The Stellar Computer

Matthew Dennison & Mike Peel

Department of Physics and Astronomy
The University of Manchester

MPhys Project Report

January 2006

Abstract
A computer program has been constructed that calculates the physical properties within a star, utilizing four differential equations to describe the relations between the mass, radius, temperature, pressure and luminosity, as well as three material functions to provide the density, opacity and energy generated at concentric shells of the star. Numerical calculations are done using the 4^{th} order Runge-Kutta method with 5^{th} order error checking, and are initiated from both the surface and core of the star. Results are given for stars with masses of $0.5M_{\odot}$ and $1M_{\odot}$, along with recommendations for further improvements of this program.

1. Introduction

Computer-based modelling of stars has been performed since the 1960's, with the majority of work done in the period 1960 to 1970. The basic problem consists of four differential equations, with three additional material equations. These equations contain 8 variables – mass, radius, pressure, temperature, luminosity, density, opacity and energy generation – one of which is set to a known value, the other seven being unknown. The aim is to utilise boundary conditions to calculate the unknown variables at various positions within the star. As the equations cannot be algebraically solved (there are as many unknown variables as there are equations), they must be numerically calculated.

The simplifying assumptions that have been made in the construction of this model are that the star is perfectly spherical and symmetrical, the effects of rotation can be ignored and that the magnetic field is unimportant. These assumptions mean that the problem can be from three dimensions to just one. It has also been assumed that the star is in hydrostatic equilibrium throughout, which is justifiable due to the long lifetime of stars and means that time-dependent terms can be neglected.

The problem of computationally modelling a star can be broadly split into two topics – the theory and equations, and the numerical technique. The former will be discussed in section 2; the latter in section 4. The boundary conditions are discussed in section 3. Results are in section 5; conclusions in section 6 and suggestions for future directions are in section 7. The complete program code can be found in the appendix at the end of this report.

2. The Theory of Stellar Interiors

In what follows, two alternative versions for each differential equation have been provided. The first version is with respect to the radius r (known as the Eulerian description), the second with respect to the mass m (the Lagrangian description). Stellar models generally use the second set of equations, as stars are liable to change their radius but their mass will remain approximately constant over their lifetime (excluding events such as supernovae). Whilst the program only uses the Lagrangian description, it is felt that also providing the Eulerian description aids comprehension.

The variables in the equations below should be evaluated at the specific point being considered, unless otherwise stated. Complete derivations of the equations can be found in any stellar evolution text, for example Hansen et. al. (2004), unless otherwise stated.

2.1 Mass and Radius

The relation between the mass m and the radius r in a star is exactly the same as for any sphere. In the form of a differential equation, it is

$$\frac{dm}{dr} = 4\pi r^2 \rho, \tag{1a}$$

or alternatively

$$\frac{dr}{dm} = \frac{1}{4\pi r^2 \rho}, \tag{1b}$$

where ρ is the density.

2.2 Equation of Hydrostatic Equilibrium

As a star evolves very slowly it can be approximated to be in hydrostatic and thermodynamic equilibrium. The internal pressure P supports the star against the gravitational force generated by the star's own mass; this balance is shown by the equation of HydroStatic Equilibrium (HSE).

$$\frac{dP}{dr} = -\frac{GM_r\rho}{r^2}, \tag{2a}$$

or alternatively

$$\frac{dP}{dm} = -\frac{GM_r}{4\pi r^4}, \tag{2b}$$

where G is the gravitational constant and M_r denotes the total mass within the radius r.

2.3 Luminosity

The change in the luminosity L at any shell within the star is equal to the energy ε generated within that shell, i.e.

$$\frac{dL}{dr} = 4\pi r^2\varepsilon, \tag{3a}$$

or alternatively,

$$\frac{dL}{dm} = \varepsilon. \tag{3b}$$

2.4 The Transport Equation

The power flow in the star is based on the temperature T, and depends on the main transport method present at that point. It can be written as

$$\frac{dT}{dr} = -\frac{GM_r\rho}{r^2}\frac{T}{P}\nabla, \tag{4a}$$

or alternatively,

$$\frac{dT}{dm} = -\frac{GM_r}{4\pi r^4}\frac{T}{P}\nabla, \tag{4b}$$

where the gradient ∇ is determined by the transport mechanism used; either radiation, conduction or convection. An approximation can be made such that either radiation or convection is considered exclusively, rather than a combination of both, by setting

$$\nabla = \begin{cases} \nabla_{rad} & \nabla_{rad} \le \nabla_{ad} \\ \nabla_{conv} & \nabla_{rad} > \nabla_{ad} \end{cases}, \tag{5}$$

in which ∇_{rad} is the radiative gradient (which includes conduction), ∇_{ad} is the adiabatic gradient and ∇_{conv} is the convective gradient. The adiabatic gradient marks the point at which the changeover between convection and radiation occurs.

The radiative gradient is given by

$$\nabla_{rad} = \frac{3}{16\pi acG}\frac{\kappa LP}{M_r T^4}, \tag{6}$$

where κ is the opacity, a is the radiation constant and c is the speed of light. The adiabatic gradient is given by

$$\nabla_{ad} = \frac{d\ln T}{d\ln P}. \tag{7}$$

Convection is more problematic, for no complete theory of convection exists. Due to time constraints, rather than implementing Mixing Length Theory the approximation

$$\nabla_{conv} = \nabla_{ad} \tag{8}$$

has been made.

2.5 Density

One of the main topics of stellar modelling is the Equation of State, which is used here to calculate the density. This calculates the pressure from the physical conditions within the star, i.e. via the Ideal Gas law, radiation pressure, degeneracy pressure etc. Only the first two of these are accounted for in this program as the other effects are negligible in the case of a main sequence star such as this program is intended to model.

The Equation of State used is

$$P = P_{gas} + P_{radiation} = \frac{\rho}{\mu m_H} kT + \frac{a}{3} T^4, \tag{9a}$$

where the first part is the Ideal Gas law, and the second part is the standard equation for radiation pressure, in which a is the radiation constant, k is the Boltzmann constant and μ is the mean atomic mass. It can be rearranged to be

$$\rho = \left(P - \frac{a}{3} T^4 \right) \frac{\mu m_H}{kT}, \tag{9b}$$

from which the density can be found at any point where the pressure and temperature is known. The mean molecular mass can be calculated using

$$\frac{1}{\mu} = \sum_i \frac{X_i}{A_i}, \tag{10a}$$

where X_i is the fraction of element i within the star, and A_i is the number of nucleons within element i. It can be approximated as

$$\frac{1}{\mu} \approx X + \frac{Y}{4} + \frac{Z}{A_z}, \tag{10b}$$

where X is the fraction of hydrogen within the star, Y the fraction of helium, Z the fraction of 'metals' (i.e. all elements higher than helium) and A_z the average number of nucleons within the metals. For the purpose of this model, it is assumed that the mass fractions are constant throughout the star.

2.6 Energy Generation

This model will only be dealing with low-mass ($2M_\odot$ or lower) main sequence stars. Hence the majority of the energy generated will come from the three proton-proton reactions, as depicted in Figure 1.

The equation used calculates the total energy provided by nuclear reactions happening in a region of specified density, temperature and chemical composition in equilibrium. For the derivations behind this equation, and sub-equations, see Clayton (1968).

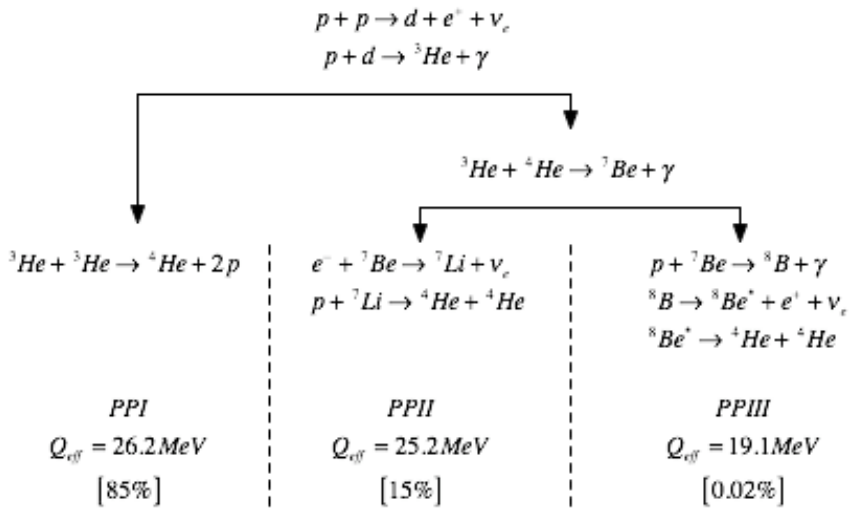


Figure 1: Reactions within the three Proton-Proton chains, with the net result $4p \rightarrow {}^4He + Q_{eff}$. Approximate probability for each decay reaction within the sun is given in brackets. From Phillips (1999) with corrections.

The energy generated through the combined PP chains operating in equilibrium can be calculated using

$$\begin{aligned}\varepsilon = {} & 2.36\times10^{2}\,\rho X^{2} T_6^{-2/3} e^{-33.81 T_6^{-1/3}} \psi_e \\ & \times\left(1 + 0.0123 T_6^{1/3} + 0.0109 T_6^{2/3} + 0.00095 T_6\right) J\,kg^{-1}\,s^{-1}\end{aligned} \tag{11}$$

where T_6 is the temperature scaled by a factor of 10^6. The combination of the second line of equation 11 and ψ_e provide the appropriate corrections to the energy generated depending on the ratio of the three PP chains – PPI, PPII or PPIII – that is occurring at that temperature.

