# AEP 4380 Final Project: Polytropic Model of the Sun

Gianfranco Grillo

December 12, 2016

## 1 Introduction

Stars are among the most important objects in astrophysics, and as such they have been studied using the methods of modern science for more than one hundred years. Early attempts at figuring out how stars worked were focused on attempting to learn more about the age of the Sun. When Charles Darwin proposed his theory of evolution by natural selection in 1859, a major argument against his theory was based on the idea that the Sun could not burn long enough in order for the extremely slow process of natural selection to have time to develop the immense variety of lifeforms that people were able to see around the planet. According to the physics of the time, the Sun was powered by the Kelvin-Helmholtz process, with the Sun's radiation being generated by its gravitational energy. Lord Kelvin famously calculated that this would only allow the Sun to shine for about 10 million years, a period of time not nearly large enough to allow for the types of organic transformations that Darwin had in mind. It wasn't until the discovery of nuclear forces, and the pioneering work of Hans Bethe in the early 20th century, that it became known that the Sun's energy originated from hydrogen fusion in its core. Around the same time, Ejnar Hertzprung and Henry Norris Russell introduced the now ubiquitous Hertzprung-Russell diagram, which classifies stars based on their luminosity and spectral temperature. In a sense, serious and successful stellar astrophysics took off at this point, as scientists attempted to derive the trajectories that stars of different masses traced on the HR diagram as they evolved, and also tried to explain their composition, which they were able to observe by examining their spectral signatures. Among their most spectaculars findings is the fact that the elements that make up organisms like ourselves were produced in the interiors of stars.

This project aims to construct a simplified model for the structure of the Sun, and use numerical methods to calculate and plot stellar quantities of interest. This document is divided as follows: Section 2 describes the relevant stellar astrophysics and the approximations used. Section 3 discusses the numerical methods used and the structure of the implemented code. Results are presented in Section 4, followed by a short analysis in Section 5. The code used is attached as an appendix.

## 2 Stellar Astrophysics

### 2.1 The Equations of Stellar Structure

The structure of a perfectly symmetric, spherical star at a specific instant in time, and which is in hydrostatic equilibrium is governed by a set of four coupled differential equations, as follows:

$$\frac{\partial r}{\partial m} = \frac{1}{4\pi r^2 \rho} \tag{2.1}$$

$$\frac{\partial P}{\partial m} = -\frac{Gm}{4\pi r^4} \tag{2.2}$$

$$\frac{\partial L}{\partial m} = \epsilon \tag{2.3}$$

$$\frac{\partial T}{\partial m} = -\frac{GmT}{4\pi r^4 P}\nabla \tag{2.4}$$

Here, $r$ corresponds to the star's radius, $m$ corresponds to its mass, $L$ is its luminosity, $T$ its temperature, and $\rho$ its density. $\epsilon$ is the star's rate of energy production, and the variable $\nabla$ is given by

$$\nabla = \frac{d \ln T}{d \ln P} \tag{2.5}$$

This is known as the Lagrangian formulation, in which the independent variable is taken to be the star's mass. The Eulerian formulation, on the other hand, takes the radius as the independent variable. We will be working with the Lagrangian formulation, which has proven more popular in the context of numerical calculations involving stellar structure and evolution.

The interpretation of these equations is straightforward. Equation (2.1) arises as a consequence of mass conservation. Equation (2.2) is the equation of hydrostatic equilibrium: the gravitational force acting inwards is exactly balanced by the force acting outwards arising from the star's pressure gradient. Equation (2.3) is the equation of conservation of energy: the rate at which the star releases energy is equal to the rate at which it produces energy. Finally, equation (2.4) is the equation of energy transport, and arises from thermodynamics. It describes the way energy moves outwards from the inner regions of the star to its outer regions, and subsequently to outer space. Equation (2.5) is trickier (which makes (2.4) trickier, too), as its exact form will depend on whether energy transport is due to convection or radiation, and will include terms that describe the opacity, which can be very complicated. $\rho$ and $\epsilon$ are (in general) functions of $P$, $T$, and the star's chemical composition.

These equations cannot be solved simply as initial value problems, or as regular boundary value problems: we do not have the proper boundary conditions. We know that, at the center, $m = r = L = 0$. At the surface, there is more than one way of setting the boundary conditions, ranging from very simple to more complicated expressions. One possible choice, which we will take here, is to adopt the so-called radiative zero boundary conditions at the surface. We assume that there is a hard boundary between the star and outer space, so that at the surface, $m = M$, the star's total mass, and $P = T = 0$. Technically, these boundary conditions are not exactly correct, because there is no such thing as a hard boundary between the star's surface and its environment; however, it is possible to construct an approximate model using these conditions. In any case, we have four boundary conditions for four equations, as we don't know either what the pressure and temperature are at the center, or what are the star's luminosity and radius. This is an example of a two point boundary value problem. In principle, given a star's total mass, we can solve equations (1)-(4) for the respective dependent quantities. In practice, this is not a task that is easily accomplished in an accurate manner, especially because the equations can be unstable, and because the quantity described in equation (4) is notoriously difficult to calculate correctly. As mentioned before, its exact form depends on whether the transport of energy is radiative or convective, and that, in turn, depends on what region of the star is being integrated; another difficulty arises from the quantity's dependence on the opacity, whose value is extremely complicated to get right. Therefore, we will only attempt to solve the first two equations in order to find the central pressure $P_c$ and the radius $R$ for a given mass $M$. We will then rely on a series of approximations in order to get values for the the central temperature $T_c$, the central density $\rho_c$, the total luminosity $L_T$, and the temperature at the surface $T_{eff}$.

## 2.2 Polytropes and the Lane-Emden Equation

We will model the dependency between $\rho$ and $P$ by assuming a polytropic model,

$$P = K \rho^{\frac{n+1}{n}} \tag{2.6}$$

where $K$ is a constant of proportionality, and $n$ is the polytropic index. This, however, introduces a new unknown into our equations, the parameter $K$. This parameter can be calculated for a given mass by solving the Lane-Emden equation, which will be derived below.

