stars. Since my final star model is bigger, warmer and more dense than the sun, I would expect the PP1 contribution to be smaller than 69% since the PP2 chain becomes more active in higher temperatures. I therefore have no problem with this deviation from the solar fractions. Indeed, in addition the PP2 chain produces more energy when we get close to the star's core. Since I only get $R/R_0 = 0.003$ there is still a lot of energy left to be produced before we are truly at the center. As displayed in figure 20 the $\epsilon(r)/\epsilon_{max}$ plot clearly show that the energy production increases rapidly as the radius decreases, specially close to the core. If I had simulated this model to $R/R_0 = 0$ I would expect the fraction of PP2 would increase.

I know that the PP2 chain dominates the energy production at temperatures higher than 14MK[3]. The problem for here was to evaluate what "dominates" means. As we can see from figure 20 the PP2 chain dominates (is much greater than PP1) the energy production at radius close to the core (hard to estimate precisely here but I would say around $0.1 * R/R_0$). When we now look at figure 25 we can see that the temperature is starting to rapidly increase in the area of $T \approx 1.4 * 10^7 K = 14MK$ and $R/R_0 \approx 0.1$ which seems to match the prediction well. However, if "dominates" in this particular situation simply means "is greater than" then there seems to be a mismatch. The PP2 chain contributes to a bigger fraction of the total energy production already at $R/R_0 \approx 0.5$, which does not match the plot of $T(R/R_0)$ in figure 25. I can't find any problems with the code to explain this if the latter interpretation of "dominates" is valid.

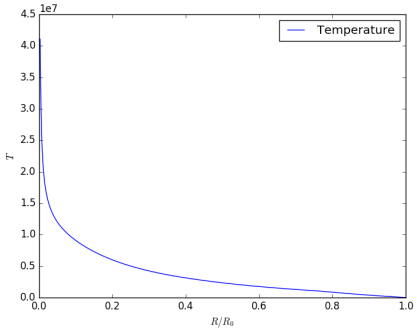

Figure 25: The temperature in unites Kelvin as a function of radius.

VII Source code files

The source code can be found at
`https://github.com/erikalev/AST3310/tree/master/Project%202`
In addition the source code has been sent in the same mail as this project

References

[1] B.V. Gudiksen (2017), textitAST3310: Astrophysical plasma and stellar interiors `http://www.uio.no/studier/emner/matnat/astro/AST3310/v17/beskjeder/notes/notesv1.pdf`

[2] `https://www.nasa.gov/mission_pages/iris/multimedia/layerzoo.html`

[3] `https://en.wikipedia.org/wiki/Proton%E2%80%93proton_chain_reaction`

VIII Source code

```
from scipy import interpolate as inter
import numpy as np
import matplotlib.pyplot as plt
import sys
from numpy.polynomial import Polynomial as P

class project2:
    def __init__(self, L_0, M_0, R_0, rho_0, T_0, X, Y, Y3, Y4, Z, Z7Li, Z7Be):
        self.R_0 = R_0                          # initial radius
        self.L_0 = L_0                          # initial luminosity
        self.M_0 = M_0                          # initial mass
        self.rho_0 = rho_0                      # initial density
        self.T_0 = T_0                          # initial temperature

        self.L_sun = 3.846e26                   # Sun's Luminosity [W]
        self.R_sun = 6.96e8                     # Sun's radius [m]
        self.M_sun = 1.989e30                   # Sun's mass [kg]
        self.rho_star_avg = 1.408e3             # Sun's average density [kg m^-3]

        self.X, self.Y, self.Y3, self.Y4, self.Z, self.Z7Li, self.Z7Be = X, Y, Y3, Y4,
             Z, Z7Li, Z7Be # initial mass fractions
        self.c = 299792458                      # Speed of light [m/s]
        self.sigma = 5.670367e-8                # Stefan-Boltzmann constant [J m^-2 s
            ^-1 K^-4]