ψ_e is given by

$$\psi_e = \left(1 - \alpha + \alpha\left(1 + \frac{2}{\alpha}\right)^{1/2}\right)\left(0.981 F_{PPI} + 0.961 F_{PPII} + 0.727 F_{PPIII}\right), \tag{12}$$

where α represents the number of alpha particles available, through

$$\alpha\left(T, \frac{Y}{X}\right) = 1.2\times10^{17}\left(\frac{Y}{X}\right)^{2} e^{-100 T_6^{-1/3}}, \tag{12}$$

and F_{PPI}, F_{PPII} and F_{PPIII} represent the fraction of the total number of reactions going through each PP chain; these are calculated using the following equations:

$$F_{PPI} = \left[\left(1 + \frac{2}{\alpha}\right)^{1/2} - 1\right]\left[\left(1 + \frac{2}{\alpha}\right)^{1/2} + 3\right]^{-1}, \tag{13}$$

$$F_{PPII} = \left(1 - F_{PPI}\right)\frac{\tau_p\left(Be^7\right)}{\tau_p\left(Be^7\right) + \tau_e\left(Be^7\right)} = \left(1 - F_{PPI}\right)\frac{1}{1 + \dfrac{\tau_e\left(Be^7\right)}{\tau_p\left(Be^7\right)}}. \tag{14a}$$

τ_p represents the average lifetime of the ^7Be atom before it absorbs a proton, which means the decay proceeds through PPII, and is given by

$$\tau_p = \frac{1}{6.3\times10^{-17} n_p T_6^{-\frac{2}{3}} \exp\left(-102.65 T_6^{-1/3}\right)} \text{ sec}. \quad (15)$$

τ_e represents the average lifetime of the ^7Be before it absorbs an electron, which means the decay proceeds through PPIII, and is given by:

$$\tau_e = \frac{1}{7.05\times10^{-33} n_e T_6^{-\frac{1}{2}}} \text{ sec}, \quad (16)$$

Assuming that the number densities of protons and electrons are equal,

$$F_{PPII} = \left(1 - F_{PPI}\right) \frac{1}{1 + \frac{6.3}{7.05}\times10^{16}\times T_6^{-1/6}\times\exp\left(-102.65 T_6^{-1/3}\right)}. \quad (14b)$$

Finally,

$$F_{PPIII} = 1 - F_{PPI} - F_{PPII}, \quad (17)$$

which comes from the fact that the three fractions must add up to 1.

2.7 Opacity

In general, opacity is complicated to calculate as it depends on a myriad of atomic effects such as electron degeneracy, quantum diffraction, collective plasma effects etc. Stellar models can either use approximate equations that fit the opacity curve, or tables such as those provided by the OPAL code. This program has both methods available, however only the OPAL data is used for the results given here.

Approximations to the opacity can be described using three equations, depending on the temperature. Up to $T \sim 10^{4.6}\ K$, the equation

$$\kappa = \kappa_3 \rho^{1/2} T^4. \quad (18)$$

can be used, where $\kappa_3 = 1.85\times10^{-14}\ m^3\ kg^{-1}$. From $T \sim 10^{4.6} K$ to $T \sim 10^{6.1}\ K$, it becomes

$$\kappa = \kappa_2 \rho T^{-3.5}. \quad (19)$$

where $\kappa_2 = 8\times10^{21}\ m^3\ kg^{-1}$. And finally, above $T \sim 10^{6.1} K$ it becomes

$$\kappa = 0.02\left(1 + X\right). \quad (20)$$

to represent the opacity plateau. See Taylor (1981) for the origin of these equations.

The OPAL tables, calculated by code written by Rogers & Iglesias (1995), provide more accurate values for the opacity at a specific temperature and density. They are provided as $\log_{10} T$ vs. $\log_{10} R$ tables, where

$$R = \frac{\rho}{T_6^3}\ g\,cm^{-3}\,K^{-3}. \quad (21)$$

with the opacity given as $\log_{10}\kappa$ where κ is in units of $cm^2\,g^{-1}$. While a thorough approach would involve interpolation of the values to the exact T and R at that point, due to time constraints this program only uses the nearest values of T and R natively available in the tables.

3. Boundary Conditions

There are two regions where boundary conditions are present: at the center and surface of the star.

3.1 Central Conditions

At the center of the star, $r = 0$, $m = 0$ and $L = 0$. Pressure will have some central value P_c, likewise temperature will be T_c; from these, central values for the density ρ_c, energy generation ε_c and opacity κ_c can be calculated.

At the first shell out from the center, the normal differential equations cannot be used as the variables with zero value in the center would cause either infinities or singularities in the equations. Hence approximations must be used; the following approximations from Kippenhahn & Weigert (1967) were used in this program:

$$r = \left(\frac{3}{4\pi\rho_c} \delta m \right)^{1/3}, \tag{22}$$

$$L = \varepsilon_c \, \delta m, \tag{23}$$

$$P = P_c - \frac{1}{2}\left(\frac{4\pi}{3} \right)^{1/3} G \rho_c^{4/3} \delta m^{2/3}, \tag{24}$$

The temperature approximation has been simplified to the following, which only considers radiative transport;

$$T = T_c - \frac{3\kappa L}{256\pi^2 \sigma r^4 T_c^{\,3}} \delta m, \tag{25}$$

where σ is the Stefan-Boltzmann constant. The density, energy generation and opacity are then calculated using the standard equations.

3.2 Surface Conditions

In order to solve the differential equations in the case of specific stars, values of the total radius R_*, the total mass M_*, the surface temperature T_* and the luminosity L_* are required, in addition to the chemical composition of the star. All of these are measurable or easily calculable.

While it would be possible to assume that the pressure at the surface of the star is zero, this will generally not be the case as the star does not have a fixed edge. Instead, the model of a photosphere given by Kippenhahn & Weigert (1990) can be used,

$$P = \frac{g}{\kappa} = \left(\frac{GMk^{1/2}}{R^2 \kappa_3 T^{3.5} \left(m_h \mu \right)^{1/2}} \right)^{2/3}, \tag{26}$$

where $g = \dfrac{GM}{r^2}$, κ is given by equation (18) and ρ has been substituted as $\rho = \dfrac{P m_h \mu}{Tk}$.

4. Program Structure & Numerical Methods

The numerical techniques described in this section can be found in more detail in Numerical Recipes in C (1992).

4.1 Euler's Method

The approach used to solving the differential equations is to split the star up into a series of shells, and turn the differential equations into difference equations. For example, take the differential equation

$$\frac{dy}{dx} = f(y,x). \tag{27}$$

The differential part of this can be represented as

$$\frac{dy}{dx} \xrightarrow[\delta x \to 0]{\lim} \frac{y_{n+1} - y_n}{\delta x} . \tag{28}$$

Hence,

$$y_{n+1} = y_n + f(y_n, x_n)\delta x . \tag{29}$$

This is known as Euler's method.

4.2 Fourth-Order Runge-Kutta Method

The Euler method has the drawback that the errors it introduces are fairly large – of the second order of δx. An improved method is the fourth-order Runge-Kutta method, which reduces the error down to the fifth order of δx. It does this by evaluating the equation at the initial point, at two trial points and at a trial end point.

Using the above example again, the fourth-order Runge-Kutta method can be expressed as

$$\begin{aligned} k_1 &= f(y_n, x_n)\,\delta x \\ k_2 &= f\left(y_n + \frac{k_1}{2}, x_n + \frac{\delta x}{2}\right)\delta x \\ k_3 &= f\left(y_n + \frac{k_2}{2}, x_n + \frac{\delta x}{2}\right)\delta x \\ k_4 &= f(y_n + k_3, x_n + \delta x)\,\delta x \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} \quad . \end{aligned} \tag{30}$$

4.3 Fifth-Order Error Checking & Adaptive Step-Size

The fourth-order Runga-Kutta method can be expanded on to provide an estimate for the error in the method. The version used requires six Runga-Kutta steps, from which both a fourth and fifth order solution can be found. The difference between the two values gives the estimate for the error.

The six steps taken are as follows

$$\begin{aligned}
k_1 &= f\left(x_n, y_n\right)\,\delta x \\
k_2 &= f\left(x_n + a_2 h, y_n + b_{21}k_1\right)\,\delta x \\
k_3 &= f\left(x_n + a_3 h, y_n + b_{31}k_1 + b_{32}k_2\right)\,\delta x \\
k_4 &= f\left(x_n + a_4 h, y_n + b_{41}k_1 + b_{42}k_2 + b_{43}k_3\right)\,\delta x \\
k_5 &= f\left(x_n + a_5 h, y_n + b_{51}k_1 + b_{52}k_2 + b_{53}k_3 + b_{54}k_4\right)\,\delta x \\
k_6 &= f\left(x_n + a_6 h, y_n + b_{61}k_1 + b_{62}k_2 + b_{63}k_3 + b_{64}k_4 + b_{65}k_5\right)\,\delta x \quad .
\end{aligned} \tag{31}$$

The solution is

$$y_{n+1} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 \,. \tag{32}$$

This has an error of

$$\Delta = y_{n+1} - y^*_{n+1}\,, \tag{33}$$

where

$$y^*_{n+1} = y_n + c^*_1\, k_1 + c^*_2\, k_2 + c^*_3\, k_3 + c^*_4\, k_4 + c^*_5\, k_5 + c^*_6\, k_6 \,. \tag{34}$$

a_i, b_{ij}, c_i and c^*_i are constants, the values of which are given in Table 1.

i	a_i	b_{i1}	b_{i2}	b_{i3}	b_{i4}	b_{i5}	c_i	c^*_i
1	0	0	0	0	0	0	$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$	0	0	$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$	0	0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$

Table 1: Cash-Karp Parameters for Embedded Runge-Kutta Method. From Press (1992).

While six steps are taken this is not a sixth order solution as some of the c_i and c^*_i values are zero, giving the fourth order solution (two of the c_i are zero) and fifth order comparison value (one c^*_i is zero).