Let us rewrite equations (2.1) and (2.2) by changing the independent variable from $m$ to $r$:

$$\frac{\partial m}{\partial r} = 4\pi r^2 \rho \tag{2.7}$$

$$\frac{\partial P}{\partial r} = -\frac{Gm}{r^2}\rho \tag{2.8}$$

We can combine these two equations into a second order differential equation:

$$\frac{1}{r^2}\frac{d}{dr}\left(\frac{r^2}{\rho}\frac{dP}{dr}\right) = -4\pi G\rho \tag{2.9}$$

By introducing the variables $\rho = \rho_c\theta^n$, $P = P_c\theta^{n+1}$, $r = \alpha\xi$, we can define $\alpha^2$ to be given by

$$\alpha^2 = \frac{K(n+1)\rho_c^{\frac{1-n}{n}}}{4\pi G} \tag{2.10}$$

With these new definitions, where $\rho_c$ is the central density, $P_c$ is the central pressure, $\theta$ a dimensionless quantity commonly referred to as the polytropic temperature, and $\xi$ another dimensionless variable, we can use (2.6) in order to express (2.9) as follows:

$$\frac{1}{\xi^2}\frac{d}{d\xi}\left(\xi^2\frac{d\theta}{d\xi}\right) + \theta^n = 0 \tag{2.11}$$

which is known as the Lane-Emden equation. The boundary conditions for this equation are

$$\theta = 1, \quad \tfrac{d\theta}{d\xi} = 0 \quad , at\, \xi = 0 \tag{2.12}$$

In the present context, we can regard this as the star's center. This explains why we require $\frac{d\theta}{d\xi} = 0$ at $\xi = 0$: we need$\theta$at that point to be finite. From the definitions above, it should be clear that the point $\theta = 0$ represents the surface of the star, the point in which the pressure and the density go to zero. We define $\xi_1$ as the value of $\xi$ at which this occurs.

We can now derive the relationship between the total mass $M$, the radius $R$, and $K$:

$$R = \alpha\xi_1 = \left(\frac{K(n+1)\rho_c^{\frac{1-n}{n}}}{4\pi G}\right)^{\frac{1}{2}}\xi_1 \tag{2.13}$$

$$M = \int_0^R 4\pi r^2\rho\, dr = 4\pi\alpha^3\rho_c\int_0^{\xi_1}\xi^2\theta^n\, d\xi \tag{2.14}$$

After some lengthy manipulation, (13) yields

$$M = 4\pi\left(\frac{K}{G}\frac{n+1}{4\pi}\right)^{\frac{3}{2}}\rho_c^{\frac{3-n}{2n}}\left(-\xi^2\frac{d\theta}{d\xi}\right)_{\xi=\xi_1} \tag{2.15}$$

Which can be combined with (12) to get

$$R^{\frac{3-n}{n}}M^{\frac{n-1}{n}} = \frac{K}{GN_n} \tag{2.16}$$

where

$$N_n = \frac{(4\pi)^{\frac{1}{n}}}{n+1}\left[\left(-\xi^2\frac{d\theta}{d\xi}\right)_{\xi=\xi_1}\right]^{\frac{1-n}{n}}\xi_1^{\frac{n-3}{n}} \tag{2.17}$$

Our model will be made using a polytropic index of $n = 3$, which eliminates $R$ from equation (2.16), and $\xi_1$ in (2.17), and we end up with

$$M^{\frac{2}{3}} = \frac{K}{GN_3} \tag{2.18}$$

$$N_3 = \frac{(4\pi)^{\frac{1}{3}}}{4}\left[\left(-\xi^2\frac{d\theta}{d\xi}\right)_{\xi=\xi_1}\right]^{-\frac{2}{3}} \tag{2.19}$$

The Lane-Emden equation is only solvable analytically when $n$ is 0, 1, or 5, so we will need to solve the equation numerically in our case. The procedure for doing so is described in section 3.3. The solution will enable us to

| $T_7$ | $\epsilon_0\,(cgs)$ | $\nu$ |
|---|---|---|
| 1 | $7 \times 10^{-2}$ | 4.60 |
| 2 | 1 | 3.54 |
| 4 | 9 | 2.72 |
| 8 | 43 | 2.08 |

Table 2.1: Temperature Dependence of $\epsilon_0$ and $\nu$ for the pp-cycle

calculate the parameter $N_3$ and, given a total mass $M$, we will be able to derive $K$, which will be assumed to be a fixed parameter. We can then use that value of $K$ to solve equations (2.1) and (2.2) using the shooting method (described in section 3.4), in order to find $P_c$ and $R$, since the $n = 3$ polytropic relationship implies that (1) is now

$$\frac{\partial r}{\partial m} = \frac{K^{\frac{3}{4}}}{4\pi r^2 P^{\frac{3}{4}}} \tag{2.20}$$

## 2.3 Ideal Gas Law

We will approximate the relationship between $P$ and $T$ throughout the star by assuming an ideal gas. This has the effect of essentially eliminating equation (2.4), as $P$ can be calculated at each step just by solving (2.1) and (2.2), and we can then calculate the value of $T$ by using the ideal gas equation, given by

$$P = \frac{k_B}{\mu m_p}\rho T \tag{2.21}$$

where $k_B$ is the Boltzmann constant, $\mu$ is the average molecular weight of the gas in units of $m_p$, the proton mass. For the chemical composition of the Sun, $\mu = 0.6$. Since we are assuming a polytropic equation of state for $\rho$, with $n = 3$, we have that

$$P = \frac{k_B}{\mu m_p}\left(\frac{P}{K}\right)^{\frac{3}{4}} T \tag{2.22}$$

$$T = \frac{\mu m_p P^{\frac{1}{4}} K^{\frac{3}{4}}}{k_B} \tag{2.23}$$

This is certainly a very crude approximation as far as the shape of the plot of $T$ vs $M$ is concerned, so we will make no attempt to actually produce a model of the temperature inside the star at arbitrary radii. Instead, we will use this approximation in order to calculate the star's central temperature, and to approximate the total amount of energy produced inside the star via the pp-chain.

## 2.4 Energy Generation

We will make no attempt to produce an accurate plot of $L$ vs $M$ inside the star, because such a plot necessarily will depend on an accurate representation of $T$ vs $M$, which we already have determined will not be accurate given the assumptions described on the previous section. Thus, we will make no attempt to accurately solve equation (2.3). Instead, we will attempt to calculate the total amount of energy generated in the star by the following procedure. First, we will assume that all the energy comes from nuclear reactions involving the pp-chain. With that assumption, the rate of energy generation $\epsilon$ can be approximated by

$$\epsilon \approx \epsilon_0 \rho T_7^{\nu} \tag{2.24}$$

where $\epsilon_0$ and $\nu$ are functions of temperature themselves, and $T_7 = \frac{T}{10^7 K}$. These functions are complicated, but we can construct approximations for them based on the following table of values, as given by [4]:

We will model $\epsilon_0(T)$ as a third-degree polynomial, and $\nu(T)$ as a logarithmic function. Performing a least-squares fit for the parameters of such models, we obtain, in cgs units,

$$\epsilon_0(T) \quad = \quad aT^3 + bT^2 + cT + d \tag{2.25}$$

whereas the dimensionless parameter $\nu(T)$ will be given by

$$\nu(T) = e\ln(T) + f \tag{2.26}$$

The values for the parameters $a$, $b$, $c$, $d$, $e$ and $f$ are given in Table 2. Figures 2.1 and 2.2 give an idea of how good these approximations are[1]. We can now use (2.24), (2.25), (2.26), and (2.21) in order to express $\epsilon$ as a function of $P$ and $T$, as follows:

$$\epsilon \approx \frac{(aT^3 + bT^2 + cT + d)P\mu m_p T_7^{e\ln(T)+f}}{k_B} \tag{2.27}$$

