CO31: Structure of white dwarf stars

Lyubomir Shoylev, shil5377
lyubomir.shoylev@st-hildas.ox.ac.uk
St Hilda's College
February 26, 2021

Abstract

In an experiment, we attempt to study the structure of white dwarf stars. We give a short derivation of the differential equations describing a white dwarf in equilibrium under some assumptions about the matter. Then we introduce the computational approach of our solution and decide on parameters for the calculation. We present the dependence of radius and mass of white dwarfs on central density, as well as the relationship between them, which gives rise to the *Chandrasekhar mass limit*. We discuss some of its implications, after which we present a summary of the experiment and comment on further improvements.

1 Introduction

The goal of this experiment is to study the internal structure of white dwarf stars. They are one possible end state in the life cycle of stars. What is particular about white dwarf stars is that the matter they are composed of is different from the matter in a usual star similar to the Sun. While in a usual stable star the equilibrium is largely supported by fusion energy and the pressure of the plasma, a white dwarf star has reached the end of the fusion processes, and the equilibrium is supported by electron degeneracy pressure, making the star extremely dense; for example, a white dwarf of one solar mass is about the size of Earth. In this report, we will first derive the coupled differential equations for mass and density in a spirit similar to Oxford Physics (2020) and Chandrasekhar (1984) in Section 2. Then, in Section 3, we develop the numerical method used to solve the coupled equations. Results and discussion take place in Section 4. Finally, we give a summary of the experiment and possible improvements in Section 5.

2 Theory of white dwarf stars

Stars are astronomical objects with ellipsoid shape made up of heated plasma. The force of gravity tries to compress the ball, holding the matter together, while the gaseous plasma opposes gravity with the pressure it exerts. When the two forces balance each other out, the star is in hydrostatic equilibrium. In usual stars, pressure is due to the gas and the radiation. In white dwarf stars, however, the densities are far greater than the ones found in usual stars, and matter behaves differently. All electrons are no longer bound to atoms and are free to roam. The behaviour can approximately be modelled by a Fermi gas of electrons at zero Kelvin (i.e. fully degenerate state). Hence, the pressure ensuring the equilibrium is the degeneracy pressure of the electrons.

2.1 Equation of equilibrium

Assume the star is spherically symmetric, in equilibrium, non-rotating, and the effect of magnetic fields is negligible. Therefore, all properties depend only on the distance to the centre of the star, r. The gravitational force on a small volume of matter with area $\mathrm{d}A$ and radial height $\mathrm{d}r$ is:

$$F_{\mathrm{G}} = -\frac{Gm\left(r\right)\mathrm{d}m}{r^2}, \tag{1}$$

where $m\left(r\right)$ is the mass of the star contained up to r, $\mathrm{d}m = \mathrm{d}A\,\mathrm{d}r\rho\left(r\right)$ is the mass of the small volume, and the density $\rho\left(r\right)$ was assumed constant (negative sign because it pulls towards the centre). The force due to the pressure is the difference of the forces at r and $r + \mathrm{d}r$:

$$F_{\mathrm{P}} = \left(P\left(r\right) - P\left(r + \mathrm{d}r\right)\right)\mathrm{d}A. \tag{2}$$

For a star in equilibrium, the two forces balance each other out, i.e. $F_{\mathrm{G}} + F_{\mathrm{P}} = 0$. After rearranging, we get:

$$\frac{\mathrm{d}P(r)}{\mathrm{d}r} = \frac{P\left(r + \mathrm{d}r\right) - P\left(r\right)}{\mathrm{d}r} = -\frac{G\rho(r)m(r)}{r^2}. \tag{3}$$

We can rewrite the hydrostatic equilibrium by using the chain rule:

$$\frac{\mathrm{d}P(r)}{\mathrm{d}r} = \frac{\mathrm{d}P}{\mathrm{d}\rho}\frac{\mathrm{d}\rho}{\mathrm{d}r}, \tag{4}$$

so equation (3) becomes:

$$\frac{\mathrm{d}\rho}{\mathrm{d}r} = -\frac{\mathrm{d}\rho}{\mathrm{d}P}\frac{G\rho(r)m(r)}{r^2}. \tag{5}$$

On the other hand, the equation for the inscribed mass is:

$$m(r) = \int_0^r \mathrm{d}r' 4\pi^2 r' \rho\left(r'\right) \quad \Rightarrow \quad \frac{\mathrm{d}m}{\mathrm{d}r} = 4\pi r^2 \rho(r) \tag{6}$$

We are now left with two coupled differential equations, (5) and (6). The last task before obtaining the full equation is to determine $\frac{\mathrm{d}P}{\mathrm{d}\rho}$ which depends on the equation of state.

2.2 Equation of state

We assume the star is made up primarily of heavy ^{56}Fe nuclei and their electrons. The nuclei carry most of the mass and little of the pressure, while the opposite is true for the electrons. Since *very high* densities are considered, we can approximate that the nuclei are stationary while the electrons move freely, not bound to any nuclei. A good model for the freely moving electron gas is the free Fermi gas at $T = 0\,\mathrm{K}$. This is the fully degenerate state, in which electrons fill all energy (momentum) levels up to the Fermi energy (momentum).

Fermions obey the Pauli exclusion principle, therefore each energy level is $2S+1$ degenerate, where $S = \frac{1}{2}$ in the case of electrons. This degeneracy factor multiplies gives the density function:

$$g(k) = \frac{2S+1}{2\pi^2} V k^2, \tag{7}$$

where k is the wave vector of particles and V is the volume of the system. The number density of particles can therefore be written as:

$$n = \frac{N}{V} = \frac{1}{V}\int_0^{k_\mathrm{F}} \mathrm{d}k\, g(k) = \int_0^{k_\mathrm{F}} \mathrm{d}k \frac{2S+1}{2\pi^2} k^2. \tag{8}$$

Here we integrate up to k_F where this is given by $p_\mathrm{F} = \hbar k_\mathrm{F}$ and p_F is the Fermi momentum. Rewrite the above equation in terms of p:

$$n = \int_0^{p_\mathrm{F}} \mathrm{d}p \frac{2S+1}{2\pi^2\hbar^3} p^2 = \int_0^{p_\mathrm{F}} \mathrm{d}p \frac{8\pi}{h^3} p^2 = \int_0^{p_\mathrm{F}} \mathrm{d}p\, n(p) = \frac{m_\mathrm{e} 8\pi}{h^3} p_\mathrm{F}^3 \tag{9}$$

where we define $n(p)\,\mathrm{d}p$ as the number of electrons per unit volume with momentum between p and $p + \mathrm{d}p$.

Pressure, on the other hand, is given by the kinetic expression:

$$P = \frac{1}{3}\int_0^{p_\mathrm{F}} p v_p n(p)\,\mathrm{d}p. \tag{10}$$

At this point we need to differentiate between non-relativistic and relativistic cases - v_p has a different value in the two cases:

$$v_p = \begin{cases} \frac{p}{m_\mathrm{e}} & \text{in the non-relativistic case} \\ \frac{pc^2}{\sqrt{p^2c^2 + m_\mathrm{e}^2 c^4}} & \text{in the relativistic case} \end{cases}. \tag{11}$$

Let us first work out the answer in the non-relativistic case. The integral becomes const $\times \int p^4\,\mathrm{d}p$, so the pressure becomes:

$$P_\mathrm{non} = \frac{8\pi}{15h^3 m_\mathrm{e}} p_\mathrm{F}^5. \tag{12}$$

For the relativistic case, rather than do the integral for P involving hyperbolic functions, differentiate (10) by parts:

$$\frac{\mathrm{d}P_\mathrm{rel}}{\mathrm{d}\rho} = \frac{\mathrm{d}P_\mathrm{rel}}{\mathrm{d}p_\mathrm{F}}\frac{\mathrm{d}p_\mathrm{F}}{\mathrm{d}\rho}, \quad \frac{\mathrm{d}P_\mathrm{rel}}{\mathrm{d}p_\mathrm{F}} = \frac{8\pi}{3h^3}\frac{\mathrm{d}}{\mathrm{d}p_\mathrm{F}}\int_0^{p_\mathrm{F}} \frac{p^4c^2}{\sqrt{p^2c^2 + m_\mathrm{e}^2c^4}}\,\mathrm{d}p = \frac{8\pi}{3h^3}\frac{p_\mathrm{F}^4 c}{\sqrt{p_\mathrm{F}^2c^2 + m_\mathrm{e}^2c^4}}. \tag{13}$$

We can collect the derivative of (12) and the result of (13):

$$\frac{\mathrm{d}P}{\mathrm{d}\rho} = \frac{\mathrm{d}p_\mathrm{F}}{\mathrm{d}\rho}\frac{8\pi}{3h^3 m_\mathrm{e}} p_\mathrm{F}^4 \times \begin{cases} 1 & \text{in the non-relativistic case} \\ \left(1 + \frac{p_\mathrm{F}^2}{m_\mathrm{e}^2 c^2}\right)^{-1/2} & \text{in the relativistic case} \end{cases}. \tag{14}$$

What is left is expressing $p_\mathrm{F}(\rho)$ to complete the equation of state in the form needed by (5). The relation between density and number density can be stated as $\rho = \mu_\mathrm{e} m_\mathrm{p} n$, where μ_e is the mean molecular weight per electron and m_p is the mass of 1 proton. In our case, we can approximate $\mu_\mathrm{e} \approx 2$. Using this relation and (9), we arrive at:

$$p_\mathrm{F} = \left(\frac{h^3}{8\pi} n\right)^{1/3} = \left(\frac{h^3}{16\pi m_\mathrm{p}} \rho\right)^{1/3} \quad \Rightarrow \quad \frac{\mathrm{d}p_\mathrm{F}}{\mathrm{d}\rho} = \frac{1}{3}\left(\frac{h^3}{16\pi m_\mathrm{p}}\right)^{1/3} \rho^{-2/3} \tag{15}$$

2.3 The limiting mass

The limiting case of ultra-relativistic electrons can be obtained by setting $v_p = c$ (i.e. taking the limit $pc \gg m_\mathrm{e} c^2$) in (10) to arrive at the equation of state:

$$P_\mathrm{ultra} = \frac{p\pi}{3h^3}\int_0^{p_\mathrm{F}} p^3 c \,\mathrm{d}p = \frac{2\pi c}{3h^3} p_\mathrm{F}^4. \tag{16}$$

As noted in Chandrasekhar (1984), this leads to a specific equation for the mass with a well-specified value $M_\mathrm{lim} = 5.76\mu_\mathrm{e}^{-2} M_\odot$, where $M_\odot$ is one solar mass; this limit is equivalent to infinite mean density and zero radius of the star. In the approximation made above the limiting mass is $M_\mathrm{lim} \approx 1.435 M_\odot$. This will be "experimentally" verified by the computations presented in Section 4. We can draw two important conclusions from this fact: first, there is an upper limit to masses of degenerate stars in the later stages of evolution; and second, we cannot predict the end evolutionary state of stars with mass $M > M_\mathrm{lim}$ in this physical framework (we expand more on this in Section 4).