Knowing the errors allows for a variable step-size δx to be used. If there is an error Δ in a particular step, but a smaller error of Δ^* is required, then the step could be repeated with a smaller δx. The new δx ($\delta x'$) can be found from

$$\delta x' = \delta x \left(\frac{\Delta^*}{\Delta}\right)^{1/5} . \tag{35}$$

where the power of 1/5 comes from the fact that the error found is a fifth order error. Using $\delta x'$ should give an error within the desired limit. As there are four variables with

errors - Pressure, Temperature, Radius and Luminosity – the largest fractional error is used to alter δx, as this will keep all values within the desired accuracy.

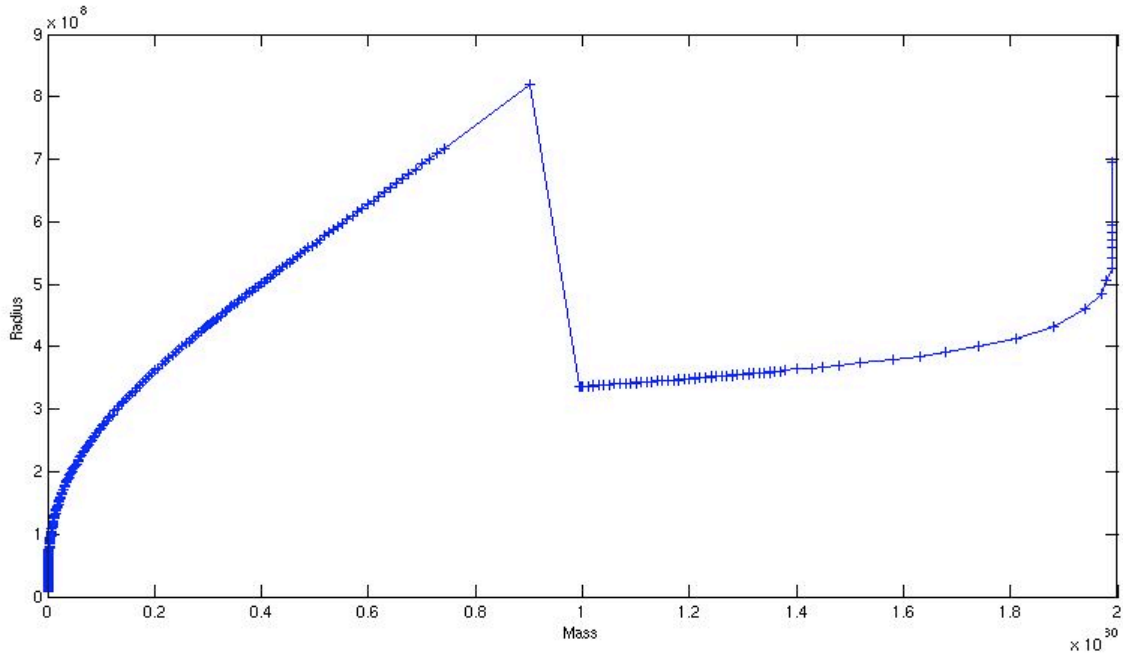
If, however, the error found is smaller than the desired error there is no need to repeat the step. Instead, equation 35 can be used to find a new, larger, δx for the next step, which should give an acceptable error. The process is then repeated, with a step being repeated each time the error is too large. Using this method should keep both errors (in the numerical method) and the number of steps to a minimum.

5. Results

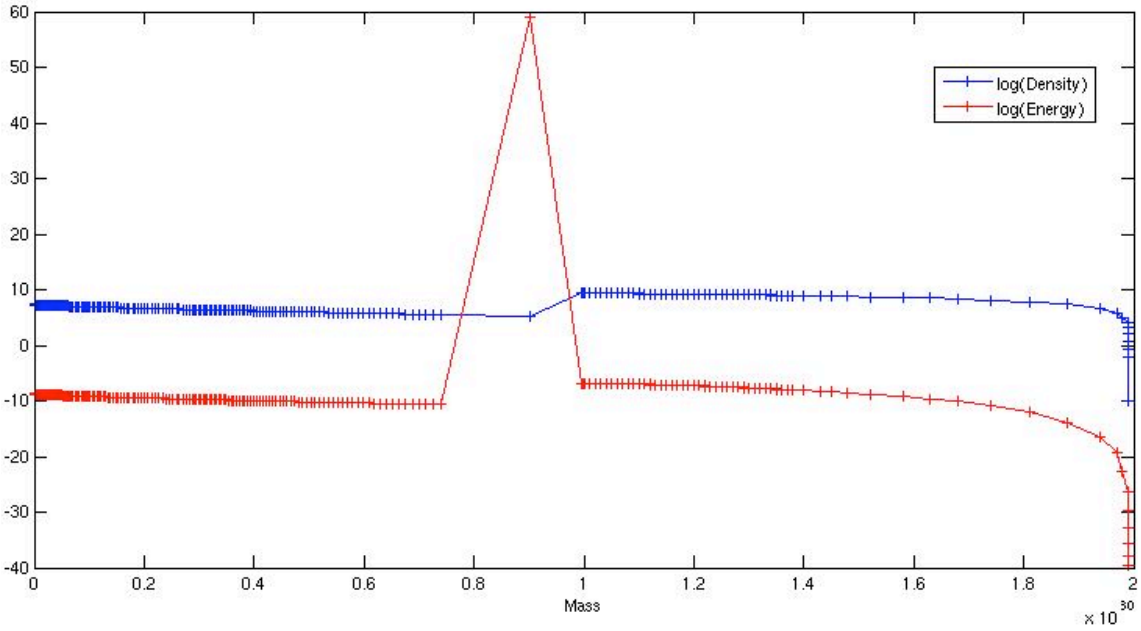
Results for two stars are presented in this section; the sun $\left(M = M_{\odot}\right)$ and a $0.5M_{\odot}$ star. Values for the variables of the latter star were found using figures 22.1 and 22.2 from Kippenhahn & Weigert (1990), and it has been assumed that it has the same chemical potential as the sun, namely $X = 0.71$, $Y = 0.27$ and $Z = 0.02$, with $A_z = 20$. For the data tables, see the appendix.

5.1 The Sun

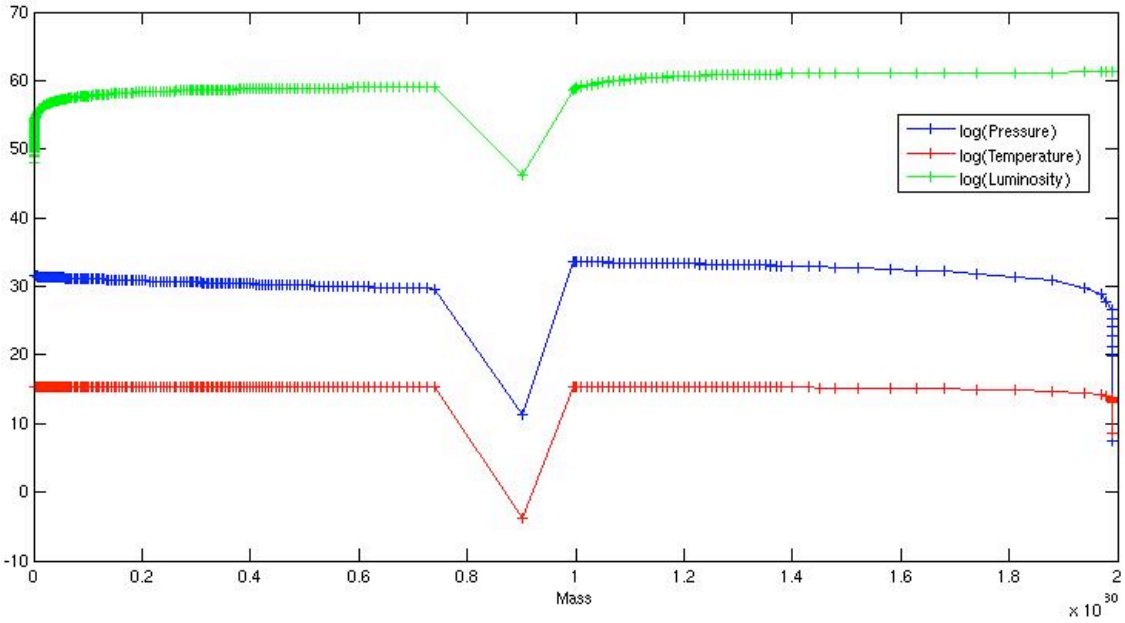
The sun has a mass of $M_{\odot} = 1.989 \times 10^{30}\, kg$, a radius of $R_{\odot} = 6.961 \times 10^{8}\, m$, a surface temperature of $T = 5800\, K$ and a total luminosity of $L_{\odot} = 3.846 \times 10^{26}\, W$.



