AST 304, Fall 2022

Stars

Group Computational Project 2

In the previous project, you wrote routines to integrate a system of ordinary differential equations. Now you are going to use these routines to compute the mass-radius relation for a cold white dwarf star and compare your results against some observational data (Joyce et al. 2018).

1 Background

1.1 White dwarf stars

White dwarfs are the remnants of low-mass ($\lesssim 8\,M_\odot$) stars after fusion reactions have ceased in their cores. The stellar core continues to contract, and, as you showed in class, this results in an increase in the density and pressure. At high densities and pressures, the wavefunctions of the electrons overlap, and this changes the equation of state (EOS). We say the electrons become *degenerate*. When the electrons become degenerate, their pressure increases—the matter becomes "stiffer"—and halts the contraction of the core. The white dwarf continues to radiate, but without gravitational contraction to supply energy, the white dwarf cools and dims.

Once the temperature of the white dwarf is sufficiently low, its equation of state is that of an ideal, degenerate, non-relativistic gas:

$$\begin{aligned} P &= \frac{2}{5} n_e E_\mathrm{F} \\ &= \frac{2}{5} \underbrace{\left(\frac{\rho}{\mu_e m_\mathrm{u}}\right)}_{=n_e} \times \underbrace{\frac{h^2}{2m_e}\left(\frac{3}{8\pi}\frac{\rho}{\mu_e m_\mathrm{u}}\right)^{2/3}}_{E_\mathrm{F}} \\ &= \frac{1}{5}\left(\frac{3}{8\pi}\right)^{2/3} \frac{h^2}{m_e}\left(\frac{\rho}{\mu_e m_\mathrm{u}}\right)^{5/3} . \end{aligned} \tag{1}$$

In this equation, n_e denotes the electron density (number of electrons per cubic meter), E_F denotes the electron Fermi energy, ρ is the mass density, m_u is

the atomic mass unit, and m_e is the electron mass. We introduce the nucleon-to-electron ratio μ_e: for white dwarfs made of carbon and oxygen, $\mu_e = 2$ (for carbon, there are 6 electrons for every 12 nucleons; for oxygen there are 8 electrons for every 16 nucleons). To lowest order we can ignore the contribution of the ions to the pressure.

Since equation (1) doesn't depend on temperature, we can integrate the structure equations

$$\frac{\mathrm{d}m}{\mathrm{d}r} = 4\pi r^2 \rho \tag{2}$$

$$\frac{\mathrm{d}P}{\mathrm{d}r} = -\rho \frac{Gm}{r^2} \tag{3}$$

by relating ρ to P via equation (1); in particular, we don't need an equation for $\mathrm{d}T/\mathrm{d}r$.

Equations (2) and (3) use r as the independent variable. But in a star, the radius is not set: stars can expand and contract, so we'd like our coordinates to follow the mass, rather than the radius. To do this, we rewrite these equations in LAGRANGIAN form, in which the mass m is the independent variable. We can do this because m varies monotonically with r according to eq. (2). To change the independent variable from r to m, we write

$$\frac{\mathrm{d}P}{\mathrm{d}r} = \frac{\mathrm{d}P}{\mathrm{d}m}\frac{\mathrm{d}m}{\mathrm{d}r} = 4\pi r^2 \rho \frac{\mathrm{d}P}{\mathrm{d}m},$$

so that

$$\frac{\mathrm{d}P}{\mathrm{d}m} = \frac{1}{4\pi r^2 \rho}\frac{\mathrm{d}P}{\mathrm{d}r} = -\frac{Gm}{4\pi r^4}.$$

Likewise,

$$1 = \frac{\mathrm{d}r}{\mathrm{d}r} = \frac{\mathrm{d}r}{\mathrm{d}m}\frac{\mathrm{d}m}{\mathrm{d}r} = 4\pi r^2 \rho \frac{\mathrm{d}r}{\mathrm{d}m}.$$

The two structure equations in Lagrangian form are therefore

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

$$\frac{\mathrm{d}P}{\mathrm{d}m} = -\frac{Gm}{4\pi r^4}. \tag{5}$$

1.2 SOLVING TWO-POINT BOUNDARY VALUE PROBLEMS BY SHOOTING

To solve equations (2) and (3) we also need to specify two boundary conditions. For stars, this is a bit tricky. Suppose we want to find the radius of a white dwarf

of mass M. We could start at the center, $m = 0$ and integrate equations (2) and (3) outwards until $m = M$—but what are the starting conditions? At the center of the star, we know that $r = 0$; but what value should we choose for $P_c = P(r = 0)$? Alternatively, we could start at the surface, where $P = 0$ and $m = M$, and integrate inward. But since we don't know R, this also won't work.