We will then use (2.22) to calculate $T$ at each $P$. This will only use this relationship to approximate the star's total luminosity; as stated before, trying to use (2.22) to plot $T$ vs $M$ will result in an extremely inaccurate picture. An accurate plot requires solving equation (2.4), which will not be attempted in this work. We will integrate (2.3) by using this approximation, but only to find the value of the total luminosity $L_T$, not to make a plot of $L$ as a function of $M$.

| $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|
| $-3.905 \times 10^{-23}$ | $1.297 \times 10^{-14}$ | $-2.687 \times 10^{-7}$ | 1.499 | $-1.209$ | 23.98 |

Table 2.2: Coefficients used in (20)

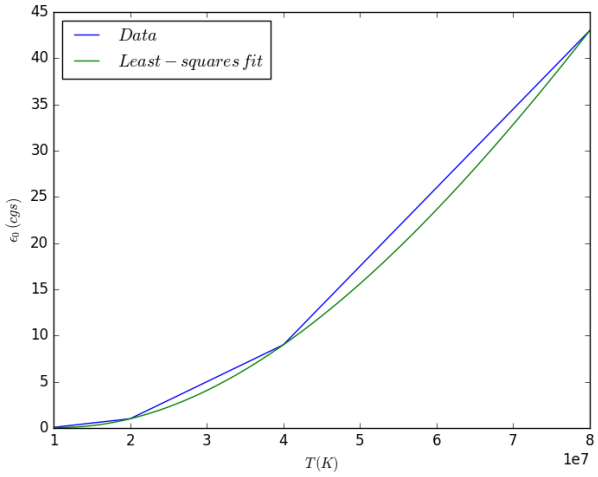


Figure 2.1: Data and least-squares fit for $\epsilon_0(T)$

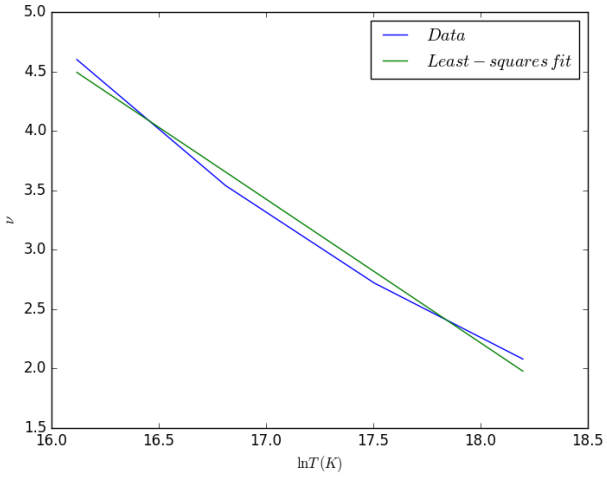


Figure 2.2: Data and least-squares fit for $\nu(T)$

## 2.5 Effective Temperature at the Surface

We can use Stefan-Boltzmann's Law to find the temperature at the surface $T_{eff}$, using the value for the total luminosity $L_T$ and the radius $R$. Stefan-Boltzmann's Law for the luminosity of a perfect spherical blackbody is

[1] I have performed the least-squares fits using Python.

given by:

$$L_T = 4\pi R^2 \sigma T_{eff}^4 \tag{2.28}$$

where $\sigma = 5.670 \times 10^{-5}\, erg\, cm^{-2}\, K^4\, s$ the Stefan-Boltzmann constant. Solving for $T_{eff}$ gives:

$$T_{eff} = \left(\frac{L_T}{4\pi R^2 \sigma}\right)^{\frac{1}{4}} \tag{2.29}$$

# 3 Numerical Procedure

## 3.1 General Considerations

As mentioned in section 1.2.1, the physics behind equations (2.1)-(2.4) imply that this is not the kind of problem that can be solved simply by numerically integrating from a specified initial value, and stopping the integration at an exact final point. Since we will only attempt to solve equations (2.1), (2.2), and a simplified version of (2.3), as discussed above, we will decrease the number of variables involved, but at the same time we will be decreasing the number of boundary conditions. We will be solving for $r$ and $P$, with the boundary conditions being that at the center, $m = 0$ and $r = 0$, whereas at the surface, $m = M$, and $P = 0$. First, however, we need to find the value of the $K$ parameter, which we will do by numerically solving (2.11) for $n = 3$. In order to do so, we will use a 5th order autostep size Runge-Kutta-Fehlberg algorithm. The next section gives a brief overview of this algorithm.

## 3.2 5th Order Runge-Kutta-Fehlberg

The Runge-Kutta-Fehlberg method for numerically solving a system of coupled ODEs is an extension of the more basic 4th/5th order regular Runge-Kutta algorithms that has the added advantage of being able to automatically adjust the step size $h$ to be used by estimating the error at each iteration.

Given a system of coupled ODEs of the form

$$\frac{d\vec{x}}{dt} = \vec{f}(t,\, \vec{x}(t)) \tag{3.1}$$

With initial conditions

$$\vec{x}(t_0) = \vec{x_0} \tag{3.2}$$

Where $t$ is an independent variable, the algorithm calculates six $k$ parameters in order to progressively reconstruct the solutions of the ODE system, and use the difference between a high order solution and a low order solution to estimate the error resulting from the use of a particular $h$. These parameters are given by

$$\begin{aligned}
\vec{k_1} &= h\vec{f}(t_n, \vec{x}_n) && (3.3)\\
\vec{k_2} &= h\vec{f}(t_n + c_2 h,\, \vec{x}_n + a_{21}\vec{k}_1) && (3.4)\\
\vec{k_3} &= h\vec{f}(t_n + c_3 h,\, \vec{x}_n + a_{31}\vec{k}_1 + a_{32}\vec{k}_2) && (3.5)\\
\vec{k_4} &= h\vec{f}(t_n + c_4 h,\, \vec{x}_n + a_{41}\vec{k}_1 + a_{42}\vec{k}_2 + a_{43}\vec{k}_3) && (3.6)\\
\vec{k_5} &= h\vec{f}(t_n + c_5 h,\, \vec{x}_n + a_{51}\vec{k}_1 + a_{52}\vec{k}_2 + a_{53}\vec{k}_3 + a_{54}\vec{k}_4) && (3.7)\\
\vec{k_6} &= h\vec{f}(t_n + c_6 h,\, \vec{x}_n + a_{61}\vec{k}_1 + a_{62}\vec{k}_2 + a_{63}\vec{k}_3 + a_{64}\vec{k}_4 + a_{65}\vec{k}_5) && (3.8)
\end{aligned}$$

where the $c$'s and $a$'s are coefficients known as the Dormund-Prince coefficients. The higher order solution is then given by