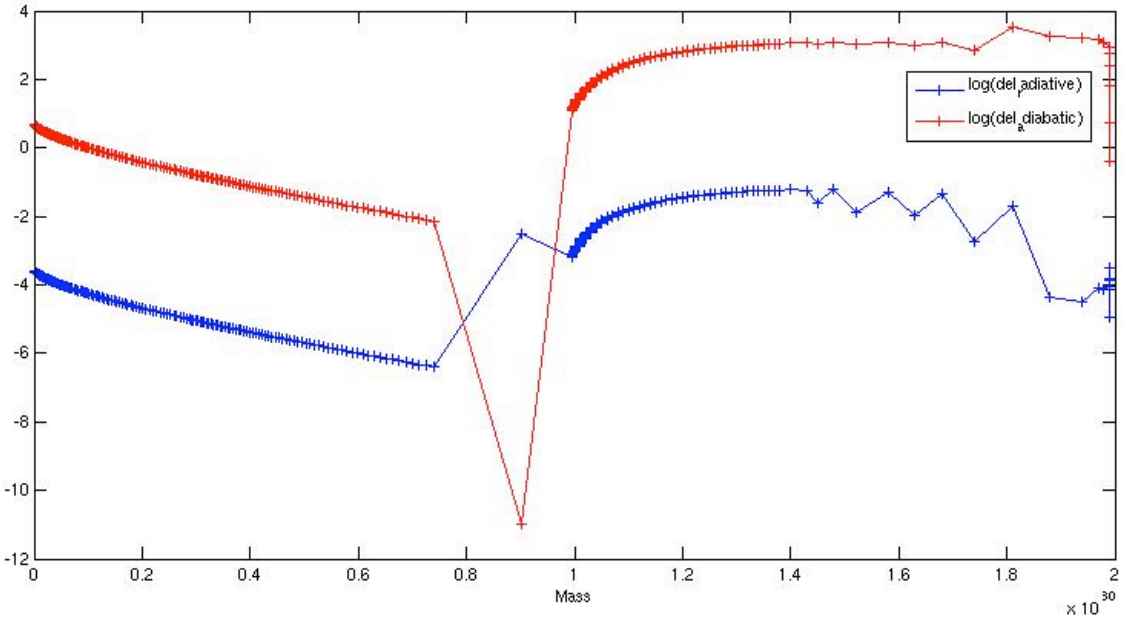
Graph 1: Radius vs. Mass for the Sun



Graph 2: Density and Energy Generation vs. Mass for the Sun



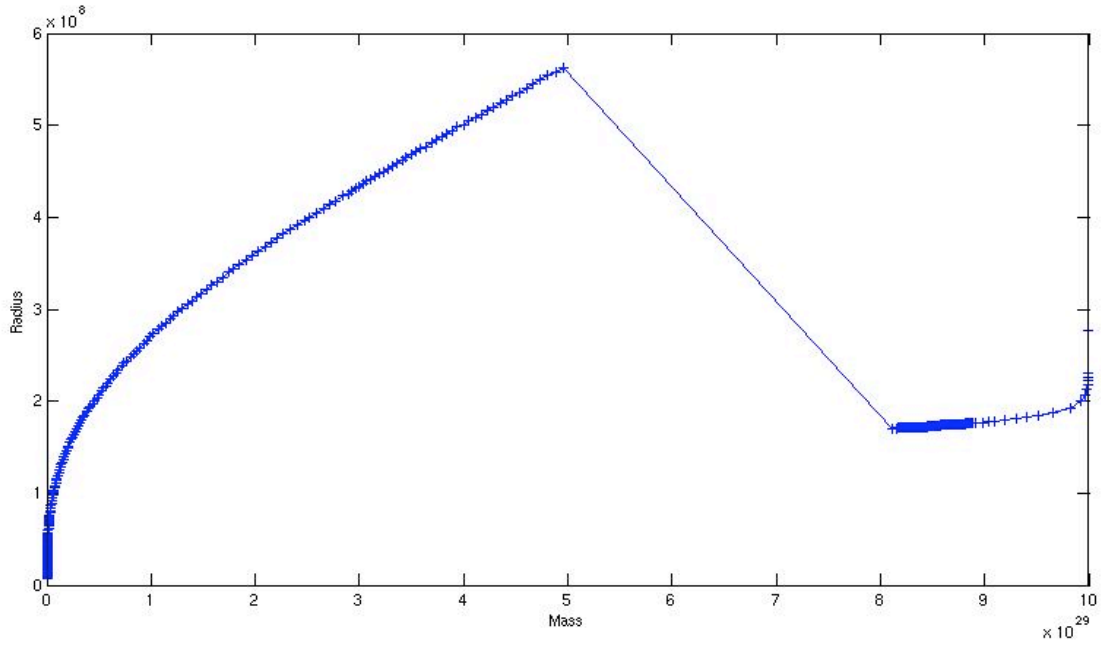
Graph 3: Pressure, Temperature and Luminosity vs. Mass for the Sun



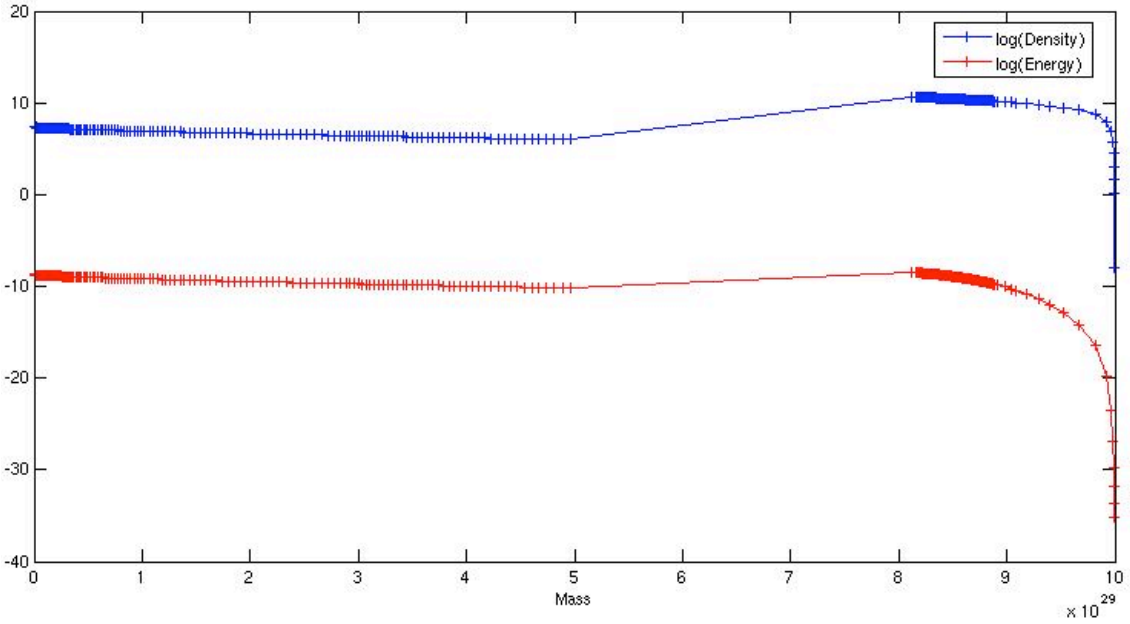
Graph 4: The Adiabatic and Radiative gradients for the Sun.

5.2 A $0.5\,M_\odot$ star

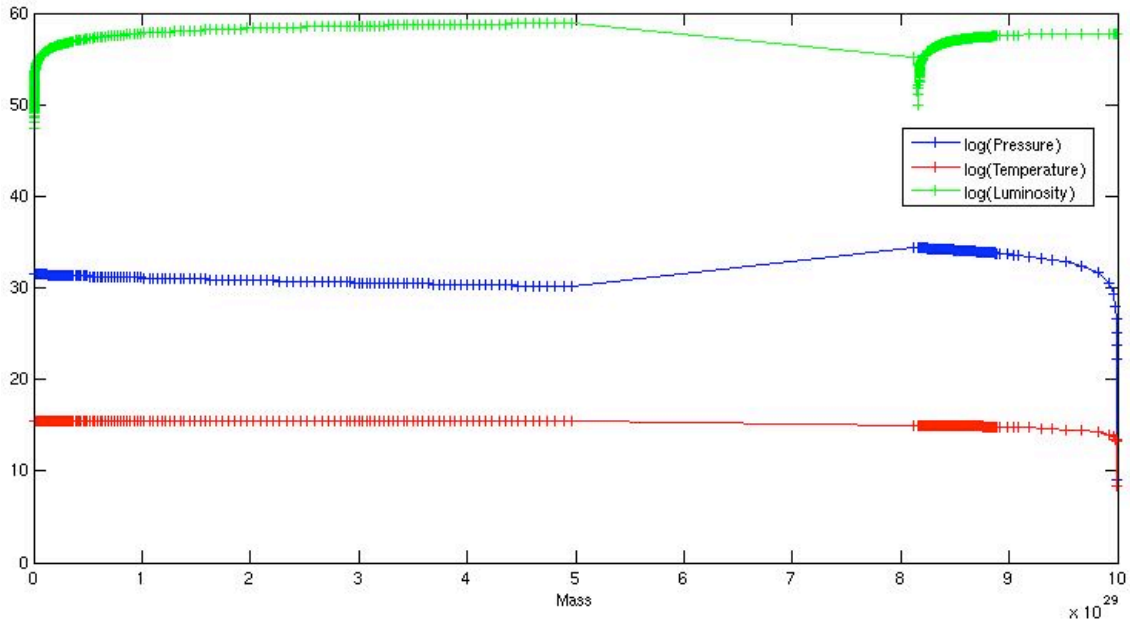
A half solar mass star $\left(M = 0.5M_\odot = 9.99\times10^{29}\,kg\right)$ will have a radius of $R \approx 0.398R_\odot \approx 2.770\times10^{8}\,m$, a surface temperature of $T_{eff} \approx 3980\,K$ and a surface luminosity of $L = 0.031L_\odot = 1.215\times10^{25}\,W$.