3 Numerical approach

To find the radius and mass of the star as a function of its central density, we will numerically integrate the coupled system. The distance r at which $\rho(r = R) \approx 0$ is where the star ends; the mass is, therefore, $m(R)$ where $m(r)$ is the inscribed mass, as defined in (6). Physically, we can consider solving the equation as an initial value problem. First, write the coupled system in vector form:

$$\boldsymbol{y} = \begin{pmatrix} \rho \\ m \end{pmatrix}, \tag{17}$$

and the derivatives of its members with respect o r given by (5) and (6). Then, for a given central density ρ_c the initial condition for the mass is specified by $m(\delta r) = \frac{4}{3}\pi(\delta r)^3 \rho_\mathrm{c}$ for some small distance from the centre δr: this is done so we can start the calculations ($r = 0$ would give initial zero mass and the derivative is therefore zero). We will set up the equation solver so that this δr is the step size in radial distance. The source code can be found in Appendix A.

3.1 Why integration?

Since we have an initial value problem, the underlying idea is the following: rewrite the derivatives as fractions of finite differences, and get $\Delta\boldsymbol{y} = \frac{\mathrm{d}\boldsymbol{y}(x)}{\mathrm{d}x}\Delta x$. This is the change in $\boldsymbol{y}$ when we step through by Δx. When the step is very small, the approximation is very good, i.e. $\lim_{\Delta x \to \mathrm{d}x} \Delta\boldsymbol{y} = \mathrm{d}\boldsymbol{y}$.

The basic idea boils down to the most elementary such method, the explicit Euler's method:

$$\begin{aligned} \boldsymbol{y}_{\mathrm{n}+1} &= \boldsymbol{y}_\mathrm{n} + h\boldsymbol{f}(x_\mathrm{n}, \boldsymbol{y}_\mathrm{n}) \\ x_{\mathrm{n}+1} &= x_\mathrm{n} + h, \end{aligned} \tag{18}$$

where the vector function is just $\boldsymbol{f} = \frac{\mathrm{d}\boldsymbol{y}(x)}{\mathrm{d}x}$ i.e. the slope at x_n and $h = \Delta x$. This is a first order method: the local error is $\sim h^2$ and the global error, therefore, is $\sim h$. This method has only one computation of the derivative and is therefore very fast, but also very inaccurate.

A similar explicit second order method is Heun's method (trapezoid rule):

$$\begin{aligned} \boldsymbol{y}_{\mathrm{n}+1} &= \boldsymbol{y}_{\mathrm{n}} + \frac{1}{2}h\Big(\boldsymbol{f}(x_{\mathrm{n}}, \boldsymbol{y}_{\mathrm{n}}) + \boldsymbol{f}\big(x_{\mathrm{n}} + h, \boldsymbol{y}_{\mathrm{n}} + \boldsymbol{f}(x_{\mathrm{n}}, \boldsymbol{y}_{\mathrm{n}})\big)\Big) \\ x_{\mathrm{n}+1} &= x_{\mathrm{n}} + h, \end{aligned} \tag{19}$$

Here the local error is $\sim h^3$ and the global error therefore is $\sim h^2$.

The method we will later use to compute properties of white dwarf stars is the 4th order Runge-Kutta (RK4) method. It features four evaluations of the derivative and a global error $\sim h^4$, offering a balance between computational time and accuracy. This method takes values for the slope at four different positions and estimates the step size by a weighted average of these four steps. In equations:

$$\begin{aligned} \boldsymbol{k_1} &= h\boldsymbol{f}(x_{\mathrm{n}}, \boldsymbol{y}_{\mathrm{n}}) \\ \boldsymbol{k_2} &= h\boldsymbol{f}\left(x_{\mathrm{n}} + \frac{1}{2}h, \boldsymbol{y}_{\mathrm{n}} + \frac{1}{2}\boldsymbol{k_1}\right) \\ \boldsymbol{k_3} &= h\boldsymbol{f}\left(x_{\mathrm{n}} + \frac{1}{2}h, \boldsymbol{y}_{\mathrm{n}} + \frac{1}{2}\boldsymbol{k_2}\right) \\ \boldsymbol{k_4} &= h\boldsymbol{f}(x_{\mathrm{n}} + h, \boldsymbol{y}_{\mathrm{n}} + \boldsymbol{k_3}). \end{aligned}$$

Finally, the new position is calculated as:

$$\boldsymbol{y}_{\mathrm{n}+1} = \boldsymbol{y}_{\mathrm{n}} + \frac{1}{6}\boldsymbol{k_1} + \frac{1}{3}\boldsymbol{k_2} + \frac{1}{3}\boldsymbol{k_3} + \frac{1}{6}\boldsymbol{k_4}. \tag{20}$$

The RK methods family gives approximate solutions to Nth order by varying the weights in the $\boldsymbol{k}$ expressions and the final expression to match coefficients in the Taylor expansion.

We should note that the equation to be solved might prove difficult to solve, even using RK4. These equations are often called *stiff* and in constructing their solutions more precaution needs to be taken, for example by using implicit or semi-implicit methods.

3.2 Comparison of methods

In this section, we compare the stability/accuracy of the three methods presented in Section 3.1. For the task, we will try to integrate a solution of the simple harmonic oscillator (SHO), namely:

$$\ddot{x} + \omega^2 x = 0, \qquad \text{with solution} \qquad x = x_0 \cos(\omega t) \tag{21}$$

where we choose $x_0 = 1$ and $\omega = 2\pi$ so the period of oscillations is 1 s. From some initial runs integrating the white dwarf equation, we are aware that the number of intervals relevant to us is on the order of 10^6, so we choose to compare the two methods on this scale (more discussion on the topic in Section 3.4).

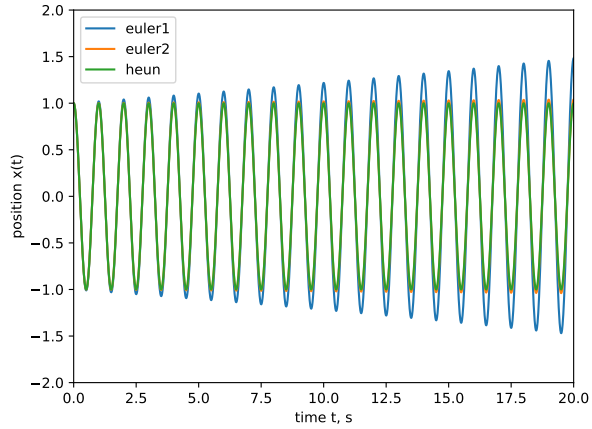
The runs are split into two groups. The first group is with 10^6 time intervals of length 1 ms for the three methods; denote the results from these runs by Euler1, Heun, and RK4, respectively. The second group is 10^7 time intervals of length 0.1 ms for the Euler method to cover the same total time interval; denote results by Euler2. Additionally, we will consider the energy in the system:

$$E = \frac{\dot{x}^2}{2} + \frac{\omega^2 x^2}{2}. \tag{22}$$

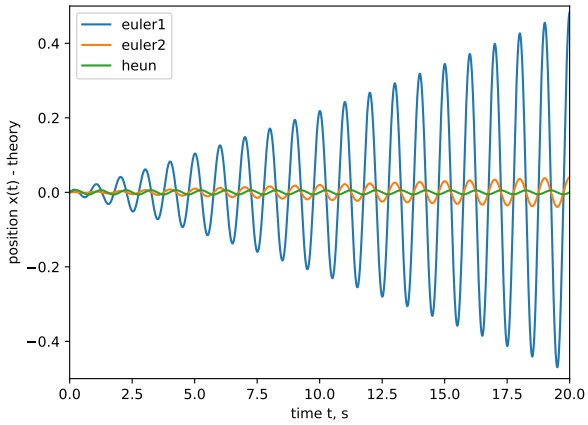
When the method over/underestimates the correction, the effect will build up and change the total energy, which is equivalent to the change in amplitude.

Results are presented in Figure 1. In 1a (first 20 periods of oscillation displayed), we see the calculated position by the three methods Euler1, Euler2, and Heun. It is immediately clear that Euler1 goes off in the solution pretty quickly. This is confirmed in 1b, where we can have a better look at how the numerical solution deviates from the actual one - Euler1 gains an additional half amplitude in 20 periods, while the other two methods oscillate about zero with Heun having the smaller error. To compare for the whole time of integration and compare these three runs with RK4, we plot the relative energy deviation compared to the theoretical value of $2E_0 = \omega^2 \times 1$ on a logarithmic scale. As it can be seen in 1c and 1d, the energy for the first three runs grows exponentially with time after some moment with different exponents, while the fourth run RK4 keeps almost a constant value of energy — the change in energy between $t_0 = 0\,\mathrm{s}$ and $t_1 = 100\,\mathrm{s}$ is:

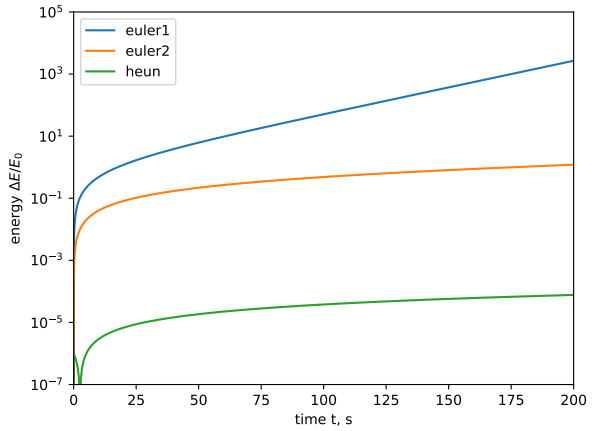
$$\frac{\big|E(t_1) - E(t_0)\big|}{E(t_0)} \approx 9.755 \times 10^{-7}. \tag{23}$$



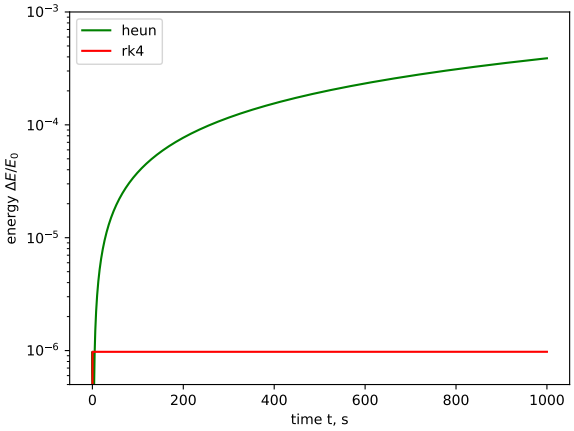
(a) Position as a function of time. Notice how eul1 deviates very fast.