This type of problem is known as a TWO-POINT BOUNDARY VALUE PROBLEM: some boundary conditions are specified at the center, and some are specified at the surface. We therefore can't simply integrate equations (2) and (3). What we can do, however, is to combine our ODE solver with a rootfinder, such as bisection, into a algorithm known as SHOOTING:

1. Define a function $M(P_c)$ that takes a given central pressure P_c, and then integrates equations (2) and (3) from the center ($r = 0$, $m = 0$, $P = P_c$) outward until the surface is reached ($P = 0$). Once you reach $P = 0$, record the value of the mass, $M(P_c)$.

   Guess P_c and take your shot.

2. To obtain the radius of a white dwarf of mass M_{wanted}, find the root of the function

   $$f(P_c, M_{\text{wanted}}) = M(P_c) - M_{\text{wanted}};$$

   that is, find P_c such that $f(P_c, M_{\text{wanted}}) = 0$. For example, you might pick two central pressures, $P_{c,\text{low}}$ and $P_{c,\text{high}}$ that bracket our desired mass, $M(P_{c,\text{low}}) < M_{\text{wanted}} < M(P_{c,\text{high}})$, and then use bisection on $f(P_c, M_{\text{wanted}})$.

   Evaluate how you did and take a better shot.

2 Instructions

Implement a shooting algorithm and use it to produce a mass-radius relation for your model white dwarf. That is, integrate the white dwarf structure for masses $0.1\,M_\odot < M < 1.0\,M_\odot$ and thereby build up a table of M, P_c, and R. You will then compare your computed M-R relations against observational data.

This project reuses the ODE solver `rk4` developed as part of the first group project. You will need to develop a number of other modules, including the equation of state, the computation of the white dwarf structure, and the construction of the M-R relation. The following subsections provide a recipe for building up a code in a modular fashion, including testing each piece as you go.

2.1 Define physical and astronomical constants

First, we are going to need some physical and astronomical constants. We could type them in everywhere they are needed, but this wastes time and is error-prone. We should define the constants in a module, once, and then we can import the module as needed. Check out `astro_const.py`: in this module we import constants from `astropy.constants` and store them. To use these constants, simply import them as follows.

```
In [1]: import astro_const as ac

In [2]: print("Msun = ",ac.Msun)
Msun =  1.988409870698051e+30

In [3]: print("G = ",ac.G)
G =  6.6743e-11
```

The mathematical constant π is defined in `numpy`. You may define any other constants needed in `astro_const.py`.

Tip: In the file `astro_const.py` you may notice the line

```
34 au = _ac.au.value
```

This bit of code tells `Python` to execute the code that follows this line if you run this file as a standalone python file. For example,

```
06:32:13$ python astro_const.py
solar mass                   =  1.9884e+30 kg
solar radius                 =  6.9570e+08 m
solar luminosity             =  3.8280e+26 W
gravitational constant       =  6.6743e-11 m**3 s**-2 kg**-1
Planck constant              =  6.6261e-34 J s
Planck constant, reduced     =  1.0546e-34 J s
electron mass                =  9.1094e-31 kg
proton mass                  =  1.6726e-27 kg
neutron mass                 =  1.6749e-27 kg
atomic mass unit             =  1.6605e-27 kg
speed of light               =  2.9979e+08 m s**-1
Boltzmann constant           =  1.3806e-23 J K**-1
parsec                       =  3.0857e+16 m
astronomical unit            =  1.4960e+11 m
```

```
year                          =  3.1558e+07 s
Stefan-Boltzmann constant     =  5.6704e-08 W m**-2 K**-4
```

This is a useful feature: it means that each module can have a small built-in unit test. If the module is imported into another python routine, then the test code is ignored.

2.2 Implement and test the equation of state

After defining physical and astronomical constants, you are ready to build the equation of state. Check out the file `eos.py`, which defines two routines:

```
13 def pressure(rho, mue):

29 def density(p, mue):
```

The first routine computes $P(\rho, \mu_e)$; the second, $\rho(P, \mu_e)$. Recall that P is found from solving the ODE (3), but on each step we need ρ to compute $\mathrm{d}m/\mathrm{d}r$ and $\mathrm{d}P/\mathrm{d}r$; we therefore need to be able to compute the density given P.