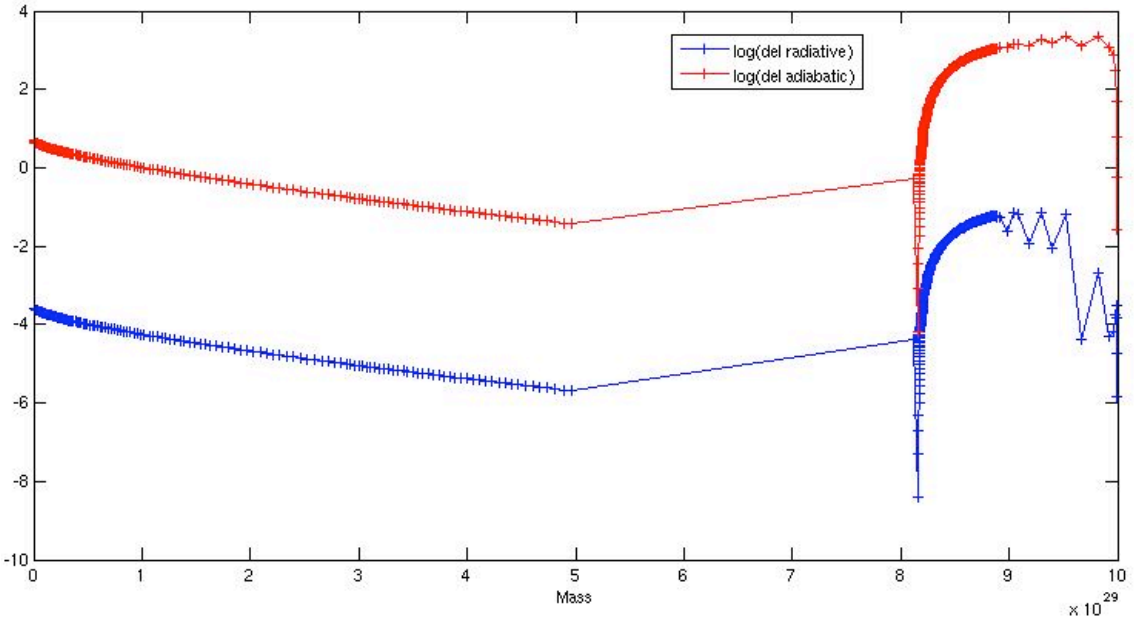
Graph 5: Radius vs. Mass for a $0.5M_\odot$ star



Graph 6: Density and Energy Generation vs. Mass for a $0.5M_{\odot}$ star



Graph 7: Pressure, Temperature and Luminosity vs. Mass for a $0.5M_{\odot}$ star



Graph 8: The Adiabatic and Radiative gradients for a $0.5M_{\odot}$ star

6. Conclusion

The aim of this project, to use a computer to numerically calculate the physical properties of a star, has only been partially met. Graphs 1 to 4 all show that for a $1M_{\odot}$ star, when moving outwards from the centre the dependent variables (i.e. radius, pressure, temperature and luminosity) tend to $\pm\infty$ before the mass variable reaches the mid-point, although it reaches the mid-point when moving inwards. Graphs 5 – 8 show that the reverse is true for a $0.5M_{\odot}$ star – while the program is fine moving outwards, infinities occur while moving inwards. The major problem appears to lie with the opacity. Without interpolating the OPAL data, the values used for the opacity at each specific density and temperature are not accurate enough. The same opacity has to be used for a wide range of densities and temperatures. When the value for the opacity does change, the temperature jumps, with a knock-on effect to the other variables. Once one variable has gone to $\pm\infty$, the rest immediately follow. Interpolating the opacity data could help prevent this.

Another fault can be clearly seen in graphs 1 and 5, and is also visible in the rest of the graphs. The values calculated for the radii of the different mass shells will not meet at the mid-point of the star. Graphs 2 to 4 and 6 to 8 appear to show that the other variables will be closer at the mid-point (if they continued in the same direction without going to $\pm\infty$), however this is deceptive as these are on log scales – this only shows that they will be within the correct order of magnitude (as will the radius). This problem most likely stems from the boundary conditions specified. The surface boundary conditions can be accurately found from observational data (especially in the case of the sun; less so for other stars), but the core conditions are harder to find as we cannot directly observe or measure them. The shooting method, where the program runs through a set of boundary conditions until the variables meet at the mid-point, could help overcome this problem.

These problems aside, the graphs give an idea of the conditions inside a star. Graphs 2 and 6 show that, moving away from the core, the density and energy generation fall, particularly near the surface. This would be expected since, as can also be seen from graphs 3 and 7, temperature and pressure also fall when moving outwards. The luminosity, shown on graphs 3 and 7, starts from nothing then rises very quickly when moving out from the centre due to the high energy generation. Pressure and temperature fall very quickly at the surface of the star (graphs 3 and 7), as would be expected due to the relatively low density at the surface of a star.

Overall, despite the problems mentioned, the program appears to be heading in the right direction. It should be feasible, given time, to produce a program that would accurately model the conditions in a star.

7. Future Directions

As this program has the potential to be very complex, it is recommended that future students doing this project are given the choice of starting from scratch, or using the version of the program described here (or that written by other students). This would enable them to look at more recent physics and computational techniques than is currently used in this program, which is mostly based on methods from the 1960's.

This program focuses on building a model of the star using fairly simple physics and more complicated numerical techniques. The most obvious and fundamental continuation of this program would be the introduction of time evolution of the star. There are a myriad of ways in which the physics could be extended, examples of which are:

- Energy generation from the CNO cycle,
- Layer-based chemical composition,
- Time evolution of the chemical composition,
- Time-dependent terms for the luminosity (i.e. storage / emission of energy),
- Improved treatment of convection,
- Inertial terms in calculating the pressure,
- Time-based variations of the mass and radius.

There are also ways in which the numerical method could be altered and improved, a few of which are:

- More elaborate adaptive stepsize algorithms,
- Implementation of the Shooting method, or
- Application of the Henyey (Relaxation) method.

More complicated extensions to the program would include such things as modelling the effects of magnetic fields, and rotation of the star, which would involve a full 3D model of the star. Models of binary systems could also be constructed. Additionally, solar winds and flares could be modelled using the base model constructed for the above effects.

In the long term, there are two logical conclusions for this project, both of which are closely related. The first is the modelling of the sun, such that the computer model holds as much data about the sun as possible and hence extrapolates what will happen in the near- to long-term future. The second is the thorough modelling of the development,

evolution and demise of other stars, where less information is known about them, or trying out models of theoretical stars to find out whether they are stable and/or how long they would survive.

References

Clayton, D.; "Principles of Stellar Evolution and Nucleosynthesis"; 1968.

Hansen, C., Kawaler, S & Trimble, V; Stellar Interiors, Physical Principles, Structure and Evolution, Second Edition; 2004; Springer.

Kippenhahn, R., Weigert, A.; Methods for Calculating Stellar Evolution; Methods in Computational Physics, Volume 7, 1967, pp 129-190; Academic Press.

Kippenhahn, R., Weigert, A.; "Stellar Structure and Evolution"; 1990 (3rd printing, 1994); p70.

Phillips, A. C.; "The Physics of Stars", Second Edition; 1999; Wiley; p119.

Press, W. H., et. al; Numerical Recipes in C: The Art of Scientific Computing; 1992; Cambridge University Press.

Rogers, F. R., Iglesias, C. A.; OPAL Opacity Tables; 1995
Described by "Updated OPAL Opacities"; Astrophys. J. No. 464, pp 943-953;1996.
Available online at http://www-phys.llnl.gov/Research/OPAL/

Taylor, R.J; The Stars: their structure and evolution; 1981.

Appendix: Program Code

The C programming language has been used for this program, as it is more familiar to the authors than the traditional FORTRAN and its variants.

All computer code, results tables and graphs are available online at:
http://www.mikepeel.net/physics/stellarcomputer/