$$\vec{x}_{n+1} = \vec{x}_n + b_1\vec{k}_1 + b_2\vec{k}_2 + b_3\vec{k}_3 + b_4\vec{k}_4 + b_5\vec{k}_5 + b_6\vec{k}_6 + \mathcal{O}(h^6) \tag{3.9}$$

whereas the lower order order solution is

$$\vec{x'}_{n+1} = \vec{x}_n + b_1^{'}\vec{k}_1 + b_2^{'}\vec{k}_2 + b_3^{'}\vec{k}_3 + b_4^{'}\vec{k}_4 + b_5^{'}\vec{k}_5 + b_6^{'}\vec{k}_6 + b_7^{'}h\vec{f}(t_n + h,\, \vec{x}_{n+1}) + \mathcal{O}(h^5) \tag{3.10}$$

where $b_n$ and $b_n^{'}$ are also Dormund-Prince coefficients. The error in the current step is given by

$$\epsilon_{n+1}^{(i)} = \left| x_{n+1}^{(i)} - x_{n+1}^{'(i)} \right| \tag{3.11}$$

where $i = 1,\, 2,\, 3, ...,\, N$. The error is then scaled in order to adjust for the different magnitudes and units of the variables using a scale given by

$$scale^{(i)} = \left| x_n^{(i)} \right| + \left| h f^{(i)}(t_n, y_n^{(i)}(t)) \right| \tag{3.12}$$

This is used to find the maximum (relative) error $\Delta_{max}$ in the series, via

$$\Delta_{max} = max \left[ \frac{\epsilon_{n+1}^{(i)}}{scale^{(i)}} \right] \tag{3.13}$$

Given a predetermined maximum allowed error $\epsilon_{max}$, the algorithm changes the step size $h$ in two cases. When $\Delta_{max} \ll \epsilon_{max}$, $h$ is increased

$$h_{new} = h_{old} \left( \frac{\epsilon_{max}}{\Delta_{max}} \right)^{1/5} \tag{3.14}$$

And when $\Delta_{max} > \epsilon_{max}$, $h$ is reduced by an arbitrary factor $n$, giving

$$h_{new} = \frac{h_{old}}{n} \tag{3.15}$$

In the last case, the current iteration step is repeated. The final $\vec{x}_{n+1}$ corresponds to the next value of the solution.

This procedure is repeated however many times is needed in order to reach the end of the integration. We will rely on this numerical procedure to solve the Lane-Emden equation. The algorithm will also be a part of the strategy used to solve the two point boundary value problem given by equations (2.1) and (2.2).

## 3.3 Solving the Lane-Emden Equation

For a polytropic index of $n = 3$, equation (2.11) becomes

$$\frac{1}{\xi^2}\frac{d}{d\xi}\left(\xi^2 \frac{d\theta}{d\xi}\right) + \theta^3 = 0 \tag{3.16}$$

The numerical procedure described in the previous section requires us to express the equation to be solved in terms of two coupled first order differential equations. We can do that by expressing it in the form

$$2\frac{d\theta}{d\xi} + \xi\frac{d^2\theta}{d\xi^2} + \xi\theta^3 = 0 \tag{3.17}$$

and performing the substitution $\Gamma = \frac{d\theta}{d\xi}$, which gives:

$$2\Gamma + \xi\frac{d\Gamma}{d\xi} + \xi\theta^3 = 0 \tag{3.18}$$

Solving for $\frac{d\Gamma}{d\xi}$ yields:

$$\frac{d\Gamma}{d\xi} = \frac{-2\Gamma - \xi\theta^3}{\xi} \tag{3.19}$$

and thus we obtain a system of coupled first order differential equations in the required form,

$$\begin{bmatrix} \frac{d\Gamma}{d\xi} \\ \frac{d\theta}{d\xi} \end{bmatrix} = \begin{bmatrix} \frac{-2\Gamma - \xi\theta^3}{\xi} \\ \Gamma \end{bmatrix} \tag{3.20}$$

The boundary conditions were given in (2.12). Notice that because the boundary conditions are given at $\xi = 0$, and equation (3.19) contains a singularity at that point. This means that we need to start the integration some distance away from that point in order for our algorithm to work. This can be done by expanding (3.20) in a Taylor series at $\xi = 0$ in order to obtain an approximate value for $\theta$ and $\Gamma$ close to $\xi = 0$, and start the integration at that point. In other words, we assume that there exists a solution close to 0 of the form

$$\theta(\xi \approx 0) = c_0 + c_1\xi + c_2\xi^2 + c_3\xi^3 + c_4\xi^4 + \cdots + \mathcal{O}(\xi^n) \tag{3.21}$$

$$\frac{d\theta}{d\xi}(\xi \approx 0) = \Gamma(\xi \approx 0) = c_1 + 2c_2\xi + 3c_3\xi^2 + 4c_4\xi^3 + \cdots + \mathcal{O}(\xi^{n-1}) \tag{3.22}$$

The boundary conditions require that $c_0 = 0$ and $c_1 = 1$. The rest of the coefficients can be obtained by plugging (3.21) and (3.22) into (3.20)[2]. We end up with

$$\theta(\xi \approx 0) = 1 - \frac{\xi^2}{6} + \frac{\xi^4}{40} - \frac{19\xi^6}{5040} + \mathcal{O}(\xi^7) \tag{3.23}$$

$$\Gamma(\xi \approx 0) = -\frac{\xi}{3} + \frac{\xi^3}{10} - \frac{19\xi^5}{840} + \mathcal{O}(\xi^6) \tag{3.24}$$

The code used starts the integration at $\xi = 0.01$. It implements the Runge-Kutta-Fehlberg algorithm and integrates until $\theta < 0$, at which points it stops and records the value of $\xi_1$ and $\frac{d\theta}{d\xi} = \Gamma$, which is then used to calculate $N_3$ as per equation (2.19).

## 3.4 Solving the Two Point Boundary Value Problem

We will attempt to solve the two point boundary value problem presented by (2.1) and (2.2), together with their initial conditions, as follows. Using the value of the $K$ parameter obtained via direct integration of the Lane-Emden equation as described in (3.3), we can rewrite (2.2) in the form shown in (2.20). We will use Runge-Kutta-Fehlberg to integrate, starting the integration at $m = 0$ and some small distance removed from the center, at $r = 10\,\mu m$, in order to avoid hitting the singularity[3], and ending at the surface where $P = 0$. We don't know the value of $P$ at the center, $P_c$, so we start with an initial guess for $P_c$ and integrate until the point before $P < 0$.[4] We proceed then to record the value of $r$ and $m$ at that point. Then, we quantify how well the chosen value of $P_c$ matches the boundary conditions. We do this by adding the absolute value of $P$ at the integration step immediately preceding the one in which $P$ becomes negative, to that of the quantity $\frac{m-M}{M}$ at that same point. The closer this value is to zero, the better the match with the boundary conditions is. We repeat this procedure for multiple values of $P_c$, after which we search for the value of $P_c$ that yields the least error. The code used relies on user input for three parameters: the lower and upper limits for the guess of $P_c$, and the number of trial integrations to be performed over that range, each of which is separated by a constant $\Delta P_c$. After finding the minimum error, the integration is performed again for the corresponding value of $P_c$, except now all the quantities calculated at each integration step are recorded in order to be able to produce plots of $P$ and $\rho$ as a function of $r$, and obtain the values of $R$, $T_c$, $\rho_c$, $L_T$, and $T_{eff}$.