Complete these two routines in `eos.py`. When finished, you will run `test_eos.py`; this routine calls the functions `pressure` and `density` and compares their results against values in the table `eos_table.txt`. Any discrepancies larger than `tolerance=1.0e-12` are flagged. You will need to run this test and ensure that your EOS routines produce the accepted values to the specified tolerance before proceeding.

For example,

```
06:32:16$ python test_eos.py
Comparing EOS to eos_table.txt...

   rho (table)    rho (test)    difference       P (table)      P (test)    difference

  1.000000e+05  1.000000e+05  1.455192e-16    6.810445e+14  6.810445e+14  0.000000e+00
  1.123655e+05  1.123655e+05 -2.590104e-16    8.271108e+14  8.271108e+14  4.533855e-16
  1.262600e+05  1.262600e+05  4.610142e-16    1.004504e+15  1.004504e+15 -7.466368e-16

  ...

  8.899530e+08  8.899530e+08  0.000000e+00    2.602879e+21  2.602879e+21  2.014262e-16
  1.000000e+09  1.000000e+09  0.000000e+00    3.161129e+21  3.161129e+21  1.658547e-16

SUCCESS: all values within tolerance
```

2.3 Implement the structure equations for the ODE solver

A template for the routine that computes the RHS of equations (4) and (5) is in `structure.py`:

```
16 def stellar_derivatives(m,z,mue):
17     """
18     RHS of Lagrangian differential equations for radius and pressure
19 
20     Arguments
21         m
22             current value of the mass
23         z (array)
24             current values of (radius, pressure)
25         mue
26             ratio, nucleons to electrons.  For a carbon-oxygen white dwarf,
27             mue = 2.
28 
29     Returns
30         dzdm (array)
31             Lagrangian derivatives dr/dm, dP/dm
32     """
33 
34     dzdm = np.zeros_like(z)
35 
36     # evaluate dzdm
37 
38     return dzdm
```

This routine will be called by `rk4` and will compute

$$\frac{\mathrm{d}z}{\mathrm{d}m} = \frac{\mathrm{d}}{\mathrm{d}m}\begin{bmatrix} r \\ P \end{bmatrix} = \begin{bmatrix} (4\pi r^2\rho)^{-1} \\ -Gm/(4\pi r^4) \end{bmatrix}.$$

2.4 Set the central boundary condition

The next challenge is what to do at the center, where $r \to 0$ and $m \to 0$. We can't simply put $\mathrm{d}P/\mathrm{d}m = -Gm/(4\pi r^4) = 0/0$: although we know how to handle limits, our poor Runge-Kutta scheme isn't as clever, so we need a numerical work-around. Recall from exercise 2.4 in the class notes that $\lim_{r\to 0} \mathrm{d}P/\mathrm{d}r = 0$; as a result, there is a small region about the center where $P = P_c = \text{const}$ to lowest order. If the pressure is constant in this region, then so is the density,

from eq. (1). We can therefore make a small, constant density core of mass $\delta_m \ll M$. Since this core has constant density, we can easily find its radius. We then start our numerical integration not from $m = 0$, but rather from $m = \delta_m$ with

$$\begin{aligned} m &= \delta_m && (6)\\ P(m = \delta_m) &= P_c && (7)\\ \rho(m = \delta_m) &= \rho(P = P_c), && (8)\\ r(m = \delta_m) &= \left(\frac{3\delta_m}{4\pi\rho_c}\right)^{1/3} && (9) \end{aligned}$$

where $\rho(P)$ is computed in the function `density(p,mue)` in the module `eos.py`. A starter routine for setting the central boundary condition is in `structure.py`,

```
40 def central_values(Pc,delta_m,mue):
41     """
42     Constructs the boundary conditions at the edge of a small, constant density
43     core of mass delta_m with central pressure P_c
44 
45     Arguments
46         Pc
47             central pressure (units = ?)
48         delta_m
49             core mass (units = ?)
50         mue
51             nucleon/electron ratio
52 
53     Returns
54         z = array([ r, p ])
55             central values of radius and pressure (units = ?)
56     """
57     z = np.zeros(2)
58     # compute initial values of z = [ r, p ]
59     return z
```

2.5 Construct the integration loop

To integrate our structure equations, we need to choose a stepsize h. As you saw in the previous project, there isn't necessarily an optimum stepsize that works throughout the domain. Moreover, if we accidentally take too large a

step near the surface of the star we could end up with a negative pressure—that would be a *complex* problem for the equation of state! We therefore want our steps to tiptoe up to the edge of the star.