Star.c

```
/*
PC4181 MPhys Project - Stellar Computer

Calculating the properties of a star given a set of input numbers
Started: 11/10/2005
Last Modified: 11/01/2006

Matthew Dennison (matthew.dennison@student.manchester.ac.uk) & Mike
    Peel (email@mikepeel.net)
*/

/*
Include the standard functions
*/
#include    <stdio.h>
#include    <math.h>
#include    <string.h>
#include    <stdlib.h>

/*
Set up the constants
N is the number of times we're doing the calculation
NUM_OPAL_I and NUM_OPAL_J are the number of rows and columns
    respectively (inc. logT and logR) of the opacity.txt file.
*/
#define N 100000
#define CONST_PI 3.14159
#define CONST_G  6.674e-11
#define CONST_K  1.381e-23
#define CONST_A  7.5657e-16
#define CONST_H  6.626e-34
#define CONST_C  299792458.0
#define MASS_H   1.673e-27
#define MASS_e   9.109e-31
#define NUM_OPAL_I 71
#define NUM_OPAL_J 20

/*
Set the input/output directory
*/
char directory[500] = "/star/";

/*
Set up the arrays
*/
long double radius[N], err_radius[N];
long double mass[N];
long double temperature[N], err_temperature[N];
long double pressure[N], err_pressure[N];
long double density[N];
long double luminosity[N], err_luminosity[N];
```

```
long double opacity[N];
long double energy[N];
long double dm[N], min_dm, new_dm;
long double degenerate_pressure[N], fermi_pressure;
long double del_adiabatic[N], del_radiative[N];
long double mu;
long double X, Y, Z, A_Z;
long double opal_data[71][20];

/*
Set up the variables used throughout the application
The temp_ variables are used in the Runge-Kutta routines.
The set number variables are the coefficients for the various elements
    within the RUk w/ 5th order error checking
*/
long double temp_density[6],
            temp_radius[6],
            temp_pressure[6],
            temp_opacity[6],
            temp_temperature[6],
            temp_energy[6],
            temp_luminosity[6];
long double *ptr_density, *ptr_opacity, *ptr_energy;
long double a2 = 0.2,
            a3 = 3.0/10.0,
            a4 = 3.0/5.0,
            a5 = 1.0,
            a6 = 7.0/8.0;
long double b21 = 0.2,
            b31 = 3.0/40.0,
            b32 = 9.0/40.0,
            b41 = 3.0/10.0,
            b42 = -9.0/10.0,
            b43 = 6.0/5.0,
            b51 = -11.0/54.0,
            b52 = 2.5,
            b53 = -70.0/27.0,
            b54 = 35.0/27.0,
            b61 = 1631.0/55296.0,
            b62 = 175.0/512.0,
            b63 = 575.0/13824.0,
            b64 = 44275.0/110592.0,
            b65 = 253.0/4096.0;
long double c1 = 37.0/378.0,
            c2 = 0.0,
            c3 = 250.0/621.0,
            c4 = 125.0/594.0,
            c5 = 0.0,
            c6 = 512.0/1771.0;
long double C1 = 2825.0/27648.0,
            C2 = 0.0,
            C3 = 18575.0/48384.0,
            C4 = 13525.0/55296.0,
            C5 = 277.0/14336.0,
            C6 = 0.25;
long double dP[6], dT[6], dR[6], dL[6], dM[6];
long double P1, T1, R1, L1;
// error is used to check the numerical error and change dm
long double error = 1e-8;
long double old_temperature, old_pressure;
int direction, loop, i, i_min, i_max, k, j;
```

```
// accuracy determines the number of decimal places the program
    outputs.
int accuracy = 12;
FILE *input_file, *output_file, *opacity_file;
char filename[100];

/*
Work out the radiative constant for later
*/
long double radiative_constant = (3.0 / (16.0 * CONST_PI * CONST_A *
    CONST_C * CONST_G));

/*
Get the functions
*/
#include    "functions.h"

/*
This is the main program
*/
int main()
{
    /*
    Get the input data from input.txt
    */
    sprintf(filename, "%sinput.txt", directory);
    input_file = fopen(filename, "r");
    if (!input_file)
    {
        printf("Failed to open %sinput.txt \n", directory);
        return 1;
    };
    fscanf(input_file, "Radius: %Lf\n", &radius[0]);
    fscanf(input_file, "Mass: %Lf\n", &mass[0]);
    fscanf(input_file, "Surface Temperature: %Lf\n", &temperature[0]);
    fscanf(input_file, "Luminosity: %Lf\n", &luminosity[0]);
    fscanf(input_file, "Core Temperature: %Lf\n", &temperature[N-1]);
    fscanf(input_file, "Core Pressure: %Lf\n", &pressure[N-1]);
    fscanf(input_file, "X: %Lf\n", &X);
    fscanf(input_file, "Y: %Lf\n", &Y);
    fscanf(input_file, "Z: %Lf\n", &Z);
    fscanf(input_file, "A_Z: %Lf\n", &A_Z);

    fclose(input_file);


    /*
    Get the OPAL data
    */
    sprintf(filename, "%sopacity.txt", directory);
    opacity_file = fopen(filename, "r");
    if (!opacity_file)
    {
        printf("Failed to open %sopacity.txt \n", directory);
        return 1;
    };
    // For each column in the csv file...
    for (i = 0; i < NUM_OPAL_I; i++)
    {
        // For each row in the csv file...
        for (j = 0; j < NUM_OPAL_J; j++)
```

```
        {
            fscanf(opacity_file, "%Lf", &opal_data[i][j]);
            /*
            For debugging...
            printf("%Lf", opal_data[i][j]);
            */
        };
        /*
        For debugging...
        printf("\n");
        */
};
fclose(opacity_file);


/*
Set a minimum mass to be 100x smaller than that defined by N -
    that's the minimum we should need to use for accuracy of
    around 1e-8.
*/
min_dm = mass[0] / (100 * N);

// Calculate mu
mu = calc_mu(X, Y, Z, A_Z);

/*
Set the remaining boundary conditions at the surface and the
    centre.
Pressure uses the surface pressure equation. See section 3.2 of
    the report for details. 1.85e-14 is kappa_3.
*/
dm[1] = min_dm;
pressure[0] = pow(((CONST_G * mass[0] * pow(CONST_K, 0.5)) /
    (pow(radius[0], 2.0) * 1.85e-14 * pow(temperature[0], 3.5) *
    pow(mu, 0.5))), 0.66);
density[0] = calc_density(X, temperature[0], pressure[0]);
opacity[0] = 1.85e-14 * pow(density[0], 0.5) * pow(temperature[0],
    4.0);

/*
Get the values for the core.
Temperature and pressure have been defined when input.txt is read.
*/
dm[N-1] = min_dm;
mass[N-1] = 0.0;
luminosity[N-1] = 0.0;
radius[N-1] = 0.0;
density[N-1] = calc_density(X, temperature[N-1], pressure[N-1]);
opacity[N-1] = calc_opacity(X, density[N-1], temperature[N-1]);
energy[N-1] = calc_energy_generation(temperature[N-1], density[N-
    1], X, Y);

/*
Get the values for the second innermost shell.
See section 3.1 of the report for details.
*/
dm[N-2] = min_dm;
mass[N-2] = dm[N-1];
radius[N-2] = pow(((3.0 * mass[N-2]) / (4.0 * CONST_PI *
    density[N-1])), 1.0 / 3.0);
luminosity[N-2] = energy[N-1] * mass[N-2];
```

```
pressure[N-2] = pressure[N-1] - 0.5 * pow((4.0 * CONST_PI / 3.0),
    (1.0 / 3.0)) * CONST_G * pow(density[N-1], (4.0 / 3.0)) *
    pow(mass[N-2], (2.0 / 3.0));
del_radiative[N-2] = radiative_constant * opacity[N-1] * energy[N-
    1] * pressure[N-1] * pow(temperature[N-1], -4.0);
temperature[N-2] = temperature[N-1] - (3.0 * opacity[N-1] *
    luminosity[N-2] * mass[N-2])/(256.0 * CONST_PI * CONST_PI *
    5.67e-8 * pow(radius[N-2],4.0) * pow(temperature[N-1], 3.0));
density[N-2] = calc_density(X, temperature[N-2], pressure[N-2]);
opacity[N-2] = calc_opacity(X, density[N-2], temperature[N-2]);
energy[N-2] = calc_energy_generation(temperature[N-2], density[N-
    2], X, Y);

dm[N-3] = min_dm;

for (loop = 0; loop < 2; loop++)
{
    if (loop == 1)
    {
        /*
        This is the loop starting at the core, and working
            outwards.
        Direction is negative as i is decreasing - i=N is the
            center.
        */
        direction = -1;
        i_min = N - 3;
        i_max = 0.5 * N;
    }
    else
    {
        /*
        This is the loop starting on the surface, and working
            inwards
        Direction is positive as i is increasing - i=1 is the
            surface.
        */
        direction = +1;
        i_min = 1;
        i_max = 0.5 * N;
    };
    for (i = i_min; (loop == 0) ? i < i_max : i > i_max; i +=
        direction)
    {

        /*
        Mass
        */
        mass[i] = mass[i - direction] - direction * dm[i];

        /*
        First Runge-Kutta loop
        */
        dP[0] = 0.0;
        dT[0] = 0.0;
        dR[0] = 0.0;
        dL[0] = 0.0;
        dM[0] = 0.0;
        ptr_density = &density[i - direction];
        ptr_energy = &energy[i - direction];
        ptr_opacity = &opacity[i - direction];
```

```
old_temperature = (temperature[i - direction * 2] +
    temperature[i - direction]) * b21;
old_pressure = (pressure[i - direction * 2] + pressure[i -
    direction]) * b21;
rungekutta(direction, i, 0);

/*
Second Runge-Kutta loop
*/
dP[1] = temp_pressure[0] * b21;
dT[1] = temp_temperature[0] * b21;
dR[1] = temp_radius[0] * b21;
dL[1] = temp_luminosity[0] * b21;
dM[1] = dm[i] * a2;
ptr_density = &temp_density[0];
ptr_energy = &temp_energy[0];
ptr_opacity = &temp_opacity[0];
old_temperature = ((temperature[i - direction * 2] +
    temperature[i - direction]) * b21);
old_pressure = ((pressure[i - direction * 2] + pressure[i
    - direction]) * b21);
rungekutta(direction, i, 1);

/*
Third Runge-Kutta loop
*/
dP[2] = (temp_pressure[0] * b31) + (temp_pressure[1] *
    b32);
dT[2] = (temp_temperature[0] * b31) + (temp_temperature[1]
    * b32);
dR[2] = (temp_radius[0] * b31) + (temp_radius[1] * b32);