        self.k_b = 1.38064852e-23               # Boltzmann constant [m^2 kg s^-2 K
            ^-1]
        self.u = 1.0/(2*X + 3.0/4.0*Y + Z/2.0)  # Mean molecular weight "dimensionless
            "
        self.m_u = 1.6605e-27                   # Atomic mass unit "dimensionless"
        self.G = 6.67408e-11                    # Gravitational constant [m^3 kg-1 s
            -2]
        self.Cp = 5./2.*self.k_b/(self.u*self.m_u)
        self.delta = 1

        """ Performing the linear 2D interpolation o6f the opacity values. These will
            be used
        later on but performing the interpolation in the innit function allows us to
            just
        call the ready function later on. We therefore only have to interpolate once
            which
        greatly improves the time performance of the programme"""

        data = np.genfromtxt("/home/erik/FAG/AST3310/opacity.txt")  # Collecting all
            data in one array
        logT = data[1:,0]                                           # logT-values
        logR = data[0,1:]                                           # logR-values
        n_T = len(logT)                                             # Number of logT-
            values
        n_R = len(logR)                                             # Number of logR-
            values
        self.f = inter.interp2d(logR, logT, data[1:, 1:], bounds_error = False)
                    # Performing the interpolation

    def opacity(self, T, rho):
        """ Function which takes emperature and density in SI-units and returns the
        opacity value corresponding to these values from the interpolation-function
        in the innit-function"""

        rho = rho/1000.0                            # Converting the density to cgs
        R = np.log10(rho/((T/1e6)**3))              # As defined by Appendix D
        T = np.log10(T)

        return float(10**(self.f(R, T))/10.0)     # Returning the opacity in SI-units

    def PP1_chain(self, r33, Q33):
        """Last reaction in the PP1 chain. The first 2 reaction are the same
        for both PP1 and PP2 and can therefore be calculated outside this function.
        Takes arguments reaction rates [kg s^-1] and energy produced in unit [MeV] """
        return r33*Q33
```

```
    def PP2_chain(self, r34, r17, re7, Q34, Q17, Qe7):
        """Last three reactions in the PP2 chain. The first 2 reaction are calculated
        outside this function. Takes arguments reaction rates [kg s^-1] for all three
        reaction and energy produced in units [MeV] for all three reactions."""
        return r34*Q34 + r17*Q17 + re7*Qe7

    def energy_produced(self, T, rho):
        """ Method for calculating the energy production in the PP1 and PP2 chains.
        Takes arguments temperature and density, both in SI units, and returns
        the energy pr second pr kg [J kg^-1 s^-1]"""

        MeVToJoule = 1.602e-13          # Converting MeV to Joule
        NA = 6.022e23                   # Avogadros number (dimensionless)
        T9 = float(T)/1e9                       # Units used to find the lambda-values
            [K]

        # Proportionality functions [m^-3 s^-1]
        lambda_pp = (4.01e-15*T9**(-2.0/3)*np.exp(-3.380*T9**(-1.0/3))*(1 + 0.123*T9
            **(1.0/3.0) + 1.09*T9**(2.0/3.0) + 0.938*T9))/NA/1e6
        lambda_33 = (6.04e10*T9**(-2.0/3.0)*np.exp(-12.276*T9**(-1.0/3.0))*(1 + 0.034*
            T9**(1.0/3.0) - 0.522*T9**(2.0/3.0) - 0.124*T9 + 0.353*T9**(4.0/3.0) +
            0.213*T9**(-5.0/3.0)))/NA/1e6;
        T9star = T9/(1.0 + 4.95e-2*T9);