One way to do this is to adjust the step h so that it is always less than the scale over which the radius or pressure changes appreciably. What is this scale? We can estimate the mass over which r and P change as

$$H_r = \frac{r}{|\mathrm{d}r/\mathrm{d}m|} = 4\pi r^3 \rho \tag{10}$$

$$H_P = \frac{P}{|\mathrm{d}P/\mathrm{d}m|} = \frac{4\pi r^4 P}{Gm}. \tag{11}$$

This is similar to the scale height you found in problem 2.2 of the coursepack. Before taking a step, we would then adjust the stepsize h to be some fraction ξ of the smaller of these two mass scales,

$$h = \xi \min(H_r, H_P). \tag{12}$$

A template routine to compute the lengthscales H_r, H_p is in `structure.py`,

```
61 def lengthscales(m,z,mue):
62     """
63     Computes the radial length scale H_r and the pressure length H_P
64 
65     Arguments
66         m
67             current mass coordinate (units = ?)
68         z (array)
69            [ r, p ] (units = ?)
70         mue
71             mean electron weight
72 
73     Returns
74         z/|dzdm| (units = ?)
75     """
76 
77     # fill this in
78     pass
79     return
```

How small should ξ be? One suggestion is to try some moderate value, say $\xi = 0.1$. Integrate with a trial P_c. Then rerun the integrations with $\xi = 0.05$.

Is the change in the values of M and R smaller than the desired tolerance? If not, cut ξ further and try again.

When should the integration stop? As mentioned above, we don't want to accidentally overshoot the edge and get a negative value of the pressure. One prescription is to stop the integration when

$$P < \eta P_c, \tag{13}$$

where η is some small value, for example $\eta = 10^{-10}$. As before, you should test this: do runs with several values of η, and decrease η until your final mass and radius have converged.

Because the stepsize is always changing, we don't know in advance how many steps we'll need to take. Is this a problem? We could simply keep marching along until condition (13) is reached; but what happens if there is an error that—for whatever reason—prevents that condition from being satisfied? We want our code to guard against entering an infinite loop! One way to handle this is to iterate over a predetermined maximum number of steps, say 10 000; and have an exit in the loop when the surface is reached, eq. (13). This is similar to what you did when writing your bisection algorithm. If we do end up taking the maximum number of steps, the routine produces an error. The following loop, in the routine

```
81 def integrate(Pc,delta_m,eta,xi,mue,max_steps=10000):
```

in `structure.py`, does this. Note the statement on lines 128–129. This is only reached if all 10 000 steps are taken, in which case an error condition is raised.

```
112     Nsteps = 0
113     for step in range(max_steps):
114         radius = z[0]
115         pressure = z[1]
116         # are we at the surface?
117         if (pressure < eta*Pc):
118             break
119         # store the step
120 
121         # set the stepsize
122 
123         # take a step
```

```
124
125         # increment the counter
126         Nsteps += 1
127     # if the loop runs to max_steps, then signal an error
128     else:
129         raise Exception('too many iterations')
```

(If we find that we legitimately do need to take more steps than this [we shouldn't], we can always reset `max_steps`.)

ALTHOUGH OUR GOAL IS TO FIND THE TOTAL MASS M AND RADIUS R OF THE WHITE DWARF, WE ARE INTERESTED IN THE INTERMEDIATE VALUES OF m, r, AND P. We'd therefore like to store the intermediate steps of our integration. Unlike in the previous project, here we don't know in advance how many steps we'll be storing.

We could overcome this by defining lists for these variables:

```
m_step = []
r_step = []
p_step = []
```

and then in the loop do something like

```
m_step.append(<current value of mass>)
. . .
```

This isn't wrong, but it is *inefficient*—allocating new memory is typically much slower than arithmetical operations. An alternative (faster, better) method for small computations is to allocate `NumPy` arrays that are at the maximum size we'll need (memory is abundant and cheap) and then just return the parts that are used, such as in the following loop.

