# THRAIN 1.0
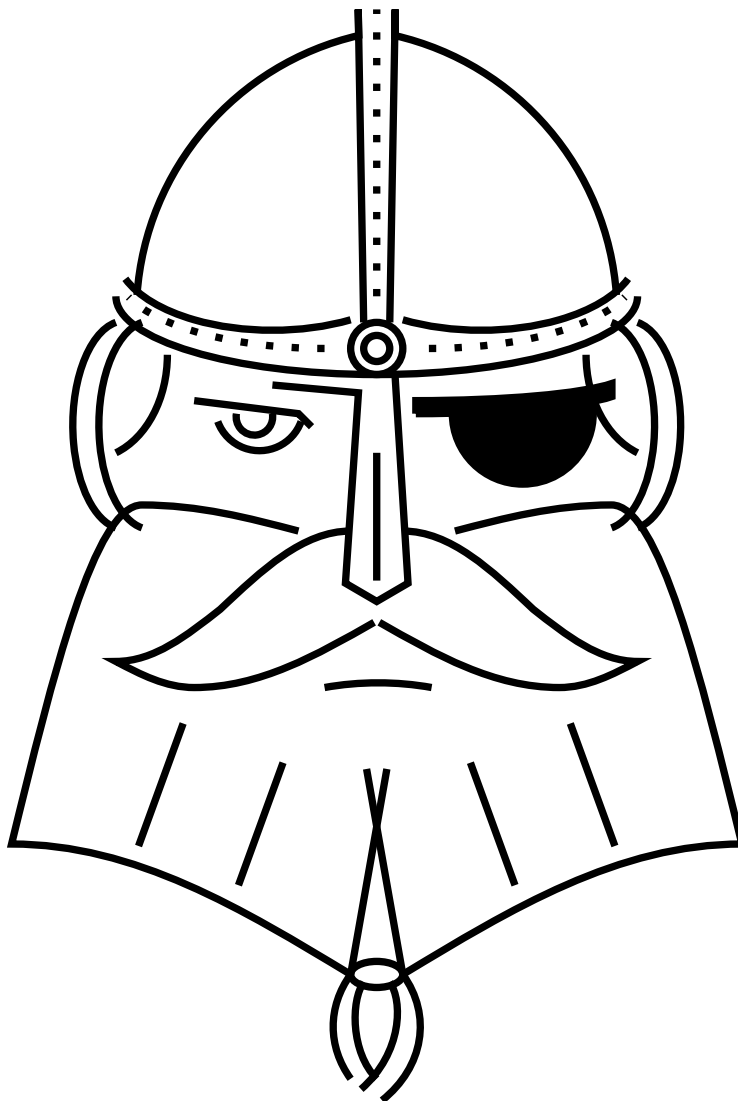
S. Reece Boston[*]

March 24, 2022

**THE MIGHTY WHITE DWARF CODE**

[*]rboston628@gmail.com

# Contents

# 1          Introduction

This document contains development notes and user notes for THRAIN developed as part of my thesis work. This document explains the equations and algorithms used, why certain choices are made, and how to use and further develop the program.

## 1.1    How to Cite This Program

If you use this program in scientific research, please cite it by the paper introducing it: Boston et al. (2022b)
    In bibtex, cite tentatively as

```
@PHDTHESIS{Boston2022,
    title = {Newtonian and Relativistic White Dwarf Asteroseismology},
    author = {Boston, S.~Reece},
    school = {University of North Carolina},
    year = {2022}
}
```

## 1.2    Version 1.0

This documentation was created for release with version 1.0, which was released with the above paper. As the code is expanded, the version may change.

## 1.3    Structure of Program

THRAIN follows OOP design. Most control of the program is contained within methods attached to classes. The classes act to encapsulate data and methods.
    One of the benefits of OOP design is polymorphism; the ability to use different classes interchangeably. This is facilitated by the use of abstract classes. There are three main abstract classes in the code:

- `Star`: Represents an equilibrium model of a star used as background in the wave equations. Described in 3.