[2] I have done this using Mathematica, following the procedure in [5].

[3] An attempt was made to properly expand the solution near $m = 0$ for $r$ and $P$ in a Taylor series, in a manner similar as it was done in the case of the Lane-Emden equation. Unfortunately, the attempt was unsuccesful, as the code started behaving improperly for some unknown reason. Starting with $r = 10\,\mu m$ when $m = 0$ circumvented the problem, but it seems like there is something weird going on here.

[4] A more straightforward way of doing this would be simply to integrate until $m = M$. However, the value of (2.20) becomes complex once $P < 0$ because of the $P^{\frac{3}{4}}$ term, and thus the code breaks down when trying to calculate the next value of $r$.

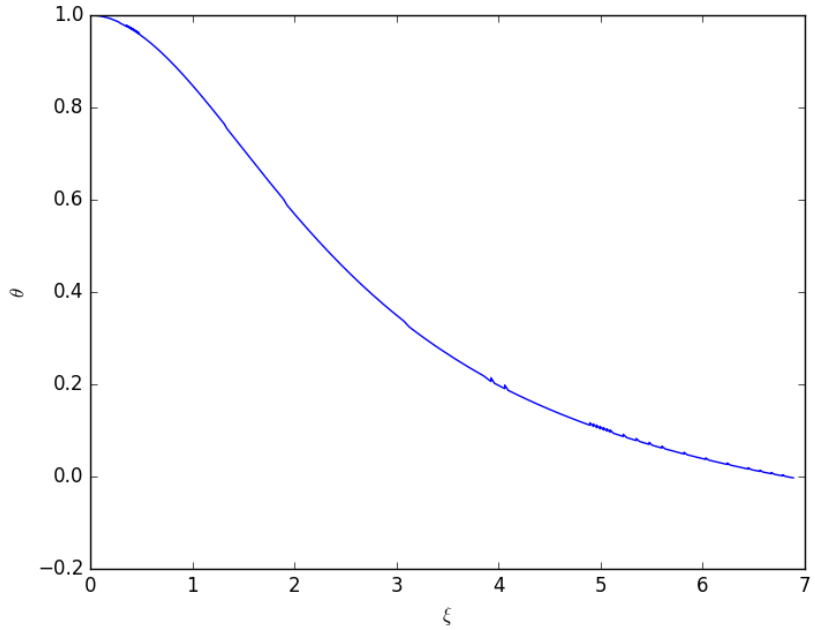


Figure 4.1: Numerical solution of the Lane-Emden equation with $n = 3$

# 4 Results

## 4.1 Numerical Solution of the Lane-Emden Equation

Figure 4.1 shows a plot of $\xi$ vs $\theta$ resulting from the procedure described in section 3.3. The values found for the dimensionless parameters of interest were:

$$\xi(\theta = 0) = \xi_1 \quad = \quad 6.888 \tag{4.1}$$

$$\Gamma(\theta = 0) = \left(\frac{d\theta}{d\xi}\right)_{\xi=\xi_1} \quad = \quad -0.04254 \tag{4.2}$$

This results in a value for $N_3$ of

$$N_3 = 0.3639 \tag{4.3}$$

which is identical to that found by Chandrasekhar, as mentioned in [6]. We now use this result to calculate the value of the parameter $K$ for a star with $M = M_\odot = 1.999 \times 10^{33}\, g$, the solar mass, using (2.18):

$$K = M_\odot^{\frac{2}{3}} G N_3 = 3.841 \times 10^{14}\, dyne\, cm^2\, g^{-\frac{4}{3}} \tag{4.4}$$

These will be the values of $K$ and $M$ that will be used moving forward.

## 4.2 Numerical Solution of the Two Point Boundary Value Problem

The results obtained for a series of combinations of the lower and upper boundaries of the initial guess range [Low $P_c$; High $P_c$] are shown in Table 4.1, as a fraction of the directly measured values of the solar quantities, and those arising from more sophisticated analyses. The number of samples in all cases was equal to 200.

| Low $P_c$ | High $P_c$ | $P_c$ | $M$ | $R$ | $T_c$ | $L_T$ | $T_{eff}$ | $\rho_c$ |
|---|---|---|---|---|---|---|---|---|
| 0.1 | 5.0 | 0.927 | 1.19 | 0.438 | 0.927 | 4.18 | 2.16 | 0.888 |
| 0.6 | 1.4 | 0.948 | 1.20 | 0.466 | 0.883 | 2.43 | 1.83 | 0.767 |
| 0.7 | 1.3 | 0.928 | 1.27 | 0.506 | 0.878 | 2.35 | 1.74 | 0.755 |
| 0.8 | 1.2 | 1.15 | 1.19 | 0.438 | 0.927 | 4.14 | 2.16 | 0.886 |
| 0.9 | 1.1 | 1.07 | 1.22 | 0.457 | 0.909 | 3.37 | 2.00 | 0.837 |

Table 4.1: Results for different ranges of initial guesses

Figures 4.2 and 4.3 show plots for $P$ vs $r$ and $\rho$ vs $r$ for the last run, the one with the smallest $\Delta P_c$, overlapped by plots made using the Solar Model S by Christensen-Dalsgaard [7]. Note that the x-axis for the "Grillo Model" is given in terms of the calculated solar radius, not the actual one.