(b) Deviation from the theoretical solution.



(c) Energy deviation $\Delta E/E_0$ for first three methods.



(d) Energy deviation $\Delta E/E_0$ for heun and rk4.

Figure 1: Test runs of SHO integrated with different methods. Blue line is for Euler's method integrated with step $1\,\mathrm{ms}$, yellow line for Euler's method integrated with step $0.1\,\mathrm{ms}$, green line is for Heun's method with step $1\,\mathrm{ms}$, and red line is for RK4 method with step $1\,\mathrm{ms}$.

With these results in mind, we make the clear choice for a method and continue forwards using the Runge-Kutta 4th order method in all of the following chapters.

3.3 Dimensionless equation for the white dwarf

To calculate values with the computer, we will go to dimensionless quantities in the differential equation. Introduce the following substitutions:

$$\rho = \rho_\mathrm{c}\theta, \quad r = \ell\xi, \quad m = \rho_\mathrm{c}(1\,\mathrm{m}^3)\mu, \text{ therefore } \boldsymbol{y}(r) = \begin{pmatrix}\rho(r)\\ m(r)\end{pmatrix} \to \boldsymbol{\eta}(\xi) = \begin{pmatrix}\theta(\xi)\\ \mu(\xi)\end{pmatrix}. \tag{24}$$

We also specify the length ℓ by the equation $\frac{4}{3}\pi\ell^3 = 1\,\mathrm{m}^3$, which gives a value of $\ell \approx 0.62 \sim 1\,\mathrm{m}$. This will be useful to make some cancellations in the coefficients in the final form of the differential equation for $\boldsymbol{\eta}$. The initial conditions therefore become $\boldsymbol{\eta}(1) = (\theta(1), \mu(1)) = (1.0, 1.0)$.

For (6) the conversion is straightforward:

$$\frac{\rho_\mathrm{c}\,\mathrm{d}\mu}{\ell\,\mathrm{d}\xi} = 4\pi\ell^2\xi^2\rho_\mathrm{c}\theta \quad \Rightarrow \quad \frac{\mathrm{d}\mu}{\mathrm{d}\xi} = 3\xi^2\theta. \tag{25}$$

For the θ equation we have to take into consideration the regime we operate in — whether we solve for a relativistic or non-relativistic equation of state. Firstly, write equations in (15) using the new coordinates we have introduced:

$$p_{\mathrm{F}} = \left(\frac{h^3\rho_{\mathrm{c}}}{16\pi m_{\mathrm{p}}}\right)^{1/3}\theta^{1/3}, \quad \frac{\mathrm{d}p_{\mathrm{F}}}{\mathrm{d}\rho} = \frac{1}{3}\left(\frac{h^3}{16\pi m_{\mathrm{p}}\rho_{\mathrm{c}}^2}\right)^{1/3}\theta^{-2/3}. \tag{26}$$

Now substitute (14) in (5) using (26). After some algebra with the constants, the final result is the following:

$$\frac{\mathrm{d}\theta}{\mathrm{d}\xi} = K_1\frac{\mu\theta^{1/3}}{\xi^2}\begin{cases}1 & \text{in the non-relativistic case}\\ \left(1+K_2\theta^{2/3}\right)^{1/2} & \text{in the relativistic case}\end{cases}, \tag{27}$$

where the constants K_1 and K_2 are given by

$$K_1 = -2^{13/3}\pi \mathrm{G} m_{\mathrm{e}} m_{\mathrm{p}}^{5/3}\rho_{\mathrm{c}}^{1/3}h^{-2}, \qquad K_2 = \frac{1}{m_{\mathrm{e}}^2c^2}\left(\frac{3h^3\rho_{\mathrm{c}}}{16\pi m_{\mathrm{p}}}\right)^{2/3}. \tag{28}$$

We have completed the task of bringing the equation to normalized coordinates. This is the setup used in implementing the code for the white dwarf ODE wrapper presented in Appendix A.1.

3.4 Convergence and number of intervals

What is left is to choose the coarseness in the grid of ξ. We know that for some $\Delta\xi$ small enough, the answers will converge to $\approx$ a single value. To find the number of intervals (NoI) at which the result converges, we make experimental runs at three different ρ_{c} — 10^6, 10^{10}, and 10^{14} $\mathrm{kg\,m^{-3}}$ (this spans the range for which we run the experiment later), and test for an NoI in the range $\left\{a\times 10^b \mid a\in\{1,2,4,8\}, b\in\{2,3,\dots 8\}\right\}$. The maximal radius is set to 8.1×10^7 in units of ξ since this turns out to be about the maximum physical radius a white dwarf can be for the range of ρ_{c} we test. Both the non-relativistic and relativistic cases are considered.

The results are presented in Figure 2. We use the relative error in computed value with respect to the converged value. The latter is estimated as the average of the last three values in the calculation, i.e. for NoI 2×10^8, 4×10^8, and 8×10^8; the result is $M_{\mathrm{conv}}, R_{\mathrm{conv}}$. The relative error is given by:

$$\frac{\Delta M}{M_{\mathrm{conv}}} = \frac{|M-M_{\mathrm{conv}}|}{M_{\mathrm{conv}}}, \quad \frac{\Delta R}{R_{\mathrm{conv}}} = \frac{|R-R_{\mathrm{conv}}|}{R_{\mathrm{conv}}}. \tag{29}$$

What we aim for in our choice for the number of intervals is: on the one side, a low enough number so the computation does not take too long, and on the other side, a high enough number so an acceptable accuracy is achieved. From the logarithmic plots for the mass and the radius determination, we see that the computed value does not really converge to a specific point; it does, however, reduce its error exponentially fast. Let us choose the desired accuracy of below $10^{-5} = 0.001\%$, given that our model is not very accurate. From the logarithmic plots again we see that this is satisfied for NoI $\geq 2\times 10^6$. Therefore, we can use this value as the NoI with a good compromise between speed and accuracy.

4 Results and discussion

We begin by choosing the parameters of the simulation. As discussed in Section 3.4, we use the RK4 method of integration. The range of integration in ξ will be $\left[1, 8.1\times 10^7\right]$ with 2×10^6 intervals (we implement the white dwarf wrapper in this way to speed up the procedure, see Appendix A.1). This corresponds to physical dimensions of approximately $r\in\left[0.6\,\mathrm{m}, 5\times 10^7\,\mathrm{m}\right]$ and a step size of $\Delta r = 25\,\mathrm{m}$, which makes sure we do not reach the end of the range before the end of the star.

For the central density ρ_{c} we are tasked to study the range $10^6 < \rho_{\mathrm{c}} < 5\times 10^{14}\,\mathrm{kg\,m^{-3}}$, so we set up to test using the following values: $\rho_{\mathrm{c}}\in\left\{a\times 10^b \mid a=1,2,\dots,9; b=6,7,\dots,14\right\}$. The results are presented in Figures 3, 4, and 5. We have chosen logarithmic plots for the first two plots to better highlight the similar behaviour at low densities and the different behaviour at high densities.

In Figure 3 and 4 we can see that at central densities $\rho_{\mathrm{c}}\approx 10^8$ the results for radius and mass start to deviate. The non-relativistic values follow an exponential relationship, as one might have predicted from (14) and (5). The relativistic case, on the other hand, exhibits more interesting behaviour. At higher densities, the mass tends to a single maximum value — the *Chandrasekhar mass*, which can be seen in Figures 4 and 5. This implies that the theoretical model of a white dwarf cannot properly describe objects with higher mass.