dL[2] = (temp_luminosity[0] * b31) + (temp_luminosity[1] *
    b32);
dM[2] = dm[i] * a3;
ptr_density = &temp_density[1];
ptr_energy = &temp_energy[1];
ptr_opacity = &temp_opacity[1];
old_temperature = (temperature[i - direction * 2] +
    temperature[i - direction]) / 2;
old_pressure = (pressure[i - direction * 2] + pressure[i -
    direction]) / 2;
rungekutta(direction, i, 2);

/*
Fourth Runge-Kutta loop
*/
dP[3] = (temp_pressure[0] * b41) + (temp_pressure[1] *
    b42) + (temp_pressure[2] * b43);
dT[3] = (temp_temperature[0] * b41) + (temp_temperature[1]
    * b42) + (temp_temperature[2] * b43);
dR[3] = (temp_radius[0] * b41) + (temp_radius[1] * b42) +
    (temp_radius[2] * b43);
dL[3] = (temp_luminosity[0] * b41) + (temp_luminosity[1] *
    b42) + (temp_luminosity[2] * b43);
dM[3] = dm[i] * a4;
ptr_density = &temp_density[2];
ptr_energy = &temp_energy[2];
ptr_opacity = &temp_opacity[2];
old_temperature = temperature[i - direction];
old_pressure = pressure[i - direction];
rungekutta(direction, i, 3);
```

```
/*
Fifth Runge-Kutta loop
*/
dP[4] = (temp_pressure[0] * b51) + (temp_pressure[1] *
    b52) + (temp_pressure[2] * b53) + (temp_pressure[3] *
    b54);
dT[4] = (temp_temperature[0] * 51) + (temp_temperature[1]
    * b52) + (temp_temperature[2] * b53) +
    (temp_temperature[3] * b54);
dR[4] = (temp_radius[0] * b51) + (temp_radius[1] * b52) +
    (temp_radius[2] * b53) + (temp_radius[3] * b54);
dL[4] = (temp_luminosity[0] * b51) + (temp_luminosity[1] *
    b52) + (temp_luminosity[2] * b53) +
    (temp_luminosity[3] * b54);
dM[4] = dm[i] * a5;
ptr_density = &temp_density[3];
ptr_energy = &temp_energy[3];
ptr_opacity = &temp_opacity[3];
old_temperature = temperature[i - direction];
old_pressure = pressure[i - direction];
rungekutta(direction, i, 4);

/*
Sixth Runge-Kutta loop
*/
dP[5] = (temp_pressure[0] * b61) + (temp_pressure[1] *
    b62) + (temp_pressure[2] * b63) + (temp_pressure[3] *
    b64) + (temp_pressure[4] * b65);
dT[5] = (temp_temperature[0] * 61) + (temp_temperature[1]
    * b62) + (temp_temperature[2] * b63) +
    (temp_temperature[3] * b64) + (temp_temperature[4] *
    b65);
dR[5] = (temp_radius[0] * b61) + (temp_radius[1] * b62) +
    (temp_radius[2] * b63) + (temp_radius[3] * b64) +
    (temp_radius[4] * b65);
dL[5] = (temp_luminosity[0] * b61) + (temp_luminosity[1] *
    b62) + (temp_luminosity[2] * b63) +
    (temp_luminosity[3] * b64) + (temp_luminosity[4] *
    b65);
dM[5] = dm[i] * a6;
ptr_density = &temp_density[4];
ptr_energy = &temp_energy[4];
ptr_opacity = &temp_opacity[4];
old_temperature = temperature[i - direction];
old_pressure = pressure[i - direction];
rungekutta(direction, i, 5);

/*
Finally, compute the values.
*/
radius[i] = radius[i - direction] - direction *
    ((temp_radius[0] * c1) + (temp_radius[1] * c2) +
    (temp_radius[2] * c3) + (temp_radius[3] * c4) +
    (temp_radius[4] * c5) + (temp_radius[5] * c6));
pressure[i] = pressure[i - direction] + direction *
    ((temp_pressure[0] * c1) + (temp_pressure[1] * c2) +
    (temp_pressure[2] * c3) + (temp_pressure[3] * c4) +
    (temp_pressure[4] * c5) + (temp_pressure[5] * c6));
temperature[i] = temperature[i - direction] + direction *
    ((temp_temperature[0] * c1) + (temp_temperature[1] *
```

```
    c2) + (temp_temperature[2] * c3) +
    (temp_temperature[3] * c4) + (temp_temperature[4] *
    c5) + (temp_temperature[5] * c6));
luminosity[i] = luminosity[i - direction] - direction *
    ((temp_luminosity[0] * c1) + (temp_luminosity[1] * c2)
    + (temp_luminosity[2] * c3) + (temp_luminosity[3] *
    c4) + (temp_luminosity[4] * c5) + (temp_luminosity[5]
    * c6));

err_radius[i] = sqrt(pow(radius [i] - (radius[i -
    direction] - direction *((temp_radius[0] * C1) +
    (temp_radius[1] * C2) + (temp_radius[2] * C3) +
    (temp_radius[3] * C4) + (temp_radius[4] * C5) +
    (temp_radius[5] * C6))), 2.0));
err_pressure[i] = sqrt(pow(pressure[i] - (pressure[i -
    direction] + direction *((temp_pressure[0] * C1) +
    (temp_pressure[1] * C2) + (temp_pressure[2] * C3) +
    (temp_pressure[3] * C4) + (temp_pressure[4] * C5) +
    (temp_pressure[5] * C6))), 2.0));
err_temperature[i] = sqrt(pow(temperature[i] -
    (temperature[i - direction] + direction
    *((temp_temperature[0] * C1) + (temp_temperature[1] *
    C2) + (temp_temperature[2] * C3) +
    (temp_temperature[3] * C4) + (temp_temperature[4] *
    C5) + (temp_temperature[5] * C6))), 2.0));
err_luminosity[i] = sqrt(pow(luminosity[i] - (luminosity[i
    - direction] - direction *((temp_luminosity[0] * C1) +
    (temp_luminosity[1] * C2) + (temp_luminosity[2] * C3)
    + (temp_luminosity[3] * C4) + (temp_luminosity[4] *
    C5) + (temp_luminosity[5] * C6))), 2.0));

L1 = pow(((error * luminosity[i]) / err_luminosity[i]),
    0.2);
T1 = pow(((error * temperature[i]) / err_temperature[i]),
    0.2);
R1 = pow(((error * radius[i]) / err_radius[i]), 0.2);
P1 = pow(((error * pressure[i]) / err_pressure[i]), 0.2);

if (L1 < T1 && L1 < R1 && L1 < P1)
{
    new_dm = dm[i] * L1;
}
else if (T1 > R1 && T1 > P1)
{
    new_dm = dm[i] * T1;
}
else if (R1 > P1)
{
    new_dm = dm[i] * R1;
}
else
{
    new_dm = dm[i] * P1;
};

// Check and see which one is the greater - new_dm or
    min_dm - and use that one.
if (new_dm < min_dm)
{
    dm[i + direction] = min_dm;
}
```

```
        else
        {
            dm[i + direction] = new_dm;
        };

        density[i] = calc_density(X, temperature[i], pressure[i]);
        opacity[i] = calc_opacity(X, density[i], temperature[i]);
        energy[i] = calc_energy_generation(temperature[i],
            density[i], X, Y);

        /*
        Calculate the degeneracy pressure.
        */
        fermi_pressure = pow((3.0 * density[i]) / (8.0 * CONST_PI
            * X * MASS_H), 1.0/3.0) * CONST_H;
        if (fermi_pressure < MASS_e * 3.0e8)
        {
            degenerate_pressure[i] = (pow(CONST_H, 2.0) * pow(3.0
                / CONST_PI, 2.0 / 3.0) * pow(density[i] / 2.0, 5.0
                / 3.0)) / ( 20.0 * MASS_e * pow(MASS_H, 5.0 /
                3.0));
        }
        else
        {
            degenerate_pressure[i] = (CONST_H * 3.0e8 * pow(3.0 /
                CONST_PI, 1.0 / 3.0) * pow(density[i] / 2.0, 4.0 /
                3.0)) / (8.0 * pow (MASS_H, 4.0 / 3.0));
        };

        /*
        Are we there yet? ("there" being half the mass of the
            star)
        Alternatively; have things gone to nan? (Could check more
            than just temperature for nan, but as all the
            variables are interlinked there's not much point doing
            so)
        */
        if (loop == 1 && (mass[i] > (mass[0] / 2.0) ||
            isnan(temperature[i]) || temperature[i] == 0.0))
        {
            j = i;
            break;
        }
        else if (loop != 1 && (mass[i] < (mass[0] / 2.0) ||
            isnan(temperature[i]) || temperature[i] == 0.0))
        {
            k = i + 1;
            break;
        };

    }; // End calculation loop
}; // End big loop

/*
Create the output file
*/
sprintf(filename, "%sOutput.csv", directory);
output_file = fopen(filename, "w");
if (!output_file)
{
    printf("Failed to open %soutput.csv \n", directory);
```

```
    return 1;
};
fprintf(output_file, "Number,Mass,Density,Radius,Radius
    Error,Pressure,Pressure
    Error,Degeneracy,Opacity,Temperature,Temperature
    Error,del_radiative,del_adiabatic,Energy,Luminosity,Luminosity
    Error\n");

/*
Output the calculated variables
*/
for (loop = 0; loop < 2; loop++)
{
    if (loop == 0)
    {
        /*
        This is the loop starting on the surface, and working
            inwards
        Direction is positive as i is increasing - i=1 is the
            surface.
        */
        direction = +1.0;
        i_min = 0;
        i_max = k;
    }
    else
    {
        /*
        This is the loop starting at the core, and working
            outwards.
        Direction is negative as i is decreasing - i=N is the
            center.
        */
        direction = +1.0;
        i_min = j;
        i_max = N;
    };
    for (i = i_min; i < i_max; i = i + direction)
    {
        if (mass[i] != 0.0 || i > N-3)
        {
            fprintf(output_file,
                "%d,%.*Le,%.*Le,%.*Le,%.*Le,%.*Le,%.*Le,%.*Le,%.*L
                e,%.*Le,%.*Le,%.*Le,%.*Le,%.*Le,%.*Le,%.*Le\n", i,
                accuracy, mass[i], accuracy, density[i], accuracy,
                radius[i],  accuracy, err_radius[i], accuracy,
                pressure[i], accuracy, err_pressure[i],  accuracy,
                degenerate_pressure[i], accuracy, opacity[i],
                accuracy, temperature[i],  accuracy,
                err_temperature[i], accuracy, del_radiative[i],
                accuracy, del_adiabatic[i], accuracy, energy[i],
                accuracy, luminosity[i], accuracy,
                err_luminosity[i]);
        };
    };
};

fclose(output_file);

/*
All is done, so end the program.
```

```
    */
    printf("\n%d positions calculated (%d inwards, %d outwards)\n", (N
        - j) + k, k, (N - j));
    return 0;
}
```