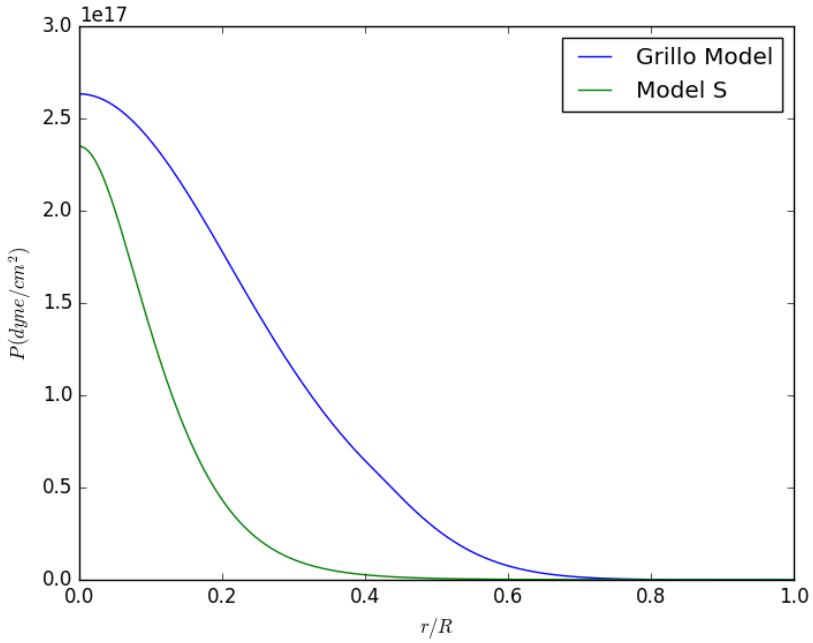


Figure 4.2: Comparison of $P$ vs $r$ plots for polytropic model vs Model S

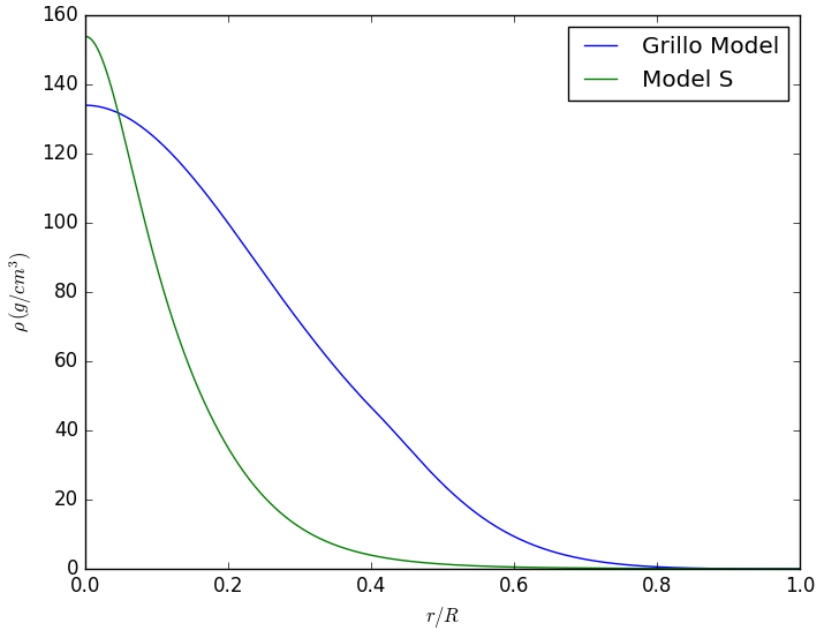


Figure 4.3: Comparison of $\rho$ vs $r$ plots for polytropic model vs Model S

# 5 Analysis

## 5.1 Software

The software worked well. The Lane-Emden equation was solved properly, and the shooting method was able to arrive at roughly the same answer for the value of $P_c$, even when the the guess range was quite wide. It could be improved by adapting it so that it always employs the same step size instead of the same number of steps as it iterates over the guess range, but this would require more computational time as the range increases in size. The simple scheme used to find the minimum error could be replaced by a more sophisticated Newton-Raphson method, or modified as to be able to distinguish instances in which it finds the minimum error at the end or beginning of the array, in which case the guess range could be made wider. A more sophisticated analysis would require the use of a relaxation method instead of the shooting method, as relaxation has proven to be more reliable in the construction of stellar models, especially ones in which the more complicated equation (4) is solved, and in which the star is evolved in time.

## 5.2 Adequacy of the Model

The results show that the model used is fairly accurate at modeling the Sun's central pressure, density, and temperature. The radius is consistently calculated to be about half of the actual radius, the luminosity is overestimated by a factor in the range between 2 and 4, and the surface temperature is overestimated by a factor of roughly 2. A better model would be required to alter the boundary conditions to account for the fact that the pressure does not actually drop to zero at $R$, given that there is no hard boundary between the star's surface and outer space. It would also have to actually solve equation (4) in order to be able to accurately describe the way the temperature changes in the star's interior, which would improve the luminosity calculations tremendously. Figures 4.3 and 4.4 provide a good comparison between this model and a more sophisticated one. On the Model S, the density and pressure go to zero faster than on the polytropic approximation used in this work. Given the crudeness of the analysis, however, the results are well within expectations.

# References


[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery *Numerical Recipes, The Art of Scientific Computing, 3rd edit.*, Camb. Univ. Press, 2007.

[2] R. Kippenhahn, A. Weigert, and A. Weiss *Stellar Structure and Evolution, 2nd edit.,* Springer-Verlag Berlin Heidelberg, 2012.

[3] R. Kippenhahn, A. Weigert, and E. Hofmeister: "Methods for Calculating Stellar Evolution," *Methods in Computational Physics*, vol. 7 (Eds.: B. Adler, S. Fernbach, and M. Rotenberg), Academic, New York, 1967, pp. 129-190.

[4] G. W. Collins, *The Fundamentals of Stellar Astrophysics,* W. H. Freeman and Company, 1989.

[5] "Lane-Emden Function: Numerical Solution". Scientificarts.com. N.p., 2016. Web. 12 Dec. 2016. http://www.scientificarts.com/laneemden/Links/laneemden_lnk_1.html.

[6] B. Paczynski. "Polytropes". AST 403 Course Notes. Web. 10 Dec. 2016. Available at http://www.astro.princeton.edu/~gk/A403/polytrop.pdf

[7] Christensen-Dalsgaard et al., "The Current State of Solar Modeling". *Science,* vol. 272, p. 1286-1292.

