University of Oslo

AST3310 - Astrophysical plasma and stellar interiors
Project 1

Modeling the solar core

Vedad Hodzic

March 10, 2014

Abstract

I here discuss the properties of the interior of the Sun based on models and simplifications addressed in [1] and [2]. Numerical calculations are used to solve the governing equations of the interior of the Sun. One should expect the luminosity, position and mass to end at zero value simultaneously, while the temperature lies around ~ 15 MK. It is found that the results heavily depend on initial physical properties, and in some cases on the mass step ∂m. Unexpectedly, it is likely the mass step has little to say given the right initial conditions.

1 Introduction

There are four differential equations that govern the internal structure of the Sun in our model. We apply here numerical methods such as the Euler integration scheme to solve these equations. This involves solving equations for nuclear fusion as well as ordinary differential equations. We look at how the inner Sun changes based on different initial parameters, and try to seek out a set of parameters that leads to the case where mass, position and luminosity reach zero value simultaneously.

2 Theory

The equations that regulate the physical properties of the interior of the Sun are expressed as

$$\frac{\partial r}{\partial m} = \frac{1}{4\pi r^2 \rho} \tag{1}$$

$$\frac{\partial P}{\partial m} = -\frac{Gm}{4\pi r^4} \tag{2}$$

$$\frac{\partial L}{\partial m} = \varepsilon \tag{3}$$

$$\frac{\partial T}{\partial m} = -\frac{3\kappa L}{256\pi^2 \sigma r^4 T^3}. \tag{4}$$

The variable ε in (3) is the total energy generation per unit mass. It is found by looking at the energy generation from nuclear reactions. It depends on the abundancy of different elements, temperature and density. The variable κ is the opacity, which is an average of frequency of photons. The pressure in eq. (2) is a sum of the gas pressure P_{G}, and the radiative pressure P_{R}.

$$\begin{aligned} P &= P_{\mathrm{G}} + P_{\mathrm{R}} \\ P &= \frac{\rho}{\mu m_{\mathrm{u}}} kT + \frac{a}{3} T^4 \\ \Rightarrow \rho &= \frac{\mu m_{\mathrm{u}}}{kT}\left(P - \frac{a}{3}T^4\right), \end{aligned} \tag{5}$$

which yields the density derived from the equation of state. Here, m_{u} is the atomic mass unit and k is the Boltzmann constant. The constant a is defined as $a = 4\sigma/c$, where σ is the

Stefan-Boltzmann constant, and c is the speed of light. The average molecular weight μ is found by

$$\mu = \frac{1}{\mu}\frac{\rho}{n_{\mathrm{tot}}}. \tag{6}$$

We can find the total particle density n_{tot} with

$$\begin{aligned} n_{\mathrm{tot}} &= n_X + n_Y + n_Z + n_e \\ &= \frac{X\rho}{m_{\mathrm{u}}} + \frac{Y\rho}{4m_{\mathrm{u}}} + \frac{Z\rho}{Am_{\mathrm{u}}} + \left(\frac{X\rho}{m_{\mathrm{u}}} + \frac{2Y\rho}{4m_{\mathrm{u}}} + \frac{Z\rho}{2m_{\mathrm{u}}}\right), \end{aligned}$$

where A is the average atomic weight of the heavier elements present, which we assume to be $A = 7$. This assumption is based on that we only know of two heavier elements present, which are ^{7}Be and ^{7}Li. We then get

$$\begin{aligned} \mu &= \frac{\rho m_{\mathrm{u}}}{\rho m_{\mathrm{u}}}\left(\frac{1}{2X + \frac{3}{4}Y + \frac{9}{14}Z}\right) \\ &= \frac{1}{2X + \frac{3}{4}Y + \frac{9}{14}Z}, \end{aligned}$$

assuming all elements are fully ionised.

The total energy generation per unit mass ε, is found by

$$\varepsilon = \sum Q'_{ik} r_{ik}, \tag{7}$$

where i, k represents two elements, Q'_{ik} is the energy released from the fusion of two elements, r_{ik} is the reaction rates per unit mass for two elements. The energies Q'_{ik} from the pp chains are listed in [1, p. 39, Table 2.1]. The reaction rates per unit mass is defined by

$$r_{ik} = \frac{n_i n_k}{\rho(1 + \delta_{ik})}\lambda_{ik}, \tag{8}$$

where n_i, n_k is the number density for an element, δ_{ik} is the Kronecker delta and λ_{ik} is the reaction rate of a fusion. The number density of an element is easily defined as

$$n = \frac{\rho \chi_a}{a m_{\mathrm{u}}}, \tag{9}$$

where χ is the number fraction of an element, and a is the atomic number of the element. We denote X, Y, Z to be the number fractions of hydrogen, helium and heavier metals, respectively. Finally, the reaction rates λ_{ik} for two elements i, k can be found in [1, p. 46, Table 2.3].