The template integration routine is thus

```
81  def integrate(Pc,delta_m,eta,xi,mue,max_steps=10000):
82      """
83      Integrates the scaled stellar structure equations
84
85      Arguments
86          Pc
87              central pressure (units = ?)
88          delta_m
89              initial offset from center (units = ?)
```

```
90          eta
91              The integration stops when P < eta * Pc
92          xi
93              The stepsize is set to be xi*min(p/|dp/dm|, r/|dr/dm|)
94          mue
95              mean electron mass
96          max_steps
97              solver will quit and throw error if this more than max_steps are
98              required (default is 10000)
99
100     Returns
101         m_step, r_step, p_step
102             arrays containing mass coordinates, radii and pressures during
103             integration (what are the units?)
104     """
105
106     m_step = np.zeros(max_steps)
107     r_step = np.zeros(max_steps)
108     p_step = np.zeros(max_steps)
109
110     # set starting conditions using central values
111
112     Nsteps = 0
113     for step in range(max_steps):
114         radius = z[0]
115         pressure = z[1]
116         # are we at the surface?
117         if (pressure < eta*Pc):
118             break
119         # store the step
120
121         # set the stepsize
122
123         # take a step
124
125         # increment the counter
126         Nsteps += 1
127     # if the loop runs to max_steps, then signal an error
128     else:
129         raise Exception('too many iterations')
```

```
130
131     return m_step[0:Nsteps],r_step[0:Nsteps],p_step[0:Nsteps]
```

2.6 Test the integration

We still haven't specified the range of central pressures that are relevant to our problem. To do that, we need a plausible estimate for P_c given M. We also can't directly use our virial estimate, because we don't know R. To construct a guess for P_c, let's start by introducing some adjustable "knobs" into our virial estimates:

$$P_c = \alpha \frac{GM^2}{R^4} \tag{14}$$

$$\rho_c = \beta \frac{M}{R^3}. \tag{15}$$

Here α and β are unknown constants. Insert these scalings for P_c and ρ_c into our equation of state

$$P_c = K_e \left(\frac{\rho_c}{\mu_e} \right)^{5/3},$$

where K_e is a combination of all the constants in eq. (1), to obtain

$$\alpha \frac{GM^2}{R^4} = K_e \beta^{5/3} \frac{M^{5/3}}{R^5 \mu_e^{5/3}}.$$

A big big hint: Wouldn't it be nice if the constant K_e were defined somewhere so that other routines could `import` it?

We then solve for R,

$$R = \frac{\beta^{5/3}}{\alpha} \frac{K_e}{G} \left(M \mu_e^5 \right)^{-1/3},$$

and insert this expression back into equation (14) to obtain

$$P_c = \left(\frac{\alpha^5}{\beta^{20/3}} \right) \frac{G^5}{K_e^4} (M \mu_e^2)^{10/3}.$$

We've now eliminated R from the expression for P_c, but we have this unknown factor $\alpha^5/\beta^{20/3}$ in our expression. In the absence of further information, let's try $\alpha = \beta = 1$ and use

$$P_c^{\rm guess} = \frac{G^5}{K_e^4} (M \mu_e^2)^{10/3} \tag{16}$$

as a trial guess for the central pressure.

The testing procedure then becomes as follows.

1. Pick a trial M, say $M = M_\odot$. Compute $P_c^{\rm guess}$ from eq. (16), and then choose the integration parameters δ_m, ξ, and η and integrate.

2. Adjust δ_m, ξ, and η, one at a time, until you find a converged solution in M and R. That is, you want to make δ_m, ξ, and η sufficiently small that the result of the integration is insensitive to their precise values.

3. Once δ_m, ξ, and η are set, adjust P_c until you find a $P_{c,\rm low}$ and a $P_{c,\rm high}$ that will produce masses in the range $(0.1 \text{ to } 1.0)\, M_\odot$.

> **Challenge:** If your group is clever, you might figure out how to *calibrate* eq. (16) to make extremely accurate guesses for $P_c(M)$.

2.7 Make the mass-radius table

You should now have a routine that takes as input P_c and returns arrays m, r, P throughout the star. The central pressure will be `p[0]`, the total mass will be `m[-1]`, and the total radius will be `r[-1]`. (In `Python` the array index -1 indicates the last element of the array.)

Now we are ready to find P_c, and all the other white dwarf properties, for a specified mass $M_{\rm want}$. To do this, you can use either of the routines `bisect` or `brentq` in the module `scipy.optimize`. You will write a function $f(P_c, M_{\rm want}, \ldots)$ that returns $M(P_c) - M_{\rm want}$, and you will use this routine to find the P_c for which $f(P_c) = 0$.