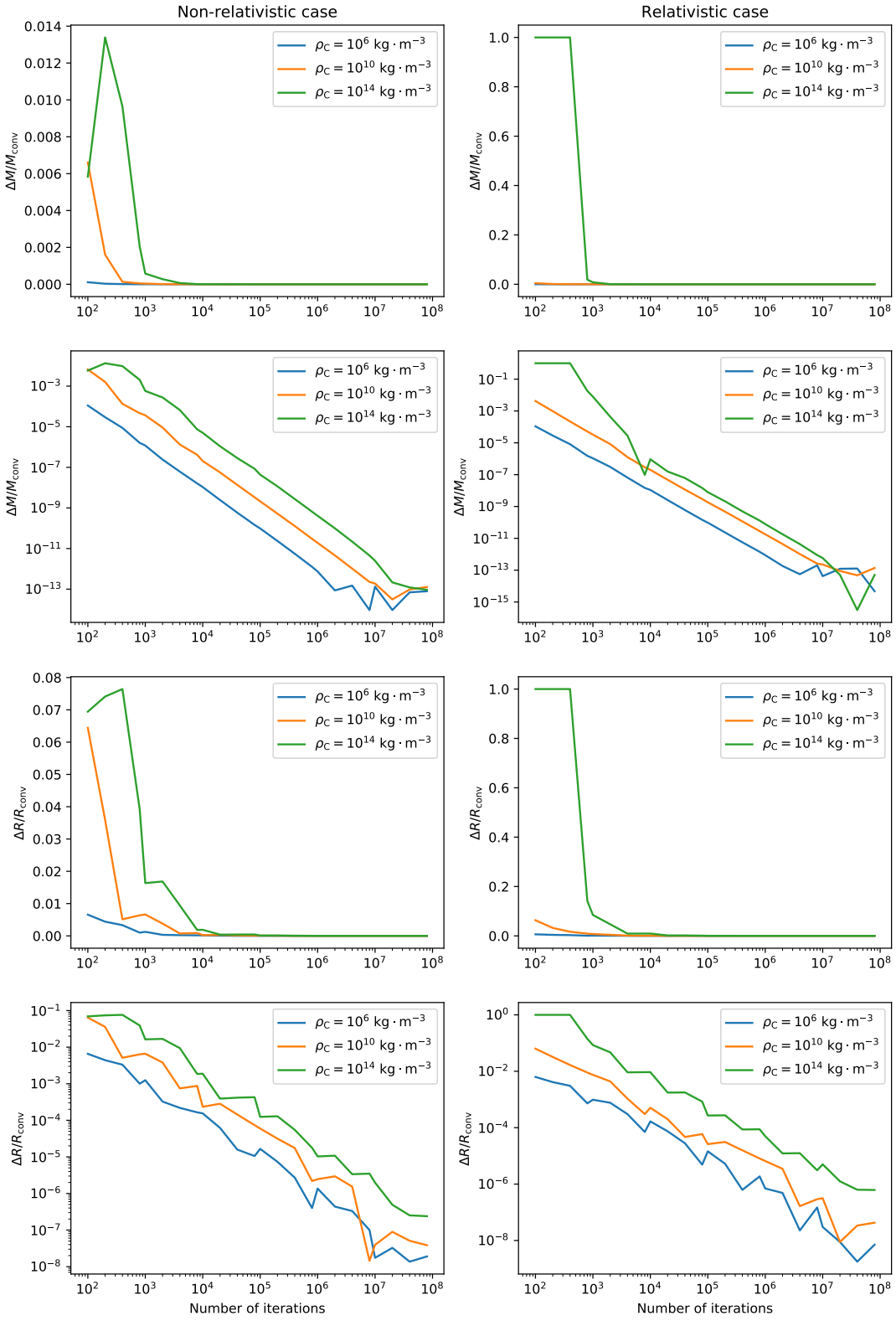


Figure 2: Plot of convergence for three central spanning the tested range of ρ_c in both non-relativistic (left column) and relativistic (right column) cases. As can be seen, convergence is slower in the relativistic case; after 10^6 intervals good accuracy is achieved in both cases for both mass and radius values. The values $M_{\text{conv}}, R_{\text{conv}}$ are computed as the average of the last three values; $\Delta M = |M - M_{\text{conv}}|$ and $\Delta R = |R - R_{\text{conv}}|$. Odd rows have a linear y-axis, even rows are the same plot with a logarithmic y-axis.

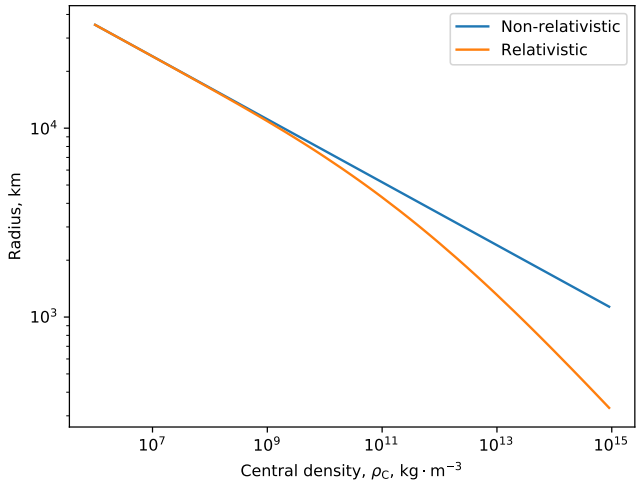


Figure 3: Radius of a white dwarf R in km as a function of central density ρ_c.

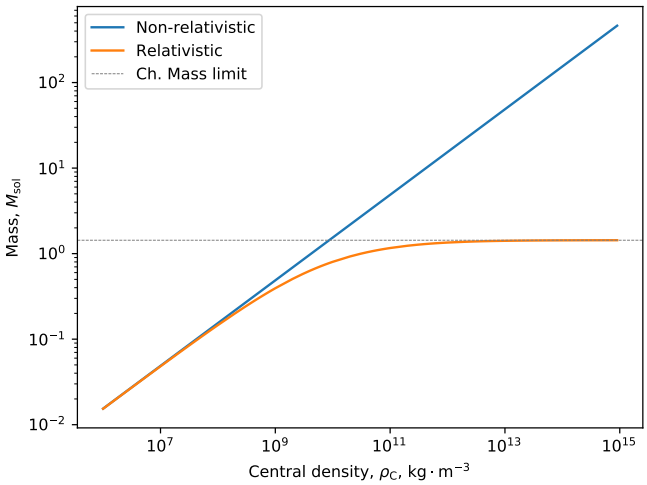


Figure 4: Mass of a white dwarf star M in solar masses as a function of central densityρ_c. Note the Chandrasekhar mass limit.

4.1 Compact objects

White dwarfs are part of the more general framework of compact objects as end stages in stellar evolution, which also include neutron stars and black holes. These objects are the final result of stellar evolution. In which one a star settles is almost completely determined by its initial mass during the stable stage on the Main sequence.

Low to medium mass stars ($0.6M_\odot \lesssim M \lesssim 8M_\odot$) usually evolve by hydrogen burning joined by helium burning in the red giant stage. Since their carbon cores are not massive enough to ignite, the stars contract again and the strong stellar wind expels all surrounding gas to form a planetary nebula with a hot central star, which in turn cools down to a white dwarf. These white dwarfs are usually carbon and oxygen-rich.

High mass stars ($M \gtrsim 8M_\odot$) have massive enough cores to ignite further production of heavier elements via the alpha and triple alpha processes. After most of the fuel is burned through and a dense iron core has formed, they undergo a similar process of implosion in which the core rapidly compactifies (in some edge cases about $8M_\odot$ only partial fusion of carbon occurs, they settle to a white dwarf state). In this case, however, the core mass has passed the (effective) Chandrasekhar limit and electron degeneracy pressure cannot support the equilibrium. Thus, the core collapses further to a neutron star or a black hole (the exact mass threshold for neutron stars is estimated to be between $2M_\odot$ and $3M_\odot$).

It is the discovery of this limit in Chandrasekhar (1931), however, that started the whole further theoretical study

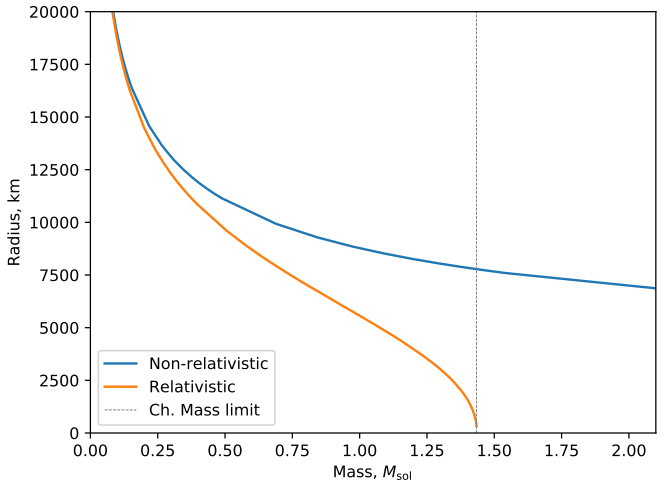


Figure 5: Radius of a white dwarf R in km as a function of the mass M in solar masses. Note the Chandrasekhar mass limit.

of the topic of compact objects. In the following year, the neutron was discovered, and Landau published a paper that discussed a more precise calculation of the white dwarf mass limit and pondered the nature of heavier stars. This laid the groundwork for further discoveries in the 20th century.

4.2 Binary systems

Additionally, white dwarfs are often part of binary or multiple systems (about a third of all star systems are multiple systems). In the case the white dwarf is part of a very close binary, there is an accretion of mass to the surface of the white dwarf to a thin dense atmosphere layer atop the surface of the star. This atmosphere, usually comprised of hydrogen, is heated up by the star, which causes the ignition of runaway fusion. This is observed as a sudden flash of light - a *nova*.

If there is accretion from a more massive companion to a carbon-oxygen white dwarf (heavier elements support further collapse since more energy is needed for their ignition), this can cause its mass to increase beyond the Chandrasekhar limit and ignite carbon fusion in the centre of the white dwarf due to compressional heating. This results in the destruction of the star and is believed to be the physical model behind Type Ia *supernovae*.

These two phenomena further highlights the critical mass limit of existence for white dwarfs. Without it, the above would not be observed in nature.

5 Summary

In this practical, we study the structure of white dwarf stars by numerically solving the equations of equilibrium. For a white dwarf star, the equilibrium is supported by the electron degeneracy pressure. We derive these equations in Section 2, differentiating between the cases of non-relativistic and relativistic electrons. Additionally, we consider the ultrarelativistic limit and anticipate the result of calculations — a limit to white dwarf mass in the case of relativistic electrons.

We choose to solve these equations as an initial value problem, where the starting point is a small region at the centre of the star. In Section 3, we compare the Euler, Heun, and Runge-Kutta 4th order (RK4) methods, which resulted in our choice to use the RK4 for solving the white dwarf equations. Then we introduce a transformation of variables to dimensionless quantities, after which we test the convergence of white dwarf mass and radius for a sample of central density values. We conclude that when the tested interval for radial distance in dimensionless units is $\xi \in \left[1, 8.1 \times 10^7\right]$ with 2×10^6 intervals we have a good compromise between accuracy and speed (this corresponds to physical dimensions of approximately $r \in \left[0.6\,\mathrm{m}, 5 \times 10^7\,\mathrm{m}\right]$ and a step size of $\Delta r = 25\,\mathrm{m}$).

We present the solution to the white dwarf equation in Section 4. We note the difference of solutions between the non-relativistic and relativistic equation of state as the central density grows, seen in Figures 3 and 4. There we also discuss the implications of the existence of the *Chandrasekhar limit* and the place of white dwarfs in the evolutionary track of stars as an end stage of medium mass stars $M \lesssim 8M_\odot$, compared to other compact objects (neutron stars

and black holes). Finally, we note some other astrophysical phenomena which occur due to the special properties of white dwarfs — novae and supernovae Type Ia.

While this project takes steps towards a numerical solution of white dwarf equations, we have made some major assumptions along the way with regards to the physics. Among the effects to be taken into account in a further study are the following:

- The interaction of electrons with their surroundings: we can introduce magnetic fields and the electrostatic interaction in the force balance equations, which modifies the white dwarf equation.
- Chemical composition: since stars form an onion-like structure of elements in their cores, the mean molecular weight per electron μ_e varies slightly, which changes the constant factors in the equation for the white dwarf.
- Effects of General Relativity should also be taken into account. White dwarfs are compact objects, so as the density grows larger, the effects are more and more significant.
- We assumed that the star is in a complete stationary equilibrium state. However, we also have to consider the cases where the star might be rotating (a compact object can have a significant rotation speed due to conservation of angular momentum) or pulsating, which can disturb the conditions in the centre.