3 Algorithm

3.1 Simplifications

In order to make a simple model I needed some simplifications. A list of assumptions and simplifications follows.

	Unit system			
	CGS		SI	
Physical property	Unit name	Unit abbr.	Unit name	Unit abbr.
Length	centimetre	cm	metre	m
Weight	gram	g	kilogram	kg
Time	second	s	second	s
Temperature	kelvin	K	kelvin	K
Energy	erg	erg	joule	J
Pressure	barye	Ba	pascal	Pa

Table 1: Overview of the difference between CGS units and SI units.

- There is no change in the composition of elements as a function of radius.
- I assume there is no change in the density of deuterium, so that the rate of destruction of deuterium is the same as the production, and that any reaction involving deuterium happens instantaneous.
- All nuclear energy comes from the three PP-chains. I have not included the CNO cycle.
- I assume all elements to be fully inonised.

3.2 Units

Given that κ is in units of $[\mathrm{cm}^2\ \mathrm{g}^{-1}]$, λ in units of $[\mathrm{cm}^3\ \mathrm{s}^{-}1]$ and so on, I chose to adapt the CGS unit system (centimetre-gram-second). Table 1 shows an overview of the differences.

3.3 Structure

We see from eq. (4) that $\partial T/\partial m$ depends on the opacity, κ. A table of opacities that correspond to different values of temperature and density has been provided. I wrote a function that reads the file and stores values of T, R and κ in separate arrays. Here, $R = R(T, \rho) = \rho/T_6$. The function compares the table temperature with the actual temperature, and the same with the variable R. It returns the κ with table values of T and R that most closely resembles the present values.

Next, I wrote a function that calculates the energy generation per mass unit from nuclear reactions. This means entering the energy releases Q'_{ik} from [1, p. 39, Table 2.1], the reaction rates λ_{ik} from [1, p. 46, Table 2.3] and finding the number densities n for all particles that are involved in a nuclear reaction. Given the number fractions X, Y, Z and sub-fractions of these for different elements, I was able to calculate the energy generation per mass by using eq. (7), (8) and (9).

I have a function that calculates the density at present time, given the temperature T and total pressure P. This is found from the equation of state, as given in eq. (5). This required me to calculate the average molecular weight μ, shown in eq. (6).

Physical properties		Element abundancies	
Parameter	Initial value	Element	Initial value
L_0	$L_\odot$	X	0.7
R_0	$0.5R_\odot$	Y_3	10^{-10}
M_0	$0.7M_\odot$	Y	0.29
ρ_0	1 g cm^{-3}	Z	0.01
T_0	10^5 K	$Z_{^7\text{Li}}$	10^{-5}
P_0	10^{12} Ba	$Z_{^7\text{Be}}$	10^{-5}

Table 2: Table that shows the initial physical parameters needed in order to initiate the calculations.

Further, I have four functions that return the right-hand side in eq. (1), (2), (3) and (4). These are called upon in another function that integrates all the equations. I implemented a crude adaptive step system, where I changed the mass step gradually while integrating further into the Sun.

3.4 The Euler integration scheme

For simplicity, I have chosen the Euler integration scheme to integrate the differential equations. Listing (1) shows how it is carried out.

Listing 1: Euler integration loop

```
for i in range(n-1):
    r[i+1] = r[i] - drdm * dm
    P[i+1] = P[i] - dPdm * dm
    L[i+1] = L[i] - dLdm * dm
    T[i+1] = T[i] - dTdm * dm
    m[i+1] = m[i] - dm
```

The Euler method advances one step by adding the previous step value to the current value of the right-hand side of the differential equations, multiplied by the mass step. Unfortunately, the scheme carries a local truncation error proportional to the step size squared, and a global truncation error that is proportional to the step size.

3.5 Initial conditions

Finally, we need a set of initial conditions to initiate the calculations. These are shown in Table 2 expressed in the CGS unit system (see Table 1).

4 Results

4.1 Using original initial values

Carrying out the integrations using the initial conditions shown in Table 2, we get the results shown in Figure 1. As we see from these figures, the luminosity reaches zero before the position does, while we have integrated through just under 0.2 % of the initial mass. Figure 1d shows a sudden leap in temperature to start with. This indicates that the initial temperature is too low relative to the initial pressure and density. The equation of state tries to compensate for this, which leads to a violent increase in temperature. The physical parameter that changes abruptly is *probably* determined by the order the differential equations are solved. Had we solved them in another order, we would probably see a change.