        lambda_34 = (5.61e6*T9star**(5.0/6.0)*T9**(-3.0/2.0)*np.exp(-12.826*T9star
            **(-1.0/3.0)))/NA/1e6;
        lambda_e7 = (1.34e-10*T9**(-1.0/2.0)*(1 - 0.537*T9**(1.0/3.0) + 3.86*T9
            **(2.0/3.0) + 0.0027/T9*np.exp(2.515e-3/T9)))/NA/1e6;
        T9star2 = T9/(1 + 0.759*T9);
        lambda_17 = (1.096e9*T9**(-2.0/3.0)*np.exp(-8.472*T9**(-1.0/3.0)) - 4.830e8*
            T9star2**(5.0/6.0)*T9**(-3.0/2.0)*np.exp(-8.472*T9star2**(-1.0/3.0)) + 1.06
            e10*T9**(-3.0/2.0)*np.exp(-30.442*T9**(-1)))/NA/1e6;

        # Number densities
        n_p = self.X*rho/self.m_u
        n_He3 = self.Y3*rho/(3*self.m_u)
        n_He4 = self.Y4*rho/(4*self.m_u)
        n_Be7 = self.Z7Be*rho/(7*self.m_u)
        n_Li7 = self.Z7Li*rho/(7*self.m_u)
        n_e = n_p + 2*n_He3 + 2*n_He4 + self.Z/2.0

        # If-test to apply the upper limit of Be7
        if (T < 1e6):
            if (NA*lambda_e7 > 1.57e-7/n_e):
                #print "Upper limit for Be7 was acchieved"
                lambda_e7 = 1.57e-7/(n_e*NA)

        #Reaction rates pr unit mass. These are really multiplied with a factor rho. [
            s^(-1) m^(-3)]
        rpp = n_p*n_p/2.0*lambda_pp
        r33 = n_He3*n_He3/2.0*lambda_33
        r34 = n_He3*n_He4*lambda_34
        r17 = n_Li7*n_p*lambda_17
        re7 = n_Be7*n_e*lambda_e7

        #Energys from the different reactions in PP1 and PP2 in units [MeV]
        Qpp = 0.15 + 1.02
        Qpd = 5.49
        Q33 = 12.86
        Q34 = 1.59
        Qe7 = 0.05
        Q17 = 17.35

        # Total power/volume to return
        E = 0
        PP1_E = 0
        PP2_E = 0

        #First 2 reaction in PP1 and PP2 combined
        First_energy_step = rpp*(Qpp + Qpd)*MeVToJoule
```

```
        # PP1 and PP2 Chain
        if (2*r33 + r34) > rpp:
            # Rescaling to avoid to much use of elements
            scale = rpp/(2*r33 + r34)
            r33 = scale*r33
            r34 = scale*r34

        if re7 > r34:
            # Re7 reaction rate is determined by the R34 reaction rate
            re7 = r34

        if r17 > re7:
            # R17 reaction rate is determined by the Re7 reaction rate
            r17 = re7

        # Adding the correspoding energies to each chain
        PP1_E = self.PP1_chain(r33, Q33)*MeVToJoule + First_energy_step*r33/(r33 + r34
            )
        PP2_E += self.PP2_chain(r34, r17, re7, Q34, Q17, Qe7)*MeVToJoule +
            First_energy_step*r34/(r33 + r34)
        E += PP1_E + PP2_E

        return E, PP1_E, PP2_E

    def find_Pr(self, T):
        """ Method which takes temperature as argument and
            returns the radiative pressure in the star """

        return 4*self.sigma/(3*self.c)*T*T*T*T

    def find_rho(self, P, T):
        """ Method which takes total pressure and temperature as arguments and
            returns the density in the star when taking into
            account an ideal gas and P = P_gass + P_radiation"""

        Pg = P - self.find_Pr(T)
        return Pg*self.u*self.m_u/(self.k_b*T)

    def find_P(self, rho, T):
        """ Method which takes density and temperature as arguments and
            returns the total pressure in the star when taking into
            account an ideal gas and P = P_gass + P_radiation
        """
        return rho/(self.u*self.m_u)*self.k_b*T + self.find_Pr(T)

    def plot(self, M, r, L, T, rho):
        """Method which plots some of the achieved values"""

        #plotting the initial value sin subplots
        fig = plt.figure()
        ax = plt.subplot("221")
        ax.set_xlabel("Mass/$M_0$")
        ax.set_ylabel("R/$R_0$")