Once you have that routine, you can then produce a mass-radius relation for your model white dwarfs, i.e., fill in Table 1. For each mass, you will determine the radius R and P_c. You will then list the central pressure P_c in MKS, and then compute $P_c/(GM^2/R^4)$. This last quantity is a check on our virial relation (14): is it the same for all masses? is it unusually large or small?

Table 1: Mass-radius relation for model white dwarf stars computed with ideal non-relativistic degenerate equation of state.

$M/M_\odot$	$R/R_\odot$	P_c (MKS)	$P_c/(GM^2R^{-4})$	ρ_c (MKS)	$\rho_c/[3M/(4\pi R^3)]$
0.1	...	...	...	...	...
0.2	...	...	...	...	...
0.3	...	...	...	...	...
⋮			⋮		
1.0	...	...	...	...	...

2.8 Compare to observations

Compare against observational data: plot your mass-radius relation, and put on the plot data points from Joyce et al. (2018) listed in Table 2. To save time, I've placed a module, `observations.py`, in your repository. The module reads in the data—stored in the file `Joyce.txt`—and makes it available as `NumPy` arrays.

Table 2: Mass and radius data from Joyce et al. (2018).

source	instrument	M	ΔM	R	ΔR
		$(M_\odot)$		$(0.01\,R_\odot)$	
Sirius B	HST	0.927	0.107	0.802	0.011
HZ43	HST	0.607	0.106	1.461	0.009
HZ43	FUSE	0.643	0.065	1.457	0.036
14 Aur Cb	HST	0.541	0.086	1.340	0.013
14 Aur Cb	FUSE	0.590	0.028	1.378	0.011
HD2133 B	HST	0.398	0.138	1.418	0.009
HD2133 B	FUSE	0.277	0.069	1.464	0.018
HR1358 B	HST	0.729	0.053	1.235	0.018
RE 0357	FUSE	0.543	0.039	1.420	0.014
HD 223816	FUSE	0.723	0.019	1.717	0.009
RE 1024	FUSE	0.559	0.030	2.183	0.043
REJ 1925	FUSE	0.524	0.040	1.452	0.024
Feige 24	FUSE	0.633	0.012	1.993	0.009
HD15638	FUSE	0.376	0.026	1.504	0.042

You use `observations.ph` as follows.

```
In [1]: from observations import MassRadiusObservations

In [2]: obs = MassRadiusObservations()

In [3]: obs.masses
Out[3]:
array([0.927, 0.607, 0.643, 0.541, 0.59 , 0.398, 0.277, 0.729, 0.543,
       0.723, 0.559, 0.524, 0.633, 0.376])

In [4]: obs.radii
Out[4]:
array([0.802, 1.461, 1.457, 1.34 , 1.378, 1.418, 1.464, 1.235, 1.42 ,
       1.717, 2.183, 1.452, 1.993, 1.504])
```

```
In [5]: obs.radius_errors
Out[5]:
array([0.011, 0.009, 0.036, 0.013, 0.011, 0.009, 0.018, 0.018, 0.014,
       0.009, 0.043, 0.024, 0.009, 0.042])

In [6]: for source, info in obs.sources.items():
    ...:     print('{0:20} M = {1:5.3f} +/- {2:5.3f} Msun'.format(source, info.m
    ...: ass, info.mass_error))
    ...:
Sirius B             M = 0.927 +/- 0.107 Msun
HZ43                 M = 0.643 +/- 0.065 Msun
14 Aur Cb            M = 0.590 +/- 0.028 Msun
HD2133 B             M = 0.277 +/- 0.069 Msun
HR1358 B             M = 0.729 +/- 0.053 Msun
RE 0357              M = 0.543 +/- 0.039 Msun
HD 223816            M = 0.723 +/- 0.019 Msun
RE 1024              M = 0.559 +/- 0.030 Msun
REJ 1925             M = 0.524 +/- 0.040 Msun
Feige 24             M = 0.633 +/- 0.012 Msun
HD15638              M = 0.376 +/- 0.026 Msun
```

References

S. R. G. Joyce, M. A. Barstow, S. L. Casewell, M. R. Burleigh, J. B. Holberg, and H. E. Bond. Testing the white dwarf mass-radius relation and comparing optical and far-UV spectroscopic results with Gaia DR2, HST, and FUSE. *MNRAS*, 479:1612–1626, September 2018. doi: 10.1093/mnras/sty1425.