An introductory account of some of these effects can be found in e.g. Sagert et al. (2005), while a more standard and in-depth textbook exposition to the topic can be found in Shapiro and Teukolsky (1991).

Additionally, there is room for improvement in the computational methods employed as well. The fairly simple RK4 method can be modified to have a varying step size that self-corrects depending on the derivative at a given point. That way the calculations can skip over a region with small a small gradient by increasing the step size, and decrease the step size to get a more accurate value for the change in a region with a big gradient. Another direction of improvement would be to change from RK4 to another, a more sophisticated method that has better accuracy and faster computational time like the Bulirsch-Stoer method. This would be especially helpful if the system of differential equations to be solved was much more complex in its behaviour. A good introductory text on the subject is Press et al. (2007), Chapter 17.

References

Chandrasekhar, S. (1931), 'The maximum mass of ideal white dwarfs', *The Astrophysical Journal* **74**, 81.

Chandrasekhar, S. (1984), 'On stars, their evolution and their stability', *Reviews of Modern Physics* **56**(2), 137–147.

Oxford Physics (2020), *CO31: Structure of white dwarf stars*, Oxford Physics.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2007), *Numerical recipes : the art of scientific computing*, third edition. edn, Cambridge University Press, Cambridge.

Sagert, I., Hempel, M., Greiner, C. and Schaffner-Bielich, J. (2005), 'Compact stars for undergraduates', *Eur.J.Phys.27:577-610,2006* .

Shapiro, S. L. and Teukolsky, S. A. (1991), *Black Holes, White Dwarfs and Neutron: the physics of compact objects*, VCH PUBN.

A Source code

The structure of the code has been guided by the guidance set in Oxford Physics (2020). Additionally, Press et al. (2007) has had some influence on general code structure. The documentation for the main ODE solver is in the Python code. A copy in an online repository can be found here.

A.1 Main ODE facilities

```
"""
Implementation of a generic ODE integrator.
"""

import numpy as np
from . import integrators

class ODEinit(object):
    """
    Base class for a general ODE to be numerically integrated. It provides
    facilities to compute a value of some linear system given the ODE form as
    `y' = f(x, y)` for some interval of values of `x` and initial conditions of
    the vector `y`.

    Parameters
    ----------
    y0 : np.ndarray
        Initial conditions of the vector `y` at position `x = span[0]`
    deriv
        The function `f` in a functional form. Should accept two parameters at
        which to compute the derivative:
            x: float, the value of the independent variable;
            y: np.ndarray, the value of the dependent variable vector `y`.
    span : np.ndarray
        The interval values over which to compute the value of the vector `y`.

    Attributes
    ----------
    y0 : np.ndarray
        The initial state of the system. Passed at initialization.
    deriv
        The function `f` in a functional form. Passed at initialization.
    span : np.ndarray
        The interval of values to be integrated over. Passed at initialization.
        **Assumed to be equidistant!!**
    interval : float
        The size of the interval step; equal to `span[1] - span[0]`.
    yOut : np.ndarray
        The computed values of `y` after integration. Initialized as a matrix
        of zeros; if integration terminates before the end of `span`, the rest
        of the points are left as zeros.
    flagIntegrated : bool
        A flag to indicate whether the integration has been done (and `yOut`
        filled) to be used by methods depending on having computed values.

    Methods
    -------
    integrate
        Integrates the differential equation.
```

```
    """

    def __init__(self, y0: np.ndarray, deriv, span: np.ndarray) -> None:
        self.y0 = y0
        self.deriv = deriv
        self.interval = span[1] - span[0]
        self.span = span

        # size of this is number of values to be computed x dimensions of y
        self.yOut = np.zeros((self.span.size, self.y0.size))

        # a flag taking note if the equation has been integrated
        self.flagIntegrated = False

    def integrate(self, integrator: int=3) -> None:
        """
        Integrates the equation using the function of the derivative `deriv`.
        User has the option to choose an integrator from the library, the
        default is a RK4 integrator. As it computes, it fills the `yOut` with
        the computed values. Can terminate prematurely if the value passed to
        `deriv` is inappropriate; then the rest of the values are left as zero.

        Parameters
        ----------
        integrator : {1, 2, 3}, optional
            Choice of the integrator to be used given by the number:
                1: the Euler first order method;
                2: the Heun method, a second order method;
                3: the Runge-Kutta 4th order method.

        See Also
        --------
        modules.integrators
        """
        # choose an integrator
        self.integrator = integrators.index[integrator]

        # fill in the initial value of y
        self.yOut[0,:] = self.y0[:]

        #cycle does the integration
        for i in range(self.span.size-1):
            # the try except handles the case where integration is impossible,
            # i.e. when the value of an element of y is nonphysical. deriv
            # decides when this is the case.
            try:
                # calculate the new value
                adder = self.integrator(
                    self.span[i], self.yOut[i,:], self.deriv, self.interval)
                # append new value
                self.yOut[i+1,:] = adder
            except ValueError:
                # ends the integration if it is impossible to integrate - Value
                # Error concerns the y-values
                break

        # to assert that the ODE has been integrated in functions that use yOut.
```

```
        self.flagIntegrated = True

if __name__=="__main__":
    print(
        "This file contains declaration of a generic ODE class.")
```

```
"""
White dwarf class wrapper.
"""
import numpy as np
from scipy import constants as cs  # constants for the derivative equation

from . import derivatives, ode

class whiteDwarf(ode.ODEinit):
    """
    Provides a wrapper of the ODEinit class in the case of white dwarfs with
    some equation of state. Initializes an ODE with appropriate initial
    conditions while taking into account a normalization which leaves the ODE
    working with dimensionless quantities.

    Parameters
    ----------
    rhoC : float
        The density of matter in the small region around the center. In units
        of kg * m^(-3)
    span : np.ndarray
        The range of values of r to be calculated at. In normalized units to
        cut down on computational time and improve calculation accuracy.
        Expects an array with first member = 1 for the starting central mass to
        be correctly defined. Unit of normalization given by l (see below).
    regime : {1, 2, 3}, optional
        The regime indicates the choice of equation of state -> derivative, by
        default 1 - non-relativistic.

    Attributes
    ----------
    rhoC : float
        The density of matter in the small region around the center
    const1 : float
        Constant to be used in the derivative, depends on the `rhoC` value.
        Value after the normalization.
    const2 : float
        Second constant to be used in the derivative for the relativistic case,
        depends on `rhoC`. Value after the normalization.
    Radius : float
        Physical radius of the star. Has a value only after `getRadius()` has
        been called.
    Mass : float
        Mass of the star. Has a value only after `getRadius()` has been called.

    Methods
    -------
    getRadius
        Returns the tuple `(Radius, Mass)` of the star if the equation has been
        integrated.