# 7 Appendix

## 7.1 Lane-Emden

```
/* Lane-Emden equation
  Run on core i5 with g++ 5.4.0
  Gianfranco Grillo 12/10/2016
*/

#include <cstdlib>
#include <cmath>

#include <iostream>
#include <fstream>
#include <iomanip>
#include "arrayt.hpp"
using namespace std;

int main() {
    int n = 2, i;
    double h, hold, x, N, pi = 4.0 * atan(1.0);
    arrayt<double> initvar(n), finalvar(n);
    void autosteprk4(arrayt<double>&, arrayt<double>&, int, double&, double, void(
       arrayt<double>&, double, arrayt<double>&, double&));
    void laneemden(arrayt<double>&, double, arrayt<double>&, double&);
    ofstream fp;
    x = 0.01;
    initvar(0) = 1-x*x/6.0+x*x*x*x/40.0-19.0*pow(x,6.0)/5040.0;
    initvar(1) = -x/3.0 + x*x*x/10.0-19.0*pow(x,5.0)/840.0;
    h = 1e-3;
    fp.open("lanemden.dat");
    while (x < 10.0) {
        hold = h;
        autosteprk4(initvar, finalvar, n, h, x, laneemden);
        fp << finalvar(0) << setw(15) << finalvar(1) << setw(15) << x << endl;
        if (finalvar(0) < 0.0) break;
        if (h > hold || h == hold) {
            for (i = 0; i < n; i++) { initvar(i) = finalvar(i); }
            x = x + hold;
        }
    }
    N = pow(4*pi,1.0/3.0)/4.0*pow(-x*x*initvar(1),-2.0/3.0);
    cout << "xi1 = " << x << setw(15) << "gamma1 = " << initvar(1) << setw(15) << "
       N = " << N << endl;
    fp.close();
    return (EXIT_SUCCESS);
}

void autosteprk4(arrayt<double>& initvar, arrayt<double>& highfinalvar, int n,
   double& h,
                                 double t0, void functions(arrayt<double>&, double,
                                     arrayt<double>&, double&)) {
```

```
    int i;
    arrayt<double> scale(n), error(n), lowfinalvar(n), k1(n), k2(n), k3(n), k4(
       n), k5(n), k6(n), k7(n), scalek(n), temp(n);
double maxerror = 1e-10, deltamax;
    double maximum(arrayt<double>&);
    double c2 = 0.2, c3 = 0.3, c4 = 0.8, c5 = 8.0/9.0, c6 = 1.0, c7 = 1.0;
    double a21 = 0.2, a31 = 3.0/40.0, a32 = 9.0/40.0, a41 = 44.0/45.0, a42 =
       -56.0/15.0, a43 = 32.0/9.0;
    double a51 = 19372.0/6561.0, a52 = -25360.0/2187.0, a53 = 64448.0/6561.0,
       a54 = -212.0/729.0;
    double a61 = 9017.0/3168.0, a62 = -355.0/33.0, a63 = 46732.0/5247.0, a64 =
       49.0/176.0, a65 = -5103.0/18656.0;
    double b1 = 35.0/384.0, b3 = 500.0/1113.0, b4 = 125.0/192.0, b5 =
       -2187.0/6784.0, b6 = 11.0/84.0;
    double d1 = 5179.0/57600.0, d3 = 7571.0/16695.0, d4 = 393.0/640.0, d5 =
       -92097.0/339200.0, d6 = 187.0/2100.0, d7 = 1.0/40.0;
    functions(initvar, t0, k1, h); // generate k1
    for (i = 0; i < n; i++) {
            temp(i) = initvar(i) + a21*k1(i);
    }
    functions(temp, t0 + c2*h, k2, h);
    for (i = 0; i < n; i++) {
            temp(i) = initvar(i) + a31*k1(i) + a32*k2(i);
    }
    functions(temp, t0 + c3*h, k3, h);
    for (i = 0; i < n; i++) {
            temp(i) = initvar(i) + a41*k1(i) + a42*k2(i) + a43*k3(i);
    }
    functions(temp, t0 + c4*h, k4, h);
    for (i = 0; i < n; i++) {
            temp(i) = initvar(i) + a51*k1(i) + a52*k2(i) + a53*k3(i) + a54*k4(i
               );
    }
    functions(temp, t0 + c5*h, k5, h);
    for (i = 0; i < n; i++) {
            temp(i) = initvar(i) + a61*k1(i) + a62*k2(i) + a63*k3(i) + a64*k4(i
               ) + a65*k5(i);
    }
    functions(temp, t0 + c6*h, k6, h);
    for (i = 0; i < n; i++) {
            highfinalvar(i) = initvar(i) + b1*k1(i) + b3*k3(i) + b4*k4(i) + b5*
               k5(i) + b6*k6(i);
    }
    functions(highfinalvar, t0 + h, k7, h);
    for (i = 0; i < n; i++) {
            lowfinalvar(i) = initvar(i) + d1*k1(i) + d3*k3(i) + d4*k4(i) + d5*
               k5(i) + d6*k6(i) + d7*k7(i);
    }
    functions(initvar, t0 + h, scalek, h); // generate scale
    for (i = 0; i < n; i++) {
            scale(i) = abs(initvar(i)) + abs(scalek(i)) + 0.01;
    }
```

```
        for (i = 0; i < n; i++) {
                error(i) = abs((highfinalvar(i) - lowfinalvar(i))/scale(i));
        }
        deltamax = maximum(error);
        if (5*deltamax < maxerror) {
                h = h*pow(maxerror/deltamax, 0.2); // if error is too small,
                    increase h
                return;
        }
        else if (deltamax > maxerror && abs(h) > 1e-12) { // if error is too big,
           decrease h
                h = h/3.0;
                return;
        }
        else {
                return;
        }
}

void laneemden(arrayt<double>& var, double x, arrayt<double>& k, double& h) {
    double y,z;
    y = var(0);
    z = var(1);
    k(0) = h*z;
    k(1) = h*(-2*z-x*y*y*y)/x;
        return;
}

double maximum(arrayt<double>& array) {
        double max;
        int i, n = array.n();
        max = array(0);
        for (i = 0; i < n; i++) {
                if (array(i) > max) {
                        max = array(i);
                }
        }
        return max;
}
```