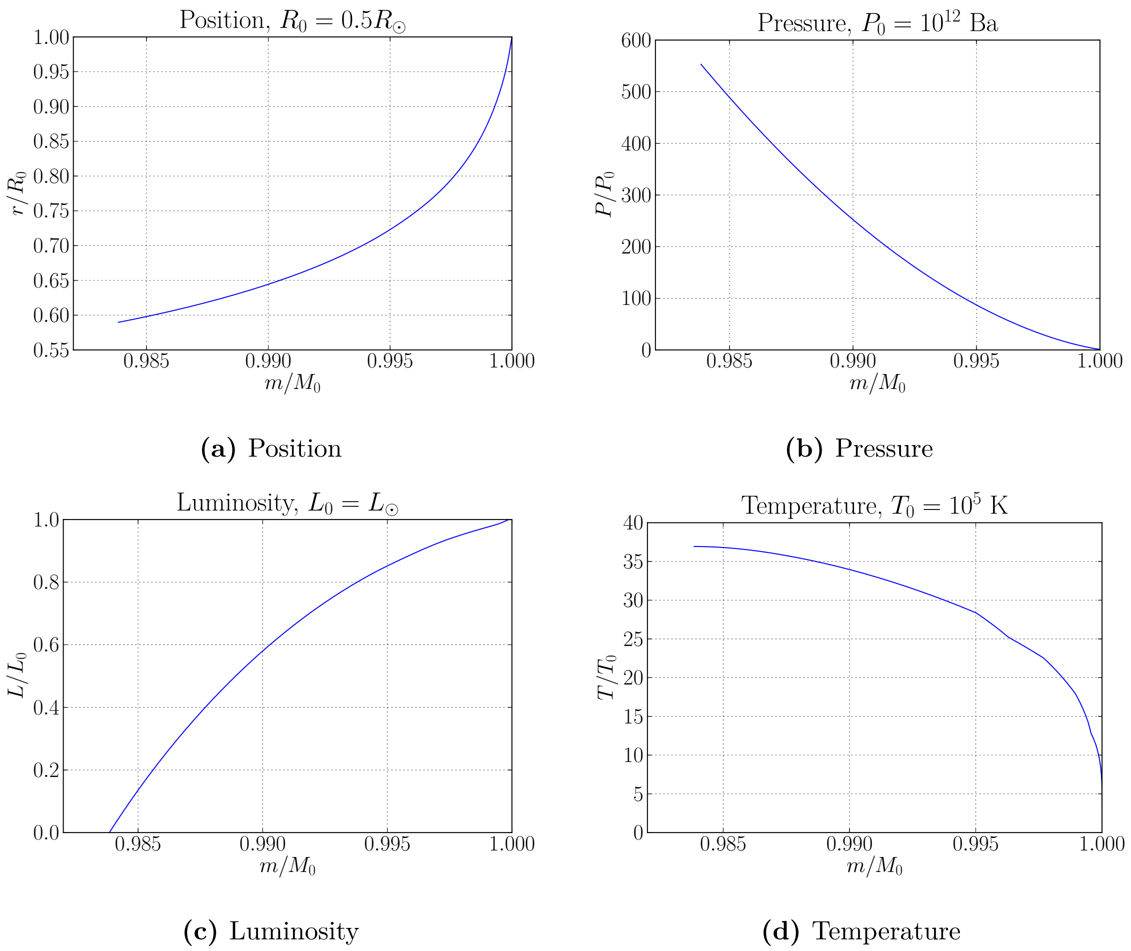


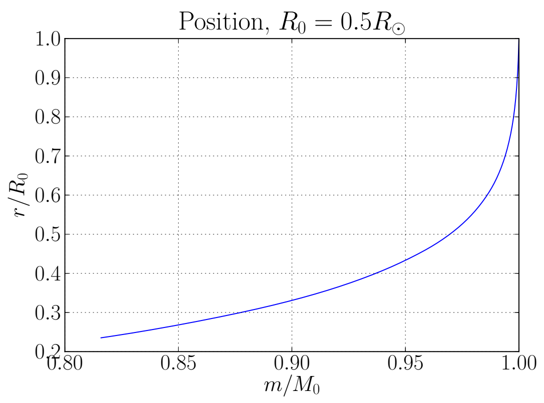
Figure 1: Showing results of solving the differential equations using the initial parameters as described in Table 2.