        ax.plot(M/M[0], r/r[0])

        ax = plt.subplot("222")
        ax.set_xlabel("Mass/$M_0$")
        ax.set_ylabel("L/$L_0$")
        ax.plot(M/M[0], L/L[0])

        ax = plt.subplot("223")
        ax.set_xlabel("Mass/$M_0$")
        ax.set_ylabel("T[MK]")
        ax.plot(M/M[0], T/1e6)

        ax = plt.subplot("224")
        #ax.set_ylim(10**0, 10**1)
        ax.set_xlabel("Mass/$M_0$")
        ax.set_ylabel("$\\rho/rho_0$")
        ax.plot(M/M[0], rho/rho[0])
        plt.subplots_adjust(hspace = .5)
```

```
        plt.subplots_adjust(wspace = .5)
        plt.show()

    def DSS(self, dm, dm1, r, rho, M, E, L, T, K, P, dTdm):
        """ Method to calculate the variable step length. Takes the old and
         the initial dm (dm and dm1) as input together with the variable values
        and returns the new dm
        """

        # Variable step length constant
        p = 1e-2;

        # Establishing variables to perform the variable step length

        drdm = 1.0/(4*np.pi*r**2*rho)
        dPdm = -self.G*M/(4*np.pi*r**4)
        dLdm = E

        # No convection
        if dTdm == 0:
            dTdm =  -3*K*L/(256*np.pi**2*self.sigma*r**4*T**3)

        test = np.zeros(5)
        test[0] = abs(p*r/drdm)
        test[1] = abs(p*P/dPdm)
        test[2] = abs(p*L/dLdm)
        test[3] = abs(p*T/dTdm)
        test[4] = abs(p*M)

        dm = -min(test)

        # Resetting initial dm to avoid to large values
        if dm < dm1:
            dm = dm1

        return dm, dTdm

    def print_int(self, i, dm, rho, L, M, r, P, E, T):
        """Method that just prints the values every designated time step """
        print "i = ", i
        print "dm = ", dm
        print "rho = ", rho
        print "L = ", L
        print "M = ", M
        print "r = ", r
        print "P = ", P
        print "E = ", E
        print "T = ", T
        print " "
        print "L/L_0= ", L/self.L_0
        print "M/M_0= ", M/self.M_0
        print "R/R_0= ", r/self.R_0
        print " "

    def test_small_enough_values(self, L, M, r):
        """ Method which ends the programme when small enough values of L, M and r is
            achieved """
        small_enough_values = False
        if (L/self.L_0 < 0.049):
            if (M/self.M_0 < 0.049):
                if (r/self.R_0 < 0.049):
                    small_enough_values = True

        return small_enough_values

    def test_negative_values(self, M, r, L, i):
        """ Method which checks for negative values when running the code. This is
            only activated
            when the DDS method (dynamic step size) is not """

        if (M[i] < 0):
```

```
            L_lim = L[i-1]/self.L_0
            M_lim = M[i-1]/self.M_0
            R_lim = r[i-1]/self.R_0

            print "Negative mass achieved"
            print "i = ", i
            print "L/L_0 = ", L_lim
            print "M/M_0 = ", M_lim
            print "R/R_0 = ", R_lim
            negative_values = True
            return negative_values

        elif (r[i] < 0):
            L_lim = L[i-1]/self.L_0
            M_lim = M[i-1]/self.M_0
            R_lim = r[i-1]/self.R_0

            print "Negative radius achieved"
            print "i = ", i
            print "L/L_0 = ", L_lim
            print "M/M_0 = ", M_lim
            print "R/R_0 = ", R_lim
            negative_values = True
            return negative_values

        elif (L[i] < 0):
            L_lim = L[i-1]/self.L_0
            M_lim = M[i-1]/self.M_0
            R_lim = r[i-1]/self.R_0

            print "Negative luminosity achieved"