- `Mode<N>`: Represents a single $klm$-mode, described by an `N`-dimensional eigenfunction (e.g. for a Cowling mode `N`=2). Described in 5.

- `ModeDriver`: Represents a particular form of a stellar wave equation (e.g. Cowling modes, Dziembowski equations, or non-adiabatic waves). Described in 4.

Classes `Star` and `ModeDriver` are abstract, meaning they only provide a template that other classes have to follow. It is up to the children of `Star` and `ModeDriver` to actually implement those methods.
    The structure of the code, the polymorphism pattern, and the relationship between the classes, is illustrated in Figure 1.2.
    In this document, we separate two facets of the classes, as they relate to the OOP structure:

**Interface** External-facing elements (data or functions) that specify how outside code can interact with the object.

**Implementation** The internal logic that calculates all relevant values inside of the object, and cannot be seen by external code.

As an example, all Stars have an method rho() in their interface, which must return the density $\rho$; however, the implementation can vary on how $\rho$ is found, whether by storing $\rho$ in an array, or calculating it from other quantities.

This is illustrated in Figure 1.1.
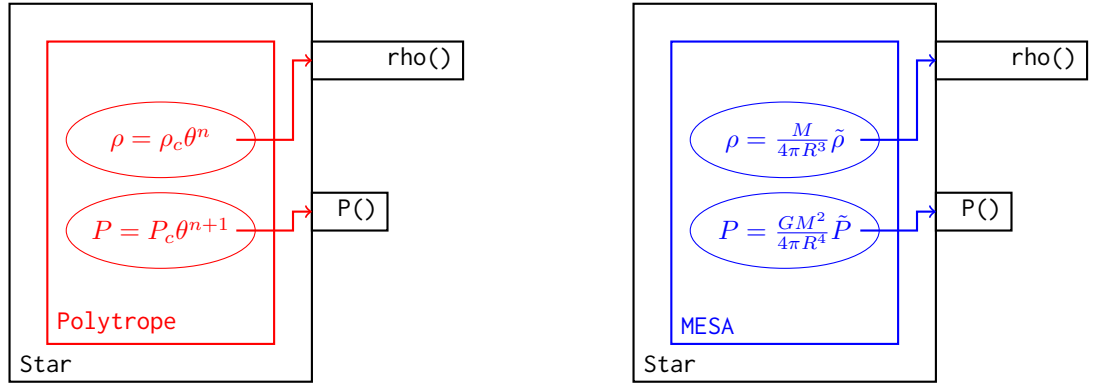


Figure 1.1: An illustration of a class (the boxes labeled Star). The interface (the "pipes" rho(), P()) can connect to external code needing a star's density and pressure. The internal logic of the implementation (the circles) is illustrated for two Star daughters: the Polytrope class (red) and the MESA class (blue). External code cannot access the internal implementation, and does not care what kind of star is returning $\rho, P$.