4.2 On the effect of changing ∂m

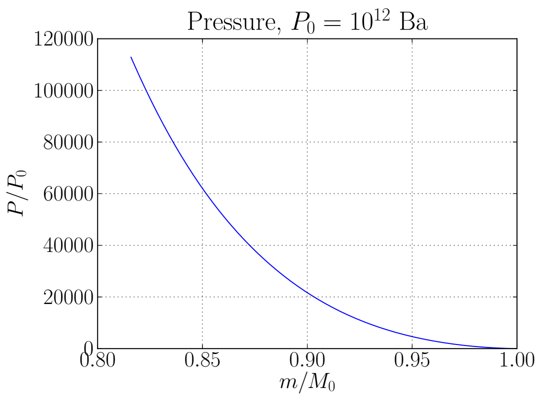
For some sets of initial conditions, it is important to choose a low ∂m to prevent the luminosity or temperature from increasing too quickly. However, this is highly time consuming. I chose to adapt a very small (of order 10^{-10} of the initial mass) step in the beginning, while gradually increasing it as we work our way into the core. I noticed however that given some initial conditions, I could choose ∂m to be as big as 1/100 of the original mass, and still get identical results as I did in e.g. Figure 3. I can only draw the conclusion that ∂m needs to be very small if we start with some physical parameters that are too low (or high) compared to others, as in the case with Figure 1d.

4.3 Decreasing the amount of ^{7}Li and ^{7}Be

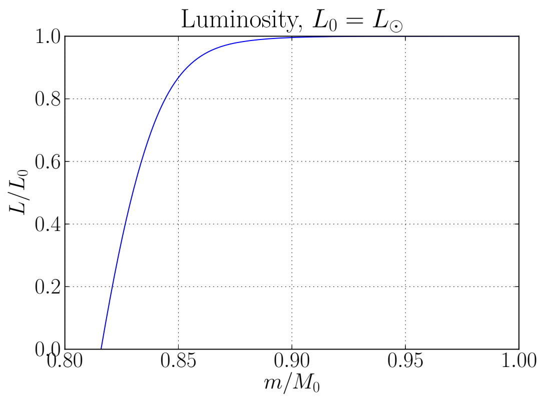
As a solution to the temperature problem, we try to lower the number fraction of ^{7}Li and ^{7}Be. We set $Z_{^7\mathrm{Li}} = Z_{^7\mathrm{Be}} = 10^{-13}$. This should lower the energy generation, thus the temperature through the luminosity. The results are shown in Figure 2. We already see a change, especially



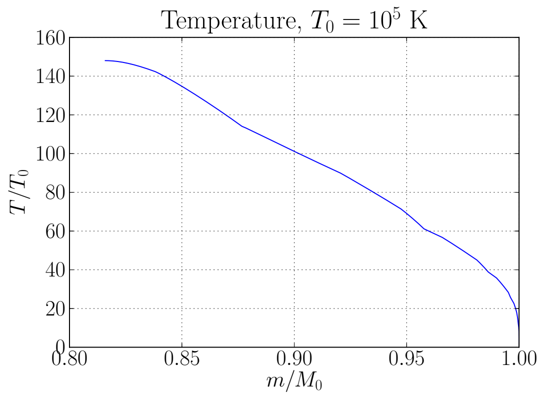
(a) Position



(b) Pressure



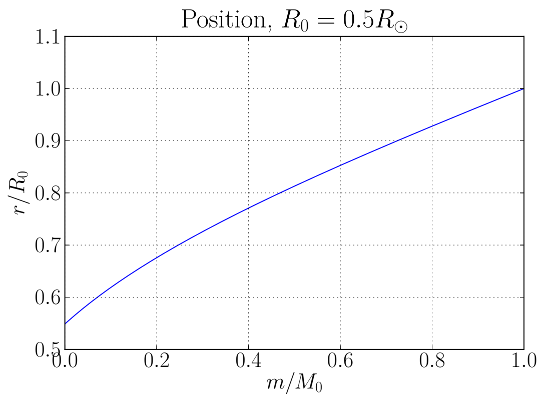
(c) Luminosity



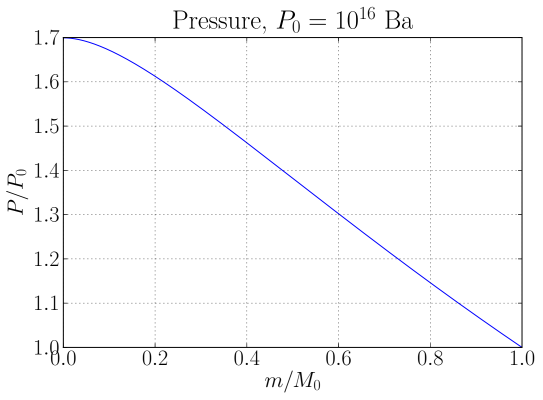
(d) Temperature

Figure 2: Integration results by using $Z_{^7\mathrm{Li}} = Z_{^7\mathrm{Be}} = 10^{-13}$.