            print "i = ", i
            print "L/L_0 = ", L_lim
            print "M/M_0 = ", M_lim
            print "R/R_0 = ", R_lim
            negative_values = True
            return negative_values
        else:
            negative_values = False
            return negative_values

    def nabla_ad(self, P, delta, T, rho):
        """Method which returns the temperature gradient for the
           adiabatic parcel """
        return P*delta/(T*rho*self.Cp)

    def nabla_st(self, L, K, rho, Hp, r, T):
        """Method which returns the temperature gradient for the
           parcel at stable conditions"""
        return 3*L*K*rho*Hp/(64*np.pi*r**2*self.sigma*T**4)

    def xi(self, U, lm, nabla_ad, nabla_st):
        """Method which returns the solution to the third degree polynomial as
            described in
        the solution to exercise 5.13"""
        B = U/(lm**2)
        K = 4*B
        p = np.poly1d([1./B, 1.0, K, nabla_ad-nabla_st])
        r = np.roots(p)
        r = r[np.isreal(r)]
        return np.linalg.norm(r)

    def nabla_star(self, U, lm, nabla_ad, nabla_st):
        """Method which returns the temperature gradient for the
           star """
        xi = self.xi(U, lm, nabla_ad, nabla_st)
        return xi**2 + xi*(4*U/lm**2) + nabla_ad

    def Fc(self, U, lm, nabla_ad, nabla_st, rho, Cp, T, g, delta, Hp):
        """ Method which return the convective flux """
        xi = self.xi(U, lm, nabla_ad, nabla_st)
```

```
        return rho*Cp*T*np.sqrt(g*delta)*Hp**(-3./2.)*(lm/2.)**2*xi**3

    def Fr(self, T, K, M, rho, Hp, nabla_star):
        """ Method whoch returns the rediative flux """
        return 16*self.sigma*T**4/(3*K*rho*Hp)*nabla_star

    def calculate_convection(self, U, lm, nabla_ad, nabla_st, nabla_star, rho, T, g,
        Hp, K, M, P, r):
        """ This method was basically just to make the code a little bit more
            understandable.
            It only call self.Fc and self.Fr """
        Fc = self.Fc(U, lm, nabla_ad, nabla_st, rho, self.Cp, T, g, self.delta, Hp)
        Fr  = self.Fr(T, K, M, rho, Hp, nabla_star)
        return Fc, Fr

    def outer_convection_restriction(self, i, N, Fc_frac_m1, Fc_frac_i, r_i, r_0):
        """ Method which makes sure the outer convection zone cover atleast 15%
            of the total radius"""
        if i < N/2:
            if Fc_frac_m1 != 0:
                if Fc_frac_i == 0:
                    if r_i/r_0 > 0.85:
                        return True
                    else:
                        return False
                else:
                    return False
            else:
                return False
        else:
            return False

    def T_r_restriction(self, r_i, r_0, T_i, T_0):            # Restriction to the T(r
        ) plot
        """ Method which makes sure that the temperature rises steadily and not
            all at once """
        if r_i/r_0 < 0.3:
            if T_i/T_i < 0.1:
                return True
            else:
                return False
        else:
            return False

    def integrate(self, dm, N, dynamic_step_size = False):
        """ Method which takes the the mass step length dm, number of integration
            points N and the dynamic step size boolean and integrates over N dm-steps.
            If none declared in the call, dymamic step size is automatically turned
            off. The method also checks if there is is negative mass, distance or
            luminosity. If so an error message is printed out and th plot-method is
            called to plot the final results.