functions.h

```
/*
PC4181 MPhys Project - Stellar Computer

Functions for star.c
Started: 23/12/2005
Last Modified: 11/01/2006

Matthew Dennison (matthew.dennison@student.manchester.ac.uk) & Mike
    Peel (email@mikepeel.net)
*/

/*
Include the standard functions
*/
#include    <math.h>
#include    <stdlib.h>

/*
Calculates the radius difference
See section 2.1 of the report for details.
*/
long double calc_radius_dif(long double prev_radius,
                            long double density)
{
    long double output;

    output = (1.0 / (4.0 * CONST_PI * pow(prev_radius, 2.0) *
        density));

    return (output);
};


/*
Calculates the pressure difference using the equation of Hydrostatic
    Equilibrium
See section 2.2 of the report for details.
*/
long double calc_pressure_dif(long double mass,
                              long double radius)
{
    long double output;

    output = ((CONST_G * mass) / (4.0 * CONST_PI * pow(radius, 4.0)));

    return (output);
};


/*
Calculates the luminosity
See section 2.3 of the report for details.
*/
```

```
long double calc_luminosity_dif(long double energy_generation)
{
    long double output;

    output = energy_generation;

    return (output);
};


/*
Calculates the temperature
See section 2.4 of the report for details.
*/
long double calc_temperature_dif(long double temperature,
                                 long double opacity,
                                 long double luminosity,
                                 long double radius,
                                 long double mass,
                                 long double pressure,
                                 long double prev_temperature,
                                 long double prev_pressure,
                                 int i)
{
    long double output;

    // This calculates the start of the differential equation, i.e.
        before del.
    output = (CONST_G * mass * temperature) / (4.0 * CONST_PI *
        pow(radius, 4.0) * pressure);

    // Calculate the adiabatic gradient, del_ad
    del_adiabatic[i] = ((log(temperature/prev_temperature)) /
        (log(pressure / prev_pressure)));

    // Calculate the radiative gradient, del_rad
    del_radiative[i] = (radiative_constant * opacity * luminosity *
        pressure) / (mass * pow(temperature, 4.0));

    if (del_radiative[i] > del_adiabatic[i] && del_adiabatic[i] > 0.0)
    {
        // This is convective.
        output = output * del_adiabatic[i];
    }
    else
    {
        // This is radiative
        output = output * del_radiative[i];
    };

    return (output);
};


/*
Calculate mu, i.e. the average molecular mass
See section 2.5 of the report for details.
*/
long double calc_mu(long double X,
                    long double Y,
                    long double Z,
```

```
                    long double A_Z)
{
    long double output;

    output = MASS_H / ((X + (Y / 4.0) + (Z / A_Z)));

    return (output);
};


/*
Calculates the density
Equation Of State - uses the Ideal Gas law plus the Radiation
    Pressure.
See section 2.5 of the report for details.
Note that mu contains m_h.
*/
long double calc_density(long double X,
                         long double temperature,
                         long double pressure)
{
    long double output;

    output = (mu / (CONST_K * temperature)) * (pressure + (1.0 / 3.0)
        * CONST_A  * pow(temperature, 4.0));

    return (output);
};


/*
Calculates the energy generation per unit volume at the specified
    temperature
See section 2.6 of the report for details.
*/
long double calc_energy_generation(long double temperature,
                                   long double density,
                                   long double X,
                                   long double Y)
{
    long double output, temp, temp_1_3, temp_2_3, alpha, psi, F_PPI,
        F_PPII, F_PPIII;

    /*
    For ease of calculation, work out temp = temperature / 10^6
    Also, temp_1_3 and temp_2_3 will also be needed several times
        over...
    */
    temp = temperature * 1e-6;
    temp_1_3 = pow(temp, (1.0 / 3.0));
    temp_2_3 = pow(temp, (2.0 / 3.0));

    /*
    F_ denotes fraction. The three variables determine the fractions
        of PPI, PPII and PPIII chains happening at the time. Used to
        adjust the output energy appropriately via psi.
    Alpha is the number of alpha particles available; it impacts psi,
        F_PPI, F_PPII and F_PPIII.
    This only becomes relevant when there's helium present in the star
        (which is nearly always the case
    */
```

```
    if (Y > 0)
    {
        alpha = 1.2e17 * pow((Y / X), 2.0) * exp(-100 / temp_1_3);
        F_PPI = (pow((1.0 + (2.0 / alpha)), 0.5) - 1.0) / (pow((1.0 +
            (2.0 / alpha)), 0.5) + 3.0);
        F_PPII = (1 - F_PPI) / (1 + (6.3 / 7.05) * 1e16 * pow(temp, (-
            1.0 / 6.0)) * exp(-102.65 * pow(temp, -1.0 / 3.0)));
        F_PPIII = (1 - F_PPI - F_PPII);
        psi = (1.0 - alpha + alpha * pow((1.0 + (2.0 / alpha)), 0.5))
            * (0.981 * F_PPI + 0.961 * F_PPII * 0.727 * F_PPIII);
    }
    else
    {
        psi = 1.0;
    };

    /*
    This deals with the pp chain
    */
    output = 2.36e2 * density * pow(X, 2.0) * pow(temp_2_3, -1) *
        exp(-33.81 / temp_1_3) * (1 + 0.0123 * temp_1_3 + 0.0109 *
        temp_2_3 * 0.00095 * temp) * psi;

    /*
    For debugging...
    printf("%e  %e  %e  %e  %e  %e", alpha, F_PPI, F_PPII, F_PPIII,
        psi, output);
    */

    return (output);
};


/*
Calculates the opacity at the specified temperature
See section 2.7 of the report for details.
*/
long double calc_opacity(long double X,
                         long double density,
                         long double temperature)
{
    long double output = 0.0;
    long double opacity_1, opacity_2, opacity_3;
    long double logR = 0.0;
    long double logT = 0.0;
    int k, k_use, l, l_use;

    // Set up a switch between the three opacities.
    int opacity_switch = 3;

    switch(opacity_switch)
    {
        case 1:
            /*
            Use temperature boundaries
            */
            if (temperature < 4e4)
            {
                output = 1.85e-14 * pow(density, 0.5) *
                    pow(temperature, 4.0);
            }
```

```
            else if (temperature < 1.26e6)
            {
                output = 4e21 * (X + Y) * (1 + X) * density *
                    pow(temperature, -3.5);
            }
            else
            {
                output = 0.02 * (1.0 + X);
            };
        break;
        case 2:
            /*
            Use the middle one of the three opacity equations - this
                should in general be the appropriate one, and avoids
                the jump between the third and second equations when
                going outwards. Unfortunately, it seems to do this by
                ignoring the third equation completely in the sun...
            */

            opacity_1 = 1.85e-14 * pow(density, 0.5) *
                pow(temperature, 4.0);
            opacity_2 = 8e21 * density * pow(temperature, -3.5);
            opacity_3 = 0.02 * (1.0 + X);

            if ((opacity_1 < opacity_2) && (opacity_1 > opacity_3))
            {
                output = opacity_1;
            }
            else if ((opacity_2 < opacity_1) && (opacity_2 >
                opacity_3))
            {
                output = opacity_2;
            }
            else
            {
                output = opacity_3;
            };
        break;
        case 3:
            /*
            Use OPAL opacity data from opacity.txt.
            Note that we start at k = l = 1, not 0 - the 0th
                row/column is the value for logT and logR.
            */

            logT = log10(temperature);
            // 1000 changes to cgs units, which the OPAL data uses
            logR = log10((density / 1000.0) * pow((1e-6 *
                temperature), 3.0));

            /*
            For debugging
            printf("%Lf, %Lf\n", logT, logR);
            */

            // Find the closest T
            for(k = 1; k < NUM_OPAL_I; k++)
            {
//              printf("%d\n", k);
                if ((opal_data[k][0] - logT) >= 0.0 || k ==
                    (NUM_OPAL_I - 1))
```

```
                {
                    // Check and see if the previous value was closer
                        or not
                    k_use = k;
                    if (k != 1)
                    {
                        if ((abs(opal_data[k-1][0] - logT)) <
                            abs(opal_data[k][0] - logT))
                        {
                            k_use = k - 1;
                        };
                    };
                    // Find the closest R
                    for(l = 1; l < NUM_OPAL_J; l++)
                    {
                        if ((opal_data[0][l] - logR) >= 0.0 || l ==
                            (NUM_OPAL_J - 1))
                        {
                            // Check and see if the previous value was
                                closer or not
                            l_use = l;
                            if (l != 1)
                            {
                                if ((abs(opal_data[0][j-1] - logR)) <
                                    abs(opal_data[0][l] - logR))
                                {
                                    l_use = l - 1;
                                };
                            };
                            output = opal_data[k_use][l_use];
                            break;
                        };
                    };
                };
            };

            // 10 changes back to SI units.
            output = pow(10, output) * 10.0;
        break;
    };

    return (output);
};

/*
The Runge-Kutta function. Calculates the next set of differences
    depending on the previous set.
All functions other than dir, i and j are globally accessible, so
    aren't explicitely passed.
dT[j], dP[j] etc. are defined previous to this function being called.
dir determines the direction we're going in - it's either -1 or 1.
*/
void rungekutta(int dir, int i, int j)
{
    temp_density[j] = calc_density(X, temperature[i - dir] + dT[j],
        pressure[i - dir] + dP[j]);
    temp_radius[j] = dm[i] * calc_radius_dif(radius[i - dir] + dR[j],
        *ptr_density);
    temp_pressure[j] = dm[i] * calc_pressure_dif(mass[i - dir] +
        dM[j], radius[i - dir] + dR[j]);
    temp_opacity[j] = calc_opacity(X, *ptr_density, temperature[i -
```

```
            dir] + dT[j]);
        temp_temperature[j] = dm[i] * calc_temperature_dif(temperature[i -
            dir] + dT[j], *ptr_opacity, luminosity[i - dir] + dL[j],
            radius[i - dir] + dR[j], mass[i - dir] + dM[j], pressure[i -
            dir] + dP[j], old_temperature, old_pressure, i);
        temp_energy[j] = calc_energy_generation(temperature[i - dir] +
            dT[j], *ptr_density, X, Y);
        temp_luminosity[j] = dm[i] * calc_luminosity_dif(*ptr_energy);


};
```