```

```
    Background
    ----------
    The differential equation is a coupled system of the density `rho` and mass
    `m` as functions of the distance from the centre `r`. These have some
    dimensions in powers of kg and m. We transform the variables with some
    scaling constants so that the ODE solver works with dimensionless
    variables. The following choice of constants is made:
        rho = rhoC * theta,
        m = rhoC * mu,
        r = l * ksi,
    where rhoC is given as input by the user and `l^3*4*pi/3 = 1`. The choice
    of l gives it a value on the order of 1, and we will choose it as our step
    size in r i.e. the step size in ksi is just 1. This is a convenient choice
    for normalization.
    """

    def __init__(self, rhoC: float, span: np.ndarray, regime: int=1) -> None:
        self.rhoC = rhoC
        # vector to be given as initial conditions, [rho, mass]
        init_condit = np.array([1.0, 1.0*(span[0]**3)])

        # constants in the derivative; computed here so that they are computed
        # only once rather than every time the derivative func is called
        self.const1 = -(2**(13.0/3))*cs.pi*cs.gravitational_constant *\
            cs.electron_mass*(cs.proton_mass**(5.0/3)) * \
            (rhoC**(1.0/3))*(cs.h**(-2.0))
        self.const2 = (3*rhoC/16/cs.pi/cs.proton_mass *
                       (cs.h/cs.c/cs.electron_mass)**3)**(2.0/3)

        # choose the equation of state and pass constants, prepares a function
        # to be passed to the constructor of ODEinit accepting (x, y) params
        if regime == 1:
            func = lambda a, b: derivatives.nonRelativGas(a, b, self.const1)
        elif regime == 2:
            func = lambda a, b: derivatives.relativGas(a, b, self.const1, self.const2)
        elif regime == 3:
            func = lambda a, b: derivatives.ultraRel(a, b, self.const1)

        super(whiteDwarf, self).__init__(init_condit, func, span)

    def getRadiusMass(self) -> tuple:
        """
        Finds the radius and mass of the white dwarf, and returns them. Takes
        into account the normalization of the ODE.

        The function scans `yOut[:,0]` column containing `rho(r)` values. We
        know that this is a decreasing function; the first instance where it
        has a value <= 0 it where the star "ends".

        Returns
        -------
        tuple
            Returns a tuple `(Radius, Mass)`. Radius and Mass are calculated
            from `yOut` with the appropriate normalization constants. For
            Radius it is `l = (3/(4*pi))^(1/3)`, result is in m. For Mass it is
            `rhoC`, result is in kg.
        """
```

```
109
110         # find index of rho = 0 (changing signs)
111         if self.flagIntegrated:
112             index = np.searchsorted(-self.yOut[:, 0], 0)
113             # index is the position where we have yOut[index,:] ~ [0,0]
114
115             l = (3/(4*np.pi))**(1.0/3.0)                # normalization const
116             self.Radius = (index-1)*self.interval*l     # in m
117             self.Mass = self.yOut[index-1, 1]*self.rhoC # in kg*m^(-3)
118             return self.Radius, self.Mass
119         else:
120             print("Integrate the eqn first")
121             return None, None
122
123 if __name__=="__main__":
124     print(
125         "This file contains declaration of a white dwarf class.")
```

```
1  """
2  Integrators Library(`integrators.py`)
3  ====================================
4
5  This is a home to the different integrators used alongside the target of the
6  exercise, namely Runge-Kutta 4th order (rk4) method. We implement two other
7  methods to evaluate the advantages of the rk4 method, both in its accuracy and
8  precision.
9
10 Integration Methods implemented
11 -------------------
12
13     euler   Euler method - 1st order.
14     heun    Heun method - a variant of a 2nd order.
15     rk4     Runge-Kutta 4th order method.
16
17 Implementation notes
18 --------------------
19 All of the functions defined are to act on np.ndarray structures, therefore
20 should be element-wise functions; all of the magic is done for us by numpy.
21 The differential equation which they iterate is set up as `dy/dx = f(x, y)`,
22 where `y` and `f` are N-dimensional vectors. Hence, all of the functions
23 receive four parameters:
24
25     x : float, the point at which we start computing
26     y : array_like, the values of the y-vector at x
27     deriv : function, the derivative of y to be computed at some (x, y) coords
28     h : float, the interval size
29
30 All of the functions return the value of `y` for the next place in the
31 integration range. How we determine its value, or more precisely the increment
32 from the previous value, is the difference between these methods.
33
34 Error handling is implemented in `rk4` (the one used for actual calculations),
35 where evaluating the `deriv` function might not be possible.
36
37 Choice of an integrator is aided by the dictionary `index`, mapping integer
38 values to the integrator functions.
39 """
```

```python

import numpy as np

def euler(x: float, y: np.ndarray, deriv, h: float) -> np.ndarray:
    """
    Compute an iteration using Euler's method: a 1st order method.

    This function computes an iteration in the numerical integration of an ODE
    of the form `dy/dx = f(x, y)` using the first order Euler method:
    `y(x + h) = y(x) + h*f(x, y(x))`, where h is the size of the step in the
    x-coordinate.

    Parameters
    ----------
    x : float
        Starting coordinate in x.
    y : ndarray
        Starting coordinate in y. In general can be an N-dimensional array.
    deriv : function
        A function which computes the derivative of the y-vector given the
        x and y coords.
    h : float
        The step size in x.

    Returns
    -------
    ndarray
        The returned value is the approximation of y(x + h) using the Euler
        algoritm.
    """
    return y + h*deriv(x,y)

def heun(x: float, y: np.ndarray, deriv, h: float) -> np.ndarray:
    """
    Compute an iteration using Heun's method, a kind of 2nd order method.

    This function computes an iteration in the numerical integration of an ODE
    of the form `dy/dx = f(x, y)` using the second order Heun method:
    `y(x + h) = y(x) + 0.5*h*(f(x, y(x)) + f(x + h, y + h*f(x, y)))`, where h
    is the size of the step in the x-coordinate. This is a predictor-corrector
    method, and uses the prediction of the slope at `x + h` to correct the
    final calculation of the slope to be added.

    Parameters
    ----------
    x : float
        Starting coordinate in x.
    y : ndarray
        Starting coordinate in y. In general can be an N-dimensional array.
    deriv : function
        A function which computes the derivative of the y-vector given the
        x and y coords.
    h : float
        The step size in x.

    Returns
    -------
```

```
    np.ndarray
        The returned value is the approximation of y(x + h) using the Heun
        algorithm.
    """
    return y + 0.5*h*(deriv(x, y) + deriv(x + h, y + h*deriv(x,y)))

def rk4(x: float, y: np.ndarray, deriv, h: float) -> np.ndarray:
    """
    Compute an iteration using a 4th order Runge-Kutta method.

    This function computes an iteration in the numerical integration of an ODE
    of the form `dy/dx = f(x, y)` using a fourth order Runge-Kutta method.

    Parameters
    ----------
    x : float
        Starting coordinate in x.
    y : ndarray
        Starting coordinate in y. In general can be an N-dimensional array.
    deriv : function
        A function which computes the derivative of the y-vector given the
        x and y coords.
    h : float
        The step size in x.

    Returns
    -------
    np.ndarray
        The returned value is the approximation of y(x + h) using the 4th order
        Runge-Kutta algorithm.

    Raises
    ------
    ValueError
        If an error is raised by the deriv function indicating impossible
        evaluation, raise an error to indicate the termination of integration.

    Notes
    -----
    Note that in the case deriv cannot be computed for any of the steps, the
    error is handled by raising a value error back to the integrator method of
    the ODE class where this is handled. Additionally, an improvement in the
    calculation order is implemented as per [1]_, where calculation cycle
    number is reduced, as well as memory usage.

    Sources
    -------
    ..[1] Press, William H., Saul A. Teukolsky, William T. Vetterling, and
    Brian P. Flannery. "Numerical Recipes : The Art of Scientific Computing."
    Third ed. Cambridge, 2007.
    """
    # TODO think of fix
    try:
        # USE OLD implementation, uses more cycles and memory (allegedly)
        k1 = h*deriv(x,y)
        k2 = h*deriv(x + 0.5*h, y + 0.5*k1)
        k3 = h*deriv(x + 0.5*h, y + 0.5*k2)
```

```
        k4 = h*deriv(x + h, y + k3)

        # 'FASTER' implementation - produced an error while running tests on
        # the SHO for the report, revert to the more abstract version - some
        # cycles but sure it is correct

        # hh = 0.5*h
        # h6 = h/6.0
        # xh = x + hh
        # k1 = deriv(x, y)
        # yTemp = y + hh*k1
        # k2 = deriv(xh, yTemp)
        # yTemp = y + hh*k2
        # k3 = deriv(xh, yTemp)
        # yTemp = y + k3
        # k3 += k2
        # k2 = deriv(x+h, yTemp)
    except ValueError:
        raise ValueError("Invalid integration, terminate integration!")

    return y + (k1 + k4)/6.0 + (k2 + k3)/3.0
    # return y + h6*(k1 + k2 + 2*k3)

# dictionary to choose integrator; for neater code in modules.ode
index = {1: euler, 2: heun, 3: rk4}

if __name__=="__main__":
    print(
        "This is a module home to several integrators to be used in solving \
ODEs: Euler, Heun, Runge-Kutta4.")
```

```
"""
Derivatives Library(`derivatives.py`)
=====================================

This is a home to several functions which give the derivative `f` in a system
of linear ODEs `y'(x) = f(x, y)` in functional form.

White Dwarf Functions
---------------------
nonRelativGas       Implements the eqn. of state for non-relativistic electrons
relaitvGas          Implements the eqn. of state for relativistic electrons.

Testing Functions
-----------------
SHO                 Implements the Simple Harmonic Oscillator.

Implementation Notes
--------------------
Choice of an equation of state for the white dwarf is aided by the dictionary
`index`, mapping integer values to the integrator functions.
"""

import numpy as np

def SHO(x: float, y: np.ndarray) -> np.ndarray:
    """
```

```
    Function of the derivative for the simple harmonic oscillator written as a
    system of linear ODEs.

    Parameters
    ----------
    x : float
        Value of the independent variable in the ODE.
    y : np.ndarray
        Value of the dependent variable in the ODE. Two element vector
        `[z', z]`, where the SHO ODE is `z''(x) + z(x) = 0`.

    Returns
    -------
    np.ndarray
        The value of the derivative at (x, y)

    Notes
    -----
    The equation `z''(x) + z(x) = 0` is written using `p = z'` in the form:
    `y' = [p, z]' = [-z, p]`. The function returns the latter vector.
    """
    return np.array([-4*np.pi*np.pi*y[1], y[0]])

def nonRelativGas(x: float, y: np.ndarray, const: float) -> np.ndarray:
    """
    Function for the coupled ODE using a non-relativistic equation of state.

    Function of the derivative for the system of coupled equations
    `y = [rho, m](x)` in the non-relativistic case. An additional argument
    `const` is passed to save computational time - it is calculated only once
    in the ODE definition, and called as value to be used in the computations.

    Parameters
    ----------
    x : float
        Value of the independent variable in the ODE.
    y : np.ndarray
        Value of the dependent variable in the ODE. *Two elements only.*
    const: float
        The scaling constant used in the definition.

    Returns
    -------
    np.ndarray
        The value of the derivative at (x, y). Look at Notes for info on the
        functional form.

    Raises
    ------
    ValueError
        If the value `y[0]` < 0, the computation is discarded on physical
        grounds - cannot have negative matter density. Raises an exception
        propagated by the integrator to be handled in the .integration method.

    Notes
    -----
    The functional form is determined by the equation of state in the
```

```
    non-relativistic case, where `pressure = someConst * rho^(2/3)`.The final
    form: `[rho, m]' = [const1*m*rho^(1/3)/x^2, 3*x^2*rho]`.
    """

    # need y[0] >= 0 to be exponentiated apart from physical grounds
    # this catches the moment y[0] becomes too small - only it is on the
    # computation of the following value
    if y[0] < 1e-10:
        raise ValueError
    return  np.array([const*y[1]*(y[0]**(1.0/3))*(x**(-2.0)),
                      3*(x**2)*y[0]])

def relativGas(x: float, y: np.ndarray, const1: float ,const2: float) -> np.ndarray:
    """
    Function for the coupled ODE using a relativistic equation of state.

    Function of the derivative for the system of coupled equations
    `y = [rho, m](x)` in the relativistic case. Two additional arguments
    `const1` and `const2` are passed to save on computational time - they are
    calculated only once in the ODE definition, and called as value to be used
    in the computations.

    Parameters
    ----------
    x : float
        Value of the independent variable in the ODE.
    y : np.ndarray
        Value of the dependent variable in the ODE. *Two elements only.*
    const1: float
        One of the scaling constants, used in the derivative of rho.
    const2 : float
        The second scaling constant, used in the relativistic correction.

    Returns
    -------
    np.ndarray
        The value of the derivative at (x, y). Look at Notes for info on the
        functional form.

    Raises
    ------
    ValueError
        If the value `y[0]` < 0, the computation is discarded on physical
        grounds - cannot have negative matter density. Raises an exception
        propagated by the integrator to be handled in the .integration method.

    Notes
    -----
    The functional form is determined by the equation of state in the
    relativistic case, where `pressure = someConst*rho^(2/3)*relCorr`.The
    relativistic correction is `relCorr = sqrt(1+const2*rho^(2/3))^(-1)`.
    The final form: `[rho, m]' = [const1*m*rho^(1/3)*relCorr/x^2, 3*x^2*rho]`.
    """

    # need y[0] >= 0 to be exponentiated apart from physical grounds
    # this catches the moment y[0] becomes too small - only it is on the
    # computation of the following value
```

```
    if y[0] < 1e-10:
        raise ValueError

    return np.array([const1*y[1]*(y[0]**(1.0/3))*(x**(-2.0))*
                        (1+const2*(y[0]**(2.0/3)))**(0.5), 3*(x**2)*y[0]])

index = {1: nonRelativGas, 2: relativGas}

if __name__=="__main__":
    print(
        "This is a home to several functions which give the derivative in a \
system of linear ODEs.")
```