        """
        K = self.opacity(self.T_0, self.rho_0)    # Initial Kappa value
        #Initializing vectors to be filled in with values
        L = np.zeros(N)                  # Luminosity
        rho = np.zeros(N)                # Density
        T = np.zeros(N)                  # Temperature
        r = np.zeros(N)                  # Radial distance
        P = np.zeros(N)                  # Pressure
        M = np.zeros(N)                  # Mass
        E = np.zeros(N)                  # Power/volume
        nabla_st = np.zeros(N)           # Temperature gradient stable
        nabla_star = np.zeros(N)         # Temperature gradient star
        nabla_ad = np.zeros(N)           # Temperature gradient adiabatic
        Fr_frac = np.zeros(N)            # Radiative flux fraction
        Fc_frac = np.zeros(N)            # Convective flux fraction
        PP1_E = np.zeros(N)              # Energy from PP1
        PP2_E = np.zeros(N)              # Energy from PP2
        PP1_frac = np.zeros(N)           # Energy fraction from PP1
        PP2_frac = np.zeros(N)           # Energy fraction from PP1
```

```
        # Setting initial values for vectors
        L[0] = self.L_0
        rho[0] = self.rho_0
        T[0] = self.T_0
        r[0] = self.R_0
        M[0] = self.M_0
        E[0] = self.energy_produced(self.T_0, self.rho_0)[0]/self.rho_0
        P[0] = self.find_P(self.rho_0, self.T_0)
        PP1_E[0] = self.energy_produced(self.T_0, self.rho_0)[1]
        PP2_E[0] = self.energy_produced(self.T_0, self.rho_0)[2]
        PP1_frac[0] = PP1_E[0]/(PP1_E[0] + PP2_E[0])
        PP2_frac[0] = PP2_E[0]/(PP1_E[0] + PP2_E[0])

        dm1 = dm                        # controle variable which is used to set dm
            back to it's original value

        negative_values = False         # Boolean variable used to check for negative
            variables when running the code
        small_enough_values = False     # Boolean variable used to check for small
            enough variables when running the code

        for i in range(N-1):
            # updating the opacity value
            K = self.opacity(T[i], rho[i])

            # Updating values needed for the temperature gradients
            g = self.G*M[i]/(r[i]**2)
            alpha = 1
            Hp = P[i]/(g*rho[i])
            lm = Hp*alpha
            U = 64*self.sigma*T[i]**3/(3*K*rho[i]**2*self.Cp)*np.sqrt(Hp/(g*self.delta
                ))

            # Temperature gradients
            nabla_ad[i] = self.nabla_ad(P[i], self.delta, T[i], rho[i])
            nabla_st[i] =  self.nabla_st(L[i], K, rho[i], Hp, r[i], T[i])
            nabla_star[i] = self.nabla_star(U, lm, nabla_ad[i], nabla_st[i])

            if nabla_st[i] > nabla_ad[i]:
                """If true then we take convection into account """
                Fc, Fr = self.calculate_convection(U, lm, nabla_ad[i], nabla_st[i],
                    nabla_star[i], rho[i], T[i], g, Hp, K, M[i], P[i], r[i])
                dTdm = -T[i]*self.G*M[i]/(4*P[i]*r[i]**4*np.pi)*nabla_star[i]
                Fc_frac[i] = Fc/(Fr + Fc)
                Fr_frac[i] = Fr/(Fr + Fc)

            else:
                """Continue with the same algorythms as in project 1 """
                nabla_star[i] = nabla_st[i]
                dTdm = 0
                Fc_frac[i] = 0
                Fr_frac[i] = 1

            # Tests to check that we dont achieve any negative values in M, r or L
            if self.test_negative_values(M, r, L, i) == True:
                break

            # Restriction for the outer convection layer
            if self.outer_convection_restriction(i, N, Fc_frac[i-1], Fc_frac[i], r[i],
                 r[0]) == True:
                break

            # Restriction to the T(r) plot
            if self.T_r_restriction(r[i], r[0], T[i], T[0]) == True:
                break

            # Restriction of the value of the resulting variables
            #if self.test_small_enough_values(L[i], M[i], r[i]) == True:
            #    break

            if dynamic_step_size == True:
```

```
                dm, dTdm = self.DSS(dm, dm1, r[i], rho[i], M[i], E[i], L[i], T[i], K,
                    P[i], dTdm)

            # Euler solvers
            M[i+1] = M[i] + dm
            r[i+1] = r[i] + dm*1.0/(4*np.pi*r[i]**2*rho[i])

            # Possible restriction of the lowest allowed radius fraction
            #if r[i+1]/r[0] < 0.04:
            #    break


            P[i+1] = P[i] + dm*((-self.G*M[i])/(4*np.pi*r[i]**4))
            L[i+1] = L[i] + E[i]*dm
            T[i+1] = T[i] + dm*((-3*K*L[i])/(256*np.pi**2*self.sigma*r[i]**4*T[i]**3))

            # Updating the temperature for convection
            if nabla_st[i] > nabla_ad[i]:
                T[i+1] = T[i] + dm*dTdm

            # Updating density and energy
            rho[i+1] = self.find_rho(P[i+1], T[i+1])
            E[i+1] = self.energy_produced(T[i+1], rho[i+1])[0]/rho[i+1]

            # Energy from PP1 and PP2
            PP1_E[i+1] = self.energy_produced(T[i+1], rho[i+1])[1]/rho[i+1]
            PP2_E[i+1] = self.energy_produced(T[i+1], rho[i+1])[2]/rho[i+1]

            # Fraction from PP1 and PP2
            PP1_frac[i+1] = PP1_E[i+1]/(PP1_E[i+1] + PP2_E[i+1])
            PP2_frac[i+1] = PP2_E[i+1]/(PP1_E[i+1] + PP2_E[i+1])


            # Printing out the values with an appropriate interval
            #if (i%1==0):
            #    self.print_int(i, dm, rho[i], L[i], M[i], r[i], P[i], E[i], T[i])

        return r[:i], M[:i], L[:i], rho[:i], P[:i], T[:i], E[:i], nabla_star[:i],
            nabla_st[:i], nabla_ad[:i], PP1_E[:i], PP2_E[:i], PP1_frac[:i], PP2_frac[:i
            ], Fc_frac[:i], Fr_frac[:i]

if __name__ == "__main__":
    # Setting initial parameters in SI-units
    L_sun = 3.846e26        # Sun's Luminosity [W]
    R_sun = 6.96e8          # Sun's radius [m]
    M_sun = 1.989e30        # Sun's mass [kg]
    rho_star_avg = 1.408e3  # Sun's average density [kg m^-3]

    #Setting initial parameters for bottom of sun's convection zone
    L_0 = 1.0*L_sun
    R_0 = 1.0*R_sun
    M_0 = 1.0*M_sun
    rho_0 = 1.42e-7*rho_star_avg
    T_0 = 5770
    # Setting mass fractions
    X = 0.7
    Y3 = 1e-10
    Y = 0.29
    Y4 = Y - Y3
    Z = 0.01
    Z7Li = 1e-13
    Z7Be = 1e-13

    # Number of integration points
    N = 25000

    # Calling the class and the integrate function
    A = project2(L_0, M_0, 1.65*R_0, 40*rho_0, 1.10175*T_0, X, Y, Y3, Y4, Z, Z7Li,
        Z7Be)
    r, M, L, rho, P, T, E, nabla_star, nabla_st, nabla_ad, PP1_E, PP2_E, PP1_frac,
        PP2_frac, Fc_frac, Fr_frac = A.integrate(-1e26, N, dynamic_step_size = True)
```

```
print "PP1 frac = ", abs(np.trapz(PP1_frac, r/r[0]))
print "PP2 frac = ", abs(np.trapz(PP2_frac, r/r[0]))

# Plotting temperature gradients
plt.plot(r/r[0], nabla_st, "-r", r/r[0], nabla_star, r/r[0], nabla_ad)
plt.legend(["$\\nabla_{st}$", "$\\nabla^{\\star}$", "$\\nabla_{ad}$"], loc="upper
    left")
plt.yscale("log")
plt.xlabel("$R/R_0$")
plt.ylabel("$log_{10}(\\nabla)$")
plt.show()

# Plotting contributions from PP1 and PP2 chains
plt.plot(r/r[0], PP1_frac, r/r[0], PP2_frac, r/r[0], E/max(E))
plt.ylabel("Energy fractions")
plt.xlabel("$R/R_0$")
plt.legend(["PP1", "PP2", "$\\epsilon (r)/\\epsilon _{max}$"], loc="center right")
plt.show()

# Plotting the radiation and convection contribution
plt.plot(r/r[0], Fc_frac, r/r[0], Fr_frac)
plt.ylabel("Flux fractions")
plt.xlabel("$R/R_0$")
plt.legend(["Flux from convection", "Flux from radiation"], loc="upper left")
plt.show()

# Plotting the cross section
fig, ax = plt.subplots()
circle1 = plt.Circle((0, 0), 0, color='red')