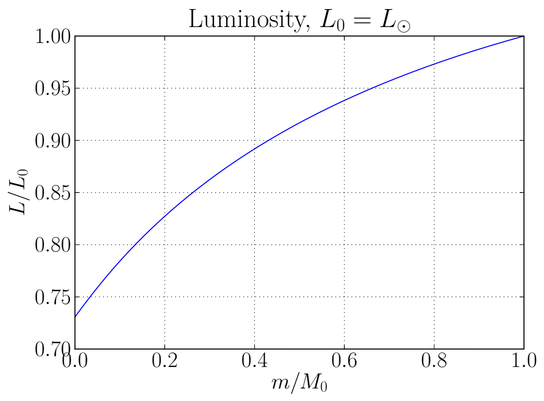
in the luminosity and temperature. Figure 2c now shows how the luminosity is close to constant to about 10 % of the initial mass. Then it suddenly dips exponentially, reaching zero value at about 82 % of M_0. Again, the luminosity decreases much faster than the position and mass. Figure 2a shows the position to be near 80 % into the core. The temperature in Figure 2d shows a steadier increase to about 15×10^6 K, which sounds reasonable as we approach the core. However, we see an abrupt increase in the pressure. It rises higher than a factor of 10^5 of the initial pressure. We increase the initial pressure to see if it stabilises our results further.



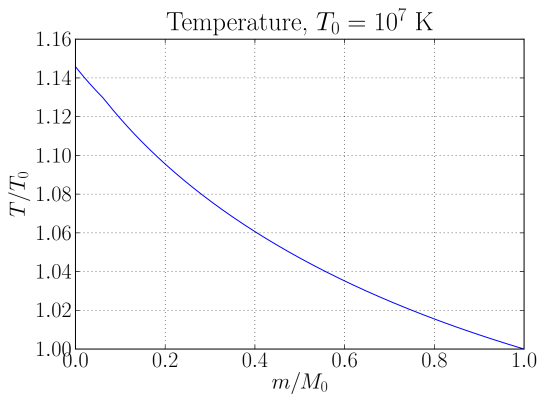
(a) Position



(b) Pressure



(c) Luminosity



(d) Temperature

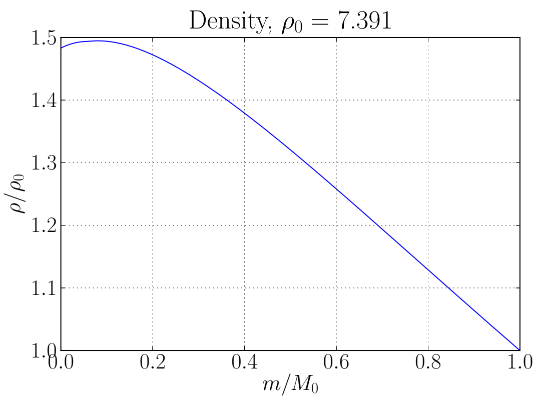
Figure 3: Integration results by increasing both the initial pressure and temperature.

4.4 Increasing initial temperature and pressure

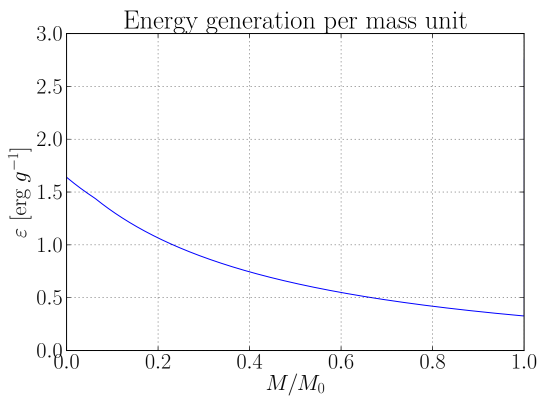
Increasing the pressure by a factor 10^4 of the original value, and the temperature by a factor of 10^2, yields Figure 3. We immediately notice that this time, it is the mass m that reaches zero. However, the luminosity and position stop at 73 % and 55 %, respectively. We also notice from Figure 3d that the temperature now increases faster towards the end, instead of in the beginning as we have had until now. This is probably more likely than the latter. This is the closest I have gotten to having the position, luminosity and mass reach zero values simultaneously. There might be a combination of ρ_0, P_0, T_0 that makes this possible, but I have not found it.

4.5 Density and energy generation

It could be interesting to see how the density and energy generation behave for a given set of initial values. We continue with the most recent initial values and have Figure 4. We see



(a) Density



(b) Energy generation per mass

Figure 4: Figures that show how the density and energy generation behave as function of mass, given the same initial conditions as in Figure 3.

that the density increases by a factor of 1.5. What is noticable however, is it how it decreases again towards the end, making it have the same value for different masses. This does not represent the physical situation, as the density should increase with increasing pressure and temperature without dropping again. We see the energy generation starts at a low value, but gradually increases as the temperature increases towards the core. This makes sense as it is strongly dependent on the temperature and density, where both increase towards the center.