A.2 Code to run the experiment and testing procedures

```
"""main.py
The main script in this experiment. It calculates the values of the white dwarf
radius and mass for different central densities (initial conditions to solve
the ODE) by using our purpose built library and outputs them to a .csv file.
"""

# parallel computing library
from multiprocessing import Pool, cpu_count

import numpy as np
# for saving files to csv (easier than the numpy implementation)
import pandas as pd
# solar mass for output scale
from astropy.constants import M_sun
# for setup of integration grid
from numpy.core.function_base import linspace

# our white dwarf integrator module
import modules


# Initialize a white dwarf with non-relativistic equation of state
def starInitNonRelativ(rhoC):
    """This function initializes a white dwarf with a non-relativistic equation
    of state and calculates its radius and mass, then returns the values.

    Parameters
    ----------
    rhoC : float
        The density of the center in kg*m^(-3), used to set the initial
        conditions for integration.

    Returns
    -------
    list
        Returns a three element list of the density in in kg*m^(-3), the radius
        in km/1000, and the mass in solar mass in this order.
    """
    # Integration grid in normalized units, optimal step size determined on
    # convergence ground for more info look at convergence.py
    span = linspace(1,8.1e7,num=2000000)
```

```
    # Initialize the white dwarf and integrate
    star = modules.whiteDwarf(rhoC, span, 1)
    star.integrate(3)

    radius, mass = star.getRadiusMass()
    # return the three values rho in kg*m^(-3), radius in km/1000, mass in solar mass
    return [rhoC, radius/1000, mass/M_sun.value]

# Initialize a white dwarf with relativistic equation of state
def starInitRelativ(rhoC):
    """This function initializes a white dwarf with a relativistic equation of
    state and calculates its radius and mass, then returns the values.

    Parameters
    ----------
    rhoC : float
        The density of the center in kg*m^(-3), used to set the initial
        conditions for integration.

    Returns
    -------
    list
        Returns a three element list of the density in in kg*m^(-3), the radius
        in km/1000, and the mass in solar mass in this order.
    """
    # Integration grid, optimal step size determined on convergence ground
    # for more info look at convergence.py
    span = linspace(1,8.1e7,num=2000000)
    # Initialize the white dwarf and integrate
    star = modules.whiteDwarf(rhoC, span, 2)
    star.integrate(3)

    radius, mass = star.getRadiusMass()
    # return the three values rho in kg*m^(-3), radius in km/1000, mass in solar mass
    return [rhoC, radius/1000, mass/M_sun.value]


def main():
    # the rho values to test in the range, change the list accordingly
    rhoVal = np.array([float(a)*10**b for b in range(6,15) for a in range(1,10)])

    # Implements parallel computation of the ODEs using multiple processes.
    # Computation is done over the range of central density values rhoVal.

    # Computation for non-relativistic gas. This process sets up the
    # multiprocessing and cleans up after the calculations are over.
    with Pool(cpu_count(), maxtasksperchild=1) as executor:
        # The executor.map(f, s) applies the function f to each member of the
        # iterable s and collects the results in a list following a First In
        # First Out procedure, so the results collected are ordered by the
        # order in which they were in s, i.e. by central density values.
        resNonRelativ = executor.map(starInitNonRelativ, rhoVal)

    # Save results for non-relativistic case
    dfNonRel = pd.DataFrame(resNonRelativ, columns=["rhoC", "radiusKM", "massSolMass"])
    dfNonRel.to_csv("nonRelativRes.csv", index=None)

```

```
    # Computation for non-relativistic gas. This process sets up the
    # multiprocessing and cleans up after the calculations are over.
    with Pool(cpu_count()-6, maxtasksperchild=1) as executor:
        # The executor.map(f, s) applies the function f to each member of the
        # iterable s and collects the results in a list following a First In
        # First Out procedure, so the results collected are ordered by the
        # order in which they were in s, i.e. by central density values.
        resRelativ = executor.map(starInitRelativ, rhoVal)

    # Save results for relativistic case
    dfRel = pd.DataFrame(resRelativ, columns=["rhoC", "radiusKM", "massSolMass"])
    dfRel.to_csv("relativRes.csv", index=None)


if __name__ == "__main__":
    main()
```

```
"""convergence.py
This is an auxiliary script, which helps determine the convergence of
computing white dwarf parameters and in turn judge the needed coarseness of
the integration space. Many of the functions here are direct copies of the ones
in main.py; look there for more explanation. Outputs .csv files of calculations.
"""
import math
from multiprocessing import Pool, cpu_count

# for saving files to csv (easier than the numpy implementation)
import pandas as pd
# solar mass for output scale
from astropy.constants import M_sun
# for setup of integration grid
from numpy.core.function_base import linspace

# our library
import modules


# Direct copy of the method in main.py, only added a variable for the number of
# intervals in the space (since we are probing it).
def starInitNonRelativ(myTuple):
    rhoC, n = myTuple
    span = linspace(1,8.1e7,num=n)
    star = modules.whiteDwarf(rhoC, span, 1)
    star.integrate(3)

    radius, mass = star.getRadiusMass()
    # return the three values No of iter, radius in km/1000, mass in solar mass
    return [n, radius/1000, mass/M_sun.value]

# Direct copy of the method in main.py, only added a variable for the number of
# intervals in the space (since we are probing it).
def starInitRelativ(myTuple):
    rhoC, n = myTuple
    span = linspace(1,8.1e7,num=n)
    star = modules.whiteDwarf(rhoC, span, 2)
    star.integrate(3)

```

```
    radius, mass = star.getRadiusMass()
    # return the three values No of iter, radius in km/1000, mass in solar mass
    return [n, radius/1000, mass/M_sun.value]

numberOfIterations = [int(float(a)*10**b) for b in range(2,8) for a in [1,2,4,8]]

# wrapped all actions in a main function because of the multiprocessing
def main():
    # Loop the three densities - two at the ends of the interval and one at the middle
    # multiprocessing copied from main.py
    for density in [1e6, 1e10, 1e14]:
        # quick hack to prepare an iterable for the multiprocessing pool
        # function, since our function needs two parameters
        poolIterable = [(density, n) for n in numberOfIterations]
        with Pool(cpu_count(), maxtasksperchild=1) as executor:
            # The executor.map(f, s) applies the function f to each member of the
            # iterable s and collects the results in a list following a First In
            # First Out procedure, so the results collected are ordered by the
            # order in which they were in s, i.e. by central density values.
            resNonRelativ = executor.map(starInitNonRelativ, poolIterable)

        # Save results for non-relativistic case
        dfNonRel = pd.DataFrame(resNonRelativ, columns=["NoIter", "radiusKM", "massSolMass"])
        dfNonRel.to_csv(
            "convergenceE{}nonRelativ.csv".format(int(math.log10(density))),
            index=None
        )

        with Pool(cpu_count()-6, maxtasksperchild=1) as executor:
            # The executor.map(f, s) applies the function f to each member of the
            # iterable s and collects the results in a list following a First In
            # First Out procedure, so the results collected are ordered by the
            # order in which they were in s, i.e. by central density values.
            resRelativ = executor.map(starInitRelativ, poolIterable)

        # Record results to csv file
        dfRelativ = pd.DataFrame(resRelativ, columns=["NoIter", "radiusKM", "massSolMass"])
        dfRelativ.to_csv(
            "convergenceE{}Relativ.csv".format(int(math.log10(density))),
            index=None
        )


if __name__ == '__main__':
    main()
```

```
# Test the error with the integrators and plot residuals
# NOTE we need some amount of iterations comparable to the white dwarf to
# compate the methods i.e. 1e6
from math import cos, sin

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from numpy.core.function_base import linspace