## 7.2 Stellar Equations

```
/* Solving the Stellar Structure Equations
  Run on core i5 with g++ 5.4.0
  Gianfranco Grillo 12/03/2016
*/

#include <cstdlib>
#include <cmath>

#include <iostream>
#include <fstream>
```

```
#include <iomanip>
#include "arrayt.hpp"
using namespace std;

int main() {
    int npts = 1e5, i, j, n = 3, nsteps, minpos;
    double K = 3.841e14, pi = 4.0 * atan(1.0), M = 1.9891e33, h, hold, startP,
       stopP, stepwP, m = 0, T, L, centralT, centralP, mp = 1.672e-24, mu = 0.6, kb
        = 1.381e-16, eps0, nu, totalL = 0.0, surfaceT, sig = 5.670e-5, R, rho, G =
       6.6732e-8;
    void stellar(arrayt<double>&, double, arrayt<double>&, double&, double);
    void autosteprk4(arrayt<double>&, arrayt<double>&, int, double&, double, double
       , void(arrayt<double>&, double, arrayt<double>&, double&, double));
    int minloc(arrayt<double>&);
    ofstream fp;
    cout << "Enter␣pressure␣lower␣limit:␣" << endl;
    cin >> startP;
    cout << "Enter␣pressure␣upper␣limit:␣" << endl;
    cin >> stopP;
    cout << "Enter␣number␣of␣steps:␣" << endl;
    cin >> nsteps;
    startP = startP*2.47e17;
    stopP = stopP*2.47e17;
    arrayt<double> initvar(n), err(nsteps), finalvar(n), Plist(nsteps);
    stepwP = (stopP-startP)/nsteps;
    h = M/npts;
    for (i = 0; i < nsteps; i++) {
        m = 0;
        Plist(i) = startP + i*stepwP;
        initvar(0) = 1e-5;
        initvar(1) = Plist(i);
        initvar(2) = 0;
        while (m < 2*M) {
            hold = h;
            autosteprk4(initvar, finalvar, n, h, m, K, stellar);
            if (finalvar(1) < 0.0) break;
            if (h > hold || h == hold) { // means error is low enough, result is
               correct
                for (j = 0; j < n; j++) { initvar(j) = finalvar(j); }
                m = m + h;
            }
        }
        err(i) = abs(m/M)+abs(finalvar(1));
    }
    minpos = minloc(err);
    m = 0;
    h = M/npts;
    centralP = Plist(minpos);
    initvar(0) = 1e-5;
    initvar(1) = centralP;
    initvar(2) = 0;
    fp.open("solution.dat");
```

```
    while (m < 2*M) {
        hold = h;
        autosteprk4(initvar, finalvar, n, h, m, K, stellar);
        if (finalvar(1) < 0.0) break;
        if (h > hold || h == hold) {
            for (i = 0; i < n; i++) { initvar(i) = finalvar(i); }
            rho = pow(initvar(1)/K, 0.75);
            fp << m << setw(15) << initvar(0) << setw(15) << initvar(1) << setw(15)
                << rho << setw(15) << endl;
            m = m + h;
        }
    }
    fp.close();
    R = initvar(0);
    centralT = mu*mp*pow(centralP, 0.25)*pow(K, 0.75)/kb;
    totalL = initvar(2);
    surfaceT = pow(totalL/(4*pi*sig*R*R), 0.25);
    cout << "Mass = " << m/M << "      " << "Radius = " << R/6.9598e10 << "       "
        << "Central pressure = " << centralP/2.47e17 << "       " << "Central
       Temperature = " << centralT/1.57e7 << "      " << "Luminosity = " << totalL
       /3.83e33 << "    " << "Surface temperature = " << surfaceT/5772 << "     "
       << "Central density = " << pow(centralP/K, 0.75)/160 << endl;
    return (EXIT_SUCCESS);
}

void autosteprk4(arrayt<double>& initvar, arrayt<double>& highfinalvar, int n,
    double& h,
                                 double t0, double K, void functions(arrayt<double
                                    >&, double, arrayt<double>&, double&, double))
                                    {
        int i;
        arrayt<double> scale(n), error(n), lowfinalvar(n), k1(n), k2(n), k3(n), k4(
            n), k5(n), k6(n), k7(n), scalek(n), temp(n);
    double maxerror = 1e-10, deltamax;
        double maximum(arrayt<double>&);
        double c2 = 0.2, c3 = 0.3, c4 = 0.8, c5 = 8.0/9.0, c6 = 1.0, c7 = 1.0;
        double a21 = 0.2, a31 = 3.0/40.0, a32 = 9.0/40.0, a41 = 44.0/45.0, a42 =
            -56.0/15.0, a43 = 32.0/9.0;
        double a51 = 19372.0/6561.0, a52 = -25360.0/2187.0, a53 = 64448.0/6561.0,
            a54 = -212.0/729.0;
        double a61 = 9017.0/3168.0, a62 = -355.0/33.0, a63 = 46732.0/5247.0, a64 =
            49.0/176.0, a65 = -5103.0/18656.0;
        double b1 = 35.0/384.0, b3 = 500.0/1113.0, b4 = 125.0/192.0, b5 =
            -2187.0/6784.0, b6 = 11.0/84.0;
        double d1 = 5179.0/57600.0, d3 = 7571.0/16695.0, d4 = 393.0/640.0, d5 =
            -92097.0/339200.0, d6 = 187.0/2100.0, d7 = 1.0/40.0;
        functions(initvar, t0, k1, h, K); // generate k1
        for (i = 0; i < n; i++) {
                temp(i) = initvar(i) + a21*k1(i);
        }
        functions(temp, t0 + c2*h, k2, h, K);
        for (i = 0; i < n; i++) {
```

```
                temp(i) = initvar(i) + a31*k1(i) + a32*k2(i);
        }
        functions(temp, t0 + c3*h, k3, h, K);
        for (i = 0; i < n; i++) {
                temp(i) = initvar(i) + a41*k1(i) + a42*k2(i) + a43*k3(i);
        }
        functions(temp, t0 + c4*h, k4, h, K);
        for (i = 0; i < n; i++) {
                temp(i) = initvar(i) + a51*k1(i) + a52*k2(i) + a53*k3(i) + a54*k4(i
                    );
        }
        functions(temp, t0 + c5*h, k5, h, K);
        for (i = 0; i < n; i++) {
                temp(i) = initvar(i) + a61*k1(i) + a62*k2(i) + a63*k3(i) + a64*k4(i
                    ) + a65*k5(i);
        }
        functions(temp, t0 + c6*h, k6, h, K);
        for (i = 0; i < n; i++) {
                highfinalvar(i) = initvar(i) + b1*k1(i) + b3*k3(i) + b4*k4(i) + b5*
                    k5(i) + b6*k6(i);
        }
        functions(highfinalvar, t0 + h, k7, h, K);
        for (i = 0; i < n; i++) {
                lowfinalvar(i) = initvar(i) + d1*k1(i) + d3*k3(i) + d4*k4(i) + d5*
                    k5(i) + d6*k6(i) + d7*k7(i);
        }
        functions(initvar, t0 + h, scalek, h, K); // generate scale
        for (i = 0; i < n; i++) {
                scale(i) = abs(initvar(i)) + abs(scalek(i)) + 0.01;
        }
        for (i = 0; i < n; i++) {
                error(i) = abs((highfinalvar(i) - lowfinalvar(i))/scale(i));
        }
        deltamax = maximum(error);
        if (5*deltamax < maxerror) {
                h = h*pow(maxerror/deltamax, 0.2); // if error is too small,
                    increase h
                return;
        }
        else if (deltamax > maxerror && abs(h) > 1e-12) { // if error is too big,
            decrease h
                h = h/3.0;
                return;
        }
        else {
                return;
        }
}

void stellar(arrayt<double>& y, double m, arrayt<double>& k, double& h, double K) {
    double pi = 4.0 * atan(1.0), r, P, G = 6.6732e-8, T, rho, mp = 1.672e-24, mu =
        0.6, kb = 1.381e-16, eps0, nu;
```

```
    r = y(0);
    P = y(1);
    rho = pow(P/K, 0.75);
    T = mu*mp*P/(kb*rho);
    eps0 = -3.90476190e-23*T*T*T+1.29666667e-14*T*T-2.68666667e-07*T+1.49904762;
    nu = -1.209*log(T)+23.98;
    k(0) = h/(4*pi*r*r*rho);
    k(1) = -h*G*m/(4*pi*r*r*r*r);
    k(2) = h*eps0*rho*pow(T/1e7,nu);
        return;
}

double maximum(arrayt<double>& array) {
        double max;
        int i, n = array.n();
        max = array(0);
        for (i = 0; i < n; i++) {
                if (array(i) > max) {
                        max = array(i);
                }
        }
        return max;
}

int minloc(arrayt<double>& array) {
    double m;
        int i, n = array.n(), minpos;
        m = array(0);
    minpos = 0;
        for (i = 0; i < n; i++) {
                if (array(i) < m) {
                        m = array(i);
            minpos = i;
                }
        }
    return minpos;
}
```