5 The code

Listing 2: The program that solves the problem.

```python
import numpy as np
import sys
import time
import matplotlib.pyplot as plt

def opacity(T, rho):
    """
    Reads opacities from a file. Finds the opacity value
    that most closely resembles the present values for
    T and R.
    Returns the opacity in units of [cm^2 g^-1].
    """

    logT = []; logK = []
    inFile = open('opacity.txt', 'r')

    # Read the header file to store log(R)
    logR = np.asarray(inFile.readline().strip().split()[1:], dtype=np.float64)

    inFile.readline() # Skip the header

    # Adding log(T) and log(khappa) in separate lists
    for line in inFile:
        logT.append(line.strip().split()[0])
        logK.append(line.strip().split()[1:])

    inFile.close()
    # Converts the array to contain 64 bit floating point numbers
    logT = np.asarray(logT, dtype=np.float64)
    logK = np.asarray(logK, dtype=np.float64)

    R = rho / (T / 1e6) # Definition of R given in the opacity file

    # Make two arrays that contain the difference in T and R from present
        values
    diffT = abs(10**(logT) - T)
    diffR = abs(10**(logR) - R)

    # Finds the index of the minimum difference values, so the most relevant
        kappa can be used.
    i = np.argmin(diffT)
    j = np.argmin(diffR)

    kappa = 10**(logK[i,j])
    return kappa


def energyGeneration(T, rho):
    """
    Function to find the full energy generation per
    unit mass from the three PP chain reactions.
    """
```

```
ergs = 1.602e-6           # Conversion from MeV to ergs (CGS)
NA_inv = 1./6.022e23      # Avogadro's constant inverse

mu = 1.6605e-24      # Units of [g]      CGS

# Abundancy of different elements
X = 0.7              # Hydrogen (ionised)
Y = 0.29             # Helium 4 (ionised)
Y_3 = 1e-10          # Helium 3 (ionised)
Z = 0.01             # Heavier elements than the above (ionised)
Z_7Li = 1e-13        # Lithium 7 (part of Z)
Z_7Be = 1e-13        # Beryllium 7 (part of Z)

### Energy values (Q) ###
# Energy values of nuclear reactions in PP I [MeV]
Q_pp = (1.177 + 5.949) * ergs ; Q_3He3He = 12.86 * ergs

# Energy values of nuclear reactions in PP II [MeV]
Q_3He4He = 1.586 * ergs ; Q_e7Be = 0.049 * ergs ; Q_p7Li = 17.346 * ergs

# Energy values of nuclear reactions in PP III [MeV]
Q_p7Be= (0.137 + 8.367 + 2.995) * ergs

### Number densities (n) ###
n_p = X * rho / mu                    # Hydrogen
n_3He = Y_3 * rho / (3 * mu)          # Helium 3
n_4He = Y * rho / (4 * mu)            # Helium 4
n_7Be = Z_7Be * rho / (7 * mu)        # Beryllium 7
n_e = n_p + 2 * n_4He                 # Electron
n_7Li = Z_7Li * rho / (7 * mu)        # Lithium 7

### Reaction rates (lambda) ### Units of reactions per second per cubic cm
    . [cm^3 s^-1]
T9 = T / (1e9)

l_pp = (4.01e-15 * T9**(-2./3) * np.exp(-3.380 * T9**(-1./3)) * (1 + 0.123
     * T9**(1./3)\
        + 1.09 * T9**(2./3) + 0.938 * T9)) * NA_inv

l_3He3He = (6.04e10 * T9**(-2./3) * np.exp(-12.276 * T9**(-1./3))\
        * (1 + 0.034 * T9**(1./3) - 0.522 * T9**(2./3) - 0.124 * T9 +\
        0.353 * T9**(4./3) + 0.213 * T9**(5./3)))*NA_inv

a = 1 + 0.0495 * T9 # Defined for efficiency in l_3He4He
l_3He4He = (5.61e6 * a**(-5./6) * T9**(-2./3) * np.exp(-12.826 * a**(1./3)
     * T9**(-1./3)))\
        * NA_inv

# Check if temperature is below 1e6 for this reaction. Special case if
    true.
if T < 1e6:
    l_e7Be = (1.51e-7 / n_e) * NA_inv
else:
    l_e7Be = (1.34e-10 * T9**(-1./2) * (1 - 0.537 * T9**(1./3) + 3.86 * T9
        **(2./3)\