circle2 = plt.Circle((0, 0), 0, color='yellow')
circle3 = plt.Circle((0, 0), 0, color='cyan')
circle4 = plt.Circle((0, 0), 0, color='blue')
plt.legend([circle1, circle2, circle3, circle4], ["Convection outside core", "
    Radiation outside core", "Radiation inside core", "Convection inside core"],
    prop={'size':8})
for i in xrange(len(r)):
    if i%10 == 0:
        # Only plot circle every 10 step
        if nabla_st[i] > nabla_ad[i]:
            # Convection
            if L[i]/L[0] < 0.995:
                # Inside core
                circle1 = plt.Circle((0, 0), r[i]/r[0], color='blue', fill=False)
                ax.add_artist(circle1)
            else:
                # Outside core
                circle2 = plt.Circle((0, 0), r[i]/r[0], color='red', fill=False)
                ax.add_artist(circle2)
        else:
            # No convection
            if L[i]/L[0] < 0.995:
                # Inside core
                circle3 = plt.Circle((0, 0), r[i]/r[0], color='cyan', fill=False)
                ax.add_artist(circle3)
            else:
                # Outside core
                circle4 = plt.Circle((0, 0), r[i]/r[0], color='yellow', fill=False
                    )
                ax.add_artist(circle4)
        plt.hold("on")
plt.axis([-1.2, 1.2, -1.2, 1.2])
plt.show()

# Plotting T(r)
plt.plot(r/r[0], T/max(T))
plt.legend(["Temperature"])
plt.xlabel("$R/R_0$")
plt.ylabel("$T/T_{max}$")
plt.show()
```

```
    # Plotting L(r)
    plt.plot(r/r[0], L/L[0])
    plt.legend(["Luminosity"])
    plt.xlabel("$R/R_0$")
    plt.ylabel("$L/L_0$")
    plt.show()

    # Plotting rho(r) and P(r)
    plt.plot(r/r[0], np.log10(rho/rho[0]), r/r[0], np.log10(P/P[0]))
    plt.legend(["Density", "Pressure"])
    plt.xlabel("$R/R_0$")
    plt.ylabel("$\\rho/\\rho_0, P/P_0$)")
    plt.show()

    # Plotting M(r)
    plt.plot(r/r[0], M/M[0])
    plt.legend(["M(r)"])
    plt.xlabel("$R/R_0$")
    plt.ylabel("$M/M_0$)")
    plt.show()

    def least_square(no_values, R_0_min, R_0_max, rho_0_min, rho_0_max, T_0_min,
        T_0_max, N, dm):
        """
        This function performes a least square method
        It takes the arguments for the min and max values of R_0, rho_0 and T_0 where
            no_values is
        the number of points in each list. It also takes the arguments number of
            integration
        points N and step length dm.
        The funciton then prints out the sum of the procentages (last_value/
            first_value) as they
        get closer and closer towards zero together with the factor in front
        """

        R_0_list = np.linspace(R_0_min, R_0_max, no_values)      # list of R_0-values
        rho_0_list = np.linspace(rho_0_min, rho_0_max, no_values)# list of rho_0-
            values
        T_0_list = np.linspace(T_0_min, T_0_max, no_values)      # list of T_0-values

        # Variables to use in the least square method
        R_0_save = 0
        rho_0_save = 0
        T_0_save = 0
        least = 1e6
        for i in range(len(R_0_list)):
            R_01 = R_0_list[i]
            for j in range(len(T_0_list)):
                T_01 = T_0_list[j]
                for k in range(len(rho_0_list)):
                    rho_01 = rho_0_list[k]
                    # Integrating through the class
                    A = project2(L_0, M_0, R_01, rho_01, T_01, X, Y, Y3, Y4, Z, Z7Li,
                        Z7Be)
                    r, M, L, rho, P, T, E, nabla_star, nabla_st, nabla_ad, PP1_E,
                        PP2_E, PP1_frac, PP2_frac, Fc_frac, Fr_frac = A.integrate(-1e26
                        , N, dynamic_step_size = True)
                    L_lim = L[-1]/L[0]
                    M_lim = M[-1]/M[0]
                    R_lim = r[-1]/r[0]
    #least_square(10, 0.85*R_sun, 1.2*R_sun, 50.*rho_0,80*rho_0, 0.85*5770, 1.2*5770,
        N, dm=-1e26)
```