```

```
12 import modules
13 from modules.derivatives import SHO
14
15
16 def plots():
17     # load data
18     eul1= pd.read_csv("method1.csv")
19     heun= pd.read_csv("method2.csv")
20     rk4= pd.read_csv("method3.csv")
21     eul2= pd.read_csv("method4.csv")
22
23     # theoretical data
24     x0 = 1.0
25     step = 0.001
26     span = np.array(step*linspace(1,1000000,num=1000000))
27     # for eul1, heun, rk4
28     theory = np.concatenate(([1], x0*np.cos(2*np.pi*span)))
29
30     # 2*energy in systems/m
31     c = 4*np.pi*np.pi
32     enEul1 = eul1["xdot"]**2 + c*eul1["x"]**2
33     enEul2 = eul2["xdot"]**2 + c*eul2["x"]**2
34     enHeun = heun["xdot"]**2 + c*heun["x"]**2
35     enRK4 = rk4["xdot"]**2 + c*rk4["x"]**2
36
37     ##  plot 1 - oscill of eul1, eul2, heun
38     plt.figure(1)
39     plt.plot(eul1.index/1000, eul1["x"])
40     plt.plot(eul2.index/1000, eul2["x"])
41     plt.plot(heun.index/1000, heun["x"])
42     #axis setup
43     plt.xlim(0, 20)
44     plt.xlabel("time t, s")
45     plt.ylim(-2,2)
46     plt.ylabel("position x(t)")
47     plt.legend(["euler1", "euler2", "heun"])
48     plt.show
49     plt.savefig("eul12+heun.pdf")
50
51     ## plot2 - (oscill - theory) of eul1, eul2, heun
52     plt.figure(2)
53     plt.plot(eul1.index/1000, eul1["x"] - theory)
54     plt.plot(eul2.index/1000, eul2["x"] - theory)
55     plt.plot(heun.index/1000, heun["x"] - theory)
56     #axis setup
57     plt.xlim(0, 20)
58     plt.xlabel("time t, s")
59     plt.ylim(-0.5,0.5)
60     plt.ylabel("position x(t) - theory")
61     plt.legend(["euler1", "euler2", "heun"], loc="upper left")
62     plt.show
63     plt.savefig("eul12+heun-theory.pdf")
64
65     ## plot3 - energy of eul1, eul2, heun
66     plt.figure(3)
67     plt.plot(enEul1.index/1000, abs(enEul1-c)/c)
68     plt.plot(enEul2.index/1000, abs(enEul2-c)/c)
```

```
    plt.plot(enHeun.index/1000, abs(enHeun-c)/c)
    #axis setup
    plt.xlim(0, 200)
    plt.xlabel("time t, s")
    plt.ylim(1e-7,100000)
    plt.yscale('log')
    plt.ylabel("energy $\Delta E/E_0$")
    plt.legend(["euler1", "euler2", "heun"], loc="upper left")
    plt.show
    plt.savefig("eul12+heun+energy.pdf")

    ## plot4 - energy of heun, rk4
    plt.figure(4)
    plt.plot(enHeun.index/1000, abs(enHeun-c)/c, color='green')
    plt.plot(enRK4.index/1000, abs(enRK4-c)/c, color='red')
    #axis setup
    plt.xlabel("time t, s")
    plt.ylim(5e-7,1e-3)
    plt.yscale('log')
    plt.ylabel("energy $\Delta E/E_0$")
    plt.legend(["heun", "rk4"], loc="upper left")
    plt.show
    plt.savefig("heunRK4+energy.pdf")


def main():
    # initial parameters of the SHO
    x0 = 1.0
    step = 0.001
    y0 = x0*np.array([-sin(step), cos(step)])
    span = step*linspace(1,1000000,num=1000000)

    # do integration for the three methods
    for integ in [1,2,3]:
        ode = modules.ODEinit(y0, SHO, span)
        ode.integrate(integ)
        res = np.vstack((np.array([0,1]), ode.yOut))
        dfOut = pd.DataFrame(res, columns=["xdot", "x"])
        dfOut.to_csv("method{}.csv".format(integ), index=None)

    # additional integration for the euler method with a step 1/10th of original
    span2 = 0.1*step*linspace(1,10000000,num=10000000)
    ode = modules.ODEinit(y0, SHO, span2)
    ode.integrate(1)
    res = np.vstack((np.array([0,1]), ode.yOut))
    # keep the data only at the same moments as the other ones
    dfOut = pd.DataFrame(res[::10], columns=["xdot", "x"])
    dfOut.to_csv("method4.csv", index=None)

    plots()


if __name__=='__main__':
    main()
```

```
# mainPlots.py
# Plots of data for convergence and white dwarf star parameters.
```

```
from matplotlib import colors
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


def convergencePlots():
    # load in data
    E6nonRelativ = pd.read_csv("convergenceE6nonRelativ.csv")
    E6Relativ = pd.read_csv("convergenceE6Relativ.csv")
    E10nonRelativ = pd.read_csv("convergenceE10nonRelativ.csv")
    E10Relativ = pd.read_csv("convergenceE10Relativ.csv")
    E14nonRelativ = pd.read_csv("convergenceE14nonRelativ.csv")
    E14Relativ = pd.read_csv("convergenceE14Relativ.csv")

    # subplot setup, tighter spacing
    f, axs = plt.subplots(4,2,figsize=(11.2,17.15),constrained_layout=False)

    # mass, non-relativistic
    # values to which result converges
    e6convM = sum(E6nonRelativ["massSolMass"][-3:])/3
    e10convM = sum(E10nonRelativ["massSolMass"][-3:])/3
    e14convM = sum(E14nonRelativ["massSolMass"][-3:])/3

    # plots of delta m / m_conv
    plt.subplot(4,2,1)
    plt.title("Non-relativistic case")
    plt.plot(E6nonRelativ["NoIter"], abs(E6nonRelativ["massSolMass"]-e6convM)/e6convM)
    plt.plot(E10nonRelativ["NoIter"], abs(E10nonRelativ["massSolMass"]-e10convM)/e10convM)
    plt.plot(E14nonRelativ["NoIter"], abs(E14nonRelativ["massSolMass"]-e14convM)/e14convM)
    plt.xscale("log")
    plt.ylabel("$\Delta M/M_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
            loc="upper right")
    # second fig
    plt.subplot(4,2,3)
    plt.plot(E6nonRelativ["NoIter"], abs(E6nonRelativ["massSolMass"]-e6convM)/e6convM)
    plt.plot(E10nonRelativ["NoIter"], abs(E10nonRelativ["massSolMass"]-e10convM)/e10convM)
    plt.plot(E14nonRelativ["NoIter"], abs(E14nonRelativ["massSolMass"]-e14convM)/e14convM)
    plt.xscale("log")
    plt.yscale("log")
    plt.ylabel("$\Delta M/M_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
            loc="upper right")

    # mass, relativistic
    # converged to values
    e6RconvM = sum(E6Relativ["massSolMass"][-3:])/3
    e10RconvM = sum(E10Relativ["massSolMass"][-3:])/3
    e14RconvM = sum(E14Relativ["massSolMass"][-3:])/3

    # plots of delta m / m_conv
    plt.subplot(4,2,2)
```

```
    plt.title("Relativistic case")
    plt.plot(E6Relativ["NoIter"], abs(E6Relativ["massSolMass"]-e6RconvM)/e6RconvM)
    plt.plot(E10Relativ["NoIter"], abs(E10Relativ["massSolMass"]-e10RconvM)/e10RconvM)
    plt.plot(E14Relativ["NoIter"], abs(E14Relativ["massSolMass"]-e14RconvM)/e14RconvM)
    plt.xscale("log")
    plt.ylabel("$\Delta M/M_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
            loc="upper right")
    # log plot
    plt.subplot(4,2,4)
    plt.plot(E6Relativ["NoIter"], abs(E6Relativ["massSolMass"]-e6RconvM)/e6RconvM)
    plt.plot(E10Relativ["NoIter"], abs(E10Relativ["massSolMass"]-e10RconvM)/e10RconvM)
    plt.plot(E14Relativ["NoIter"], abs(E14Relativ["massSolMass"]-e14RconvM)/e14RconvM)
    plt.xscale("log")
    plt.yscale("log")
    plt.ylabel("$\Delta M/M_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
            "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
            loc="upper right")

    # radius, non relativistic
    # converged to values
    e6convR = sum(E6nonRelativ["radiusKM"][-3:])/3
    e10convR = sum(E10nonRelativ["radiusKM"][-3:])/3
    e14convR = sum(E14nonRelativ["radiusKM"][-3:])/3

    # plots of delta m / m_conv
    plt.subplot(425)
    plt.plot(E6nonRelativ["NoIter"], abs(E6nonRelativ["radiusKM"]-e6convR)/e6convR)
    plt.plot(E10nonRelativ["NoIter"], abs(E10nonRelativ["radiusKM"]-e10convR)/e10convR)
    plt.plot(E14nonRelativ["NoIter"], abs(E14nonRelativ["radiusKM"]-e14convR)/e14convR)
    plt.xscale("log")
    plt.ylabel("$\Delta R/R_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
                    loc="upper right")
    # log plot
    plt.subplot(427)
    plt.plot(E6nonRelativ["NoIter"], abs(E6nonRelativ["radiusKM"]-e6convR)/e6convR)
    plt.plot(E10nonRelativ["NoIter"], abs(E10nonRelativ["radiusKM"]-e10convR)/e10convR)
    plt.plot(E14nonRelativ["NoIter"], abs(E14nonRelativ["radiusKM"]-e14convR)/e14convR)
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel("Number of iterations")
    plt.ylabel("$\Delta R/R_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
                    loc="upper right")

    # radius, relativistic
    # converged to values
    e6RconvR = sum(E6Relativ["radiusKM"][-3:])/3
```

```
    e10RconvR = sum(E10Relativ["radiusKM"][-3:])/3
    e14RconvR = sum(E14Relativ["radiusKM"][-3:])/3

    # plots of delta m / m_conv
    plt.subplot(426)
    plt.plot(E6Relativ["NoIter"], abs(E6Relativ["radiusKM"]-e6RconvR)/e6RconvR)
    plt.plot(E10Relativ["NoIter"], abs(E10Relativ["radiusKM"]-e10RconvR)/e10RconvR)
    plt.plot(E14Relativ["NoIter"], abs(E14Relativ["radiusKM"]-e14RconvR)/e14RconvR)
    plt.xscale("log")
    plt.ylabel("$\Delta R/R_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
                    loc="upper right")
    plt.subplot(428)
    plt.plot(E6Relativ["NoIter"], abs(E6Relativ["radiusKM"]-e6RconvR)/e6RconvR)
    plt.plot(E10Relativ["NoIter"], abs(E10Relativ["radiusKM"]-e10RconvR)/e10RconvR)
    plt.plot(E14Relativ["NoIter"], abs(E14Relativ["radiusKM"]-e14RconvR)/e14RconvR)
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel("Number of iterations")
    plt.ylabel("$\Delta R/R_\mathrm{conv}$")
    plt.legend(["$\\rho_\mathrm{C} = 10^6$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{10}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$",
                    "$\\rho_\mathrm{C} = 10^{14}$ $\mathrm{kg} \cdot \mathrm{m}^{-3}$"],
                    loc="upper right")

    # save fig, clip white margins to fit on report page with caption
    plt.savefig("convergencePlot.pdf", bbox_inches="tight")

def whiteDwarfPlots():
    # load data
    nonRres = pd.read_csv("nonRelativRes.csv")
    Rres = pd.read_csv("relativRes.csv")

    # fig 1 radius(rhoc)
    plt.figure()
    plt.plot(nonRres["rhoC"], nonRres["radiusKM"])
    plt.plot(Rres["rhoC"], Rres["radiusKM"])
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel("Central density, $\\rho_\mathrm{C}$, $\mathrm{kg} \cdot \mathrm{m}^{-3}$")
    plt.ylabel("Radius, km")
    plt.legend(["Non-relativistic", "Relativistic"])
    plt.savefig("radius-rhoC.pdf")

    # mass(rhoC)
    plt.figure()
    plt.plot(nonRres["rhoC"], nonRres["massSolMass"])
    plt.plot(Rres["rhoC"], Rres["massSolMass"])
    plt.axhline(1.434, c="grey", ls='--', lw=0.5)
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel("Central density, $\\rho_\mathrm{C}$, $\mathrm{kg} \cdot \mathrm{m}^{-3}$")
    plt.ylabel("Mass, $M_\mathrm{sol}$")
    plt.legend(["Non-relativistic", "Relativistic", "Ch. Mass limit"])
    plt.savefig("mass-rhoC.pdf")
```

```
174
175     # radius(mass)
176     plt.figure()
177     plt.plot(nonRres["massSolMass"], nonRres["radiusKM"])
178     plt.plot(Rres["massSolMass"], Rres["radiusKM"])
179     plt.axvline(1.434, c="grey", ls='--', lw=0.5)
180     plt.xlim(0, 2.1)
181     plt.ylim(0, 20000)
182     plt.xlabel("Mass, $M_\mathrm{sol}$")
183     plt.ylabel("Radius, km")
184     plt.legend(["Non-relativistic", "Relativistic", "Ch. Mass limit"])
185     plt.savefig("radius-mass.pdf")
186
187 def main():
188     convergencePlots()
189     whiteDwarfPlots()
190
191 if __name__ == "__main__":
192     main()
```