        + 0.0027 * T9**(-1.) * np.exp(2.515e-3 * T9**(-1.)))) * NA_inv
    a = 1 + 0.759 * T9 # Defined for efficiency in l_p7Li
    l_p7Li = (1.096e9 * T9**(-2./3) * np.exp(-8.472 * T9**(-1./3)) - 4.830e8 *
         a**(-5./6)\
    * T9**(-2./3) * np.exp(-8.472 * a**(1./3) * T9**(-1./3))) * NA_inv

    l_p7Be = (3.11e5 * T9**(-2./3) * np.exp(-10.262 * T9**(-1./3))) * NA_inv

    ### Rates per unit mass (r) ###
    r_pp = l_pp * (n_p * n_p) / (rho * 2)
    r_3He3He = l_3He3He * (n_3He * n_3He) / (rho * 2)
    r_3He4He = l_3He4He * (n_3He * n_4He) / rho
    r_e7Be = l_e7Be * (n_7Be * n_e) / rho
    r_p7Li = l_p7Li * (n_p * n_7Li) / rho
    r_p7Be = l_p7Be * (n_p * n_7Be) / rho

    ### Energy generation per unit mass from PP I, II and III ###
    epsilon = (Q_pp * r_pp) + (Q_3He3He * r_3He3He) + (Q_3He4He * r_3He4He) +
        (Q_e7Be *
            r_e7Be) + (Q_p7Li * r_p7Li) + (Q_p7Be * r_p7Be)

    return epsilon

def getRho(T, P):
    """
    Calculates the density at present location.
    Returns density in units of [g cm^-3].
    """
    ### Abundancy ###
    X = 0.7
    Y = 0.29
    Z = 0.01

    ### Constants ###

    # Stefan-Boltzmann constant
    sigma = 5.67e-5 # Units of [erg cm^-2 s^-1 K^-4]     CGS

    # Boltzmann constant
    k = 1.38e-16            # Units of [erg K^-1]        CGS

    # Atonmic mass unit
    m_u = 1.6605e-24        # Units of [g]               CGS

    # Speed of light
    c = 3e10                # Units of [cm s^-1]         CGS

    # Radiation constant
    a = 4 * sigma / c

    # Average molecular weight
    mu = 1./(2*X + 3./4 * Y + 9.*Z/14)

    rho = mu * m_u / (k * T) * (P - a/3 * T*T*T*T)
    return rho

def drdm(r, rho):
    """
    Calculates the right-hand side of dr/dm.
    """
    return 1./(4 * np.pi * r * r * rho)

def dPdm(r, m):
    """
    Calculates the right-hand side of dP/dm.
    """
    G = 6.67e-8      # Units of [cm^3 g^-1 s^-2]      CGS
    return - G * m / (4 * np.pi * r * r * r * r)

def dLdm(T, rho):
    """
    Calculates the right-hand side of dL/dm.
    """
    return energyGeneration(T, rho)

def dTdm(T, L, r, kappa):
    """
    Calculates the right-hand side of dT/dm.
    """
    sigma = 5.67e-5          # Units of [erg cm^-2 s^-1 K^-4]     CGS

    return -3 * kappa * L / (256 * np.pi*np.pi * sigma * r*r*r*r * T*T*T)


def integration():
    """
    Function that integrates the equation governing
    the internal structure of the radiative core
    of the Sun.
    """

    L0 = 3.839e33            # Units of [erg s^-1]        CGS
    R0 = 0.5 * 6.96e10       # Units of [cm]              CGS
    M0 = 0.7 * 1.99e33       # Units of [g]               CGS
    T0 = 1e5                 # Units of [K]               SI, CGS
    P0 = 1e12                # Units of [Ba]              CGS
    rho0 = getRho(T0, P0)    # Units of [g cm^-3]         CGS

    n = int(2e6)     # Length of integration arrays

    rho = np.zeros(n)    # Initialising density array
    rho[0] = rho0

    epsilon = np.zeros(n)    # Initialising energy generation array
    epsilon[0] = L0 / M0

    # Initialising integration arrays
    m = np.zeros(n)
    m[0] = M0

    r = np.zeros(n)
    r[0] = R0

    P = np.zeros(n)
    P[0] = P0

    L = np.zeros(n)
    L[0] = L0

    T = np.zeros(n)
    T[0] = T0

    start = time.time()
    for i in range(n-1):
        # Progress counter
        percent = (i / float(n-1)) * 100
        sys.stdout.write('Progress: %4.2f %s\r' % (percent, '%'))
        sys.stdout.flush()

        if m[i] < 0 or r[i] < 0 or L[i] < 0 or getRho(T[i], P[i]) < 0 or P[i]
           < 0 or T[i] < 0:
            end = i
            print "Something went below zero. Stopping."
            print "Step:", i
            print "Mass:", m[i]/M0
            print "Radius:", r[i]/R0
            print "Luminosity:", L[i]/L0
            print "Density:", rho[i]/rho0
            print "Pressure:", P[i]/P0
            print "Temperature:", T[i]/T0
            break
        else:
            # Gradually increasing dm
            if i < 200000:
                dm = 1.39e23
            elif i < 400000:
                dm = 1.39e24
            elif i < 600000:
                dm = 1.39e25
            elif i < 400000:
                dm = 1.39e26
            else:
                dm = 1.39e27

            r[i+1] = r[i] - drdm(r[i],rho[i]) * dm
            P[i+1] = P[i] - dPdm(r[i],m[i]) * dm
            L[i+1] = L[i] - dLdm(T[i], rho[i]) * dm
            T[i+1] = T[i] - dTdm(T[i], L[i], r[i], opacity(T[i], rho[i])) * dm
            rho[i+1] = getRho(T[i],P[i])
            epsilon[i+1] = energyGeneration(T[i],rho[i])
            m[i+1] = m[i] - dm

    # Writing elapsed time upon completion
    finish = time.time()
    print "Time elapsed:", ((finish-start)/60.), " min"

    return r, P, L, T, m, rho, rho0, P0, T0, L0, M0, R0, epsilon, end

def plots():
    """
    Function that plots the physical parameters.
```

```
"""

# Setting axis labels etc. in plots to the LaTeX font.
plt.rcParams['text.latex.preamble']=[r"\usepackage{lmodern}"]
params = {'text.usetex' : True,
          'font.size' : 22,
          'font.family' : 'lmodern',
          'text.latex.unicode': True,
          }
plt.rcParams.update(params)

# Calling the integration function
r, P, L, T, m, rho, rho0, P0, T0, L0, M0, R0, epsilon, end = integration()


# Plotting r(m)
fig_r = plt.figure()
ax_r = fig_r.add_subplot(111)

ax_r.set_title('Position, $R_0 = 0.5R_{\odot}$')
ax_r.set_xlabel('$m/M_0$')
ax_r.set_ylabel('$r/R_0$')
ax_r.plot(m[0:end]/M0,r[0:end]/R0)
ax_r.grid('on')
fig_r.tight_layout()

# Plotting P(m)
fig_P = plt.figure()
ax_P = fig_P.add_subplot(111)

ax_P.set_title('Pressure, $P_0 = 10^{12}$ Ba')
ax_P.set_xlabel('$m/M_0$')
ax_P.set_ylabel('$P/P_0$')
ax_P.plot(m[0:end]/M0,P[0:end]/P0)
ax_P.grid('on')
fig_P.tight_layout()

# Plotting L(m)
fig_L = plt.figure()
ax_L = fig_L.add_subplot(111)

ax_L.set_title('Luminosity, $L_0 = L_{\odot}$')
ax_L.set_xlabel('$m/M_0$')
ax_L.set_ylabel('$L/L_0$')
ax_L.plot(m[0:end]/M0,L[0:end]/L0)
ax_L.grid('on')
fig_L.tight_layout()

# Plotting T(m)
fig_T = plt.figure()
ax_T = fig_T.add_subplot(111)

ax_T.set_title('Temperature, $T_0 = 10^5$ K')
ax_T.set_xlabel('$m/M_0$')
ax_T.set_ylabel('$T/T_0$')
ax_T.plot(m[0:end]/M0,T[0:end]/T0)
ax_T.grid('on')
```

```
    fig_T.tight_layout()

    # Plotting rho(T, P)
    fig_p = plt.figure()
    ax_p = fig_p.add_subplot(111)

    ax_p.set_title('Density, $\\rho_0 =$ %.3f' % rho0)
    ax_p.set_xlabel('$m/M_0$')
    ax_p.set_ylabel('$\\rho/\\rho_0$')
    ax_p.plot(m[0:end]/M0,rho[0:end]/rho0)
    ax_p.grid('on')
    fig_p.tight_layout()

    # Plotting epsilon(T, rho)
    fig_e = plt.figure()
    ax_e = fig_e.add_subplot(111)

    ax_e.set_title('Energy generation per mass unit')
    ax_e.set_xlabel('$M/M_0$')
    ax_e.set_ylabel('$\\varepsilon$ [erg $g^{-1}$]')
    ax_e.plot(m[0:end]/M0,epsilon[0:end])
    ax_e.grid('on')
    fig_e.tight_layout()

    plt.show()

    pass

plots()
```

References

[1] Michael Stix, *The Sun*. Springer, New York, 2nd Edition, 2002.

[2] Boris Gudiksen *AST3310: Astrophysical plasma and stellar interiors*. 2014.