## 1.4    Diagram of Program Structure
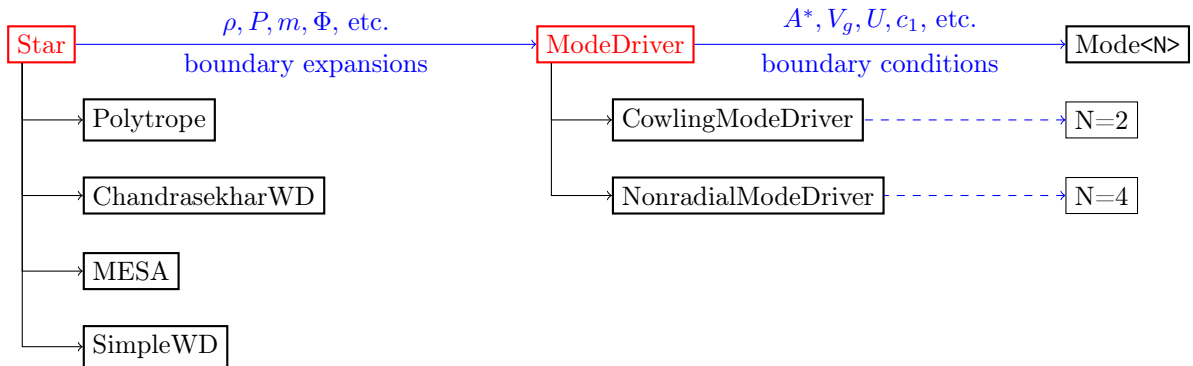


Figure 1.2: Structure of program, showing hierarchies. Red boxes are abstract classes, which provide rules for child classes to follow. Blue lines show friendship. Any figure lower in the hierarchy (a child) can be used in place of a parent or grandparent class. Therefore, all stellar and wave models within the code are interchangeable.

# 2 Using the command line program

## 2.1 Installation and Set Up

This program was developed in macOS and originally compiled with `gcc` and `c++11`. If your `gcc` is up-to-date, then the included makefile should prepare everything for you. Linux users should be able to make this work without too much intervention. Windows users should consider installing a real operating system.

Compiling should work by running `make` from the THRAIN home directory.

If you encounter errors, manually type

```
mkdir -p obj/STARS
mkdir -p obj/MODES
```

to create the correct folders for object files. Then manually compile the library files by first `cd`'ing to the `lib` directory and typing

```
make -f makelib
```

This will create the library. Next `cd` back to main directory and type `make`. That will create the entire program.

If you have persistent problems, check that you have the latest version of `gcc` installed, capable of C++11. If there are still problems, please email me.

## 2.2 Creating a white dwarf input file

THRAIN is meant to be run from the command line, from executable `thrain`. User specification for the stellar background and the modes should be placed in the input file. The input file should be structured as follows:

- Line 1: the word `Name:` followed by a keyword to be used when naming files related to this calculation.

- Line 2: the word `Model:` followed by the following specifications, separated by a space:

  - the keyword `newtonian`
  - the keyword `swd`
  - an integer for the number of grid points to use in calculation.

- Line 3: the keyword `EOS:`

- Line 4: the word `core`, followed by keywords indicating the **core** equation of state. Any of the following, separated by whitespace:

  - `rad`, radiation pressure
  - `ideal`, pressure from ideal gas law
  - `coul`, Coulomb pressure
  - `deg_zero`, electron degeneracy pressure with zero temperature

- **deg_trap**, electron degeneracy pressure with finite temperature by integrating the Fermi distribution

- Line 5: the word `atm`, followed by keywrods indicating the **atmosphere** equation of state. Any of the above keywords, separated by whitespace.

- Line 6: the words `chemical parameters:`

- Line 7: the keyword `he` followed by three parameters specifying the log-logistic curve of the Helium abundance (see `SimpleWD`).

- Line 8: the keyword `c` followed by three parameters specifying the log-logistic curve of the Carbon abundance (as above).

- Line 8: the keyword `o` followed by three parameters specifying the log-logistic curve of the Oxygen abundance (as above).

- Line 10: the word `Params:` followed by:

  - the word `mass` and a real number indicating the total mass of the star
  - the word `teff` and a real number indicating the effective temperature in Kelvin

- Line 11: the word `Units:` followed by a keyword specifying:

  - `CGS`, currently the main one working
  - `geo`, for using geometric units as in GR
  - `SI`, for using SI units
  - `astro`, for using "astronomical units", where mass is in $M_\odot$, distance is in km, and time in seconds

  Currently, only `CGS` and `astro` are confirmed to behave properly.

- Line 12: this must be a blank line.

- Line 13: the word `Frequencies:` followed by the mode type to use and the adiabatic index to use. Mode type options are:

  - `cowling`
  - `nonradial`

  The adiabatic index $\Gamma_1$ is a number. It may be specified as a simple fraction, as in `5/3` or `4/3`. Other indices must be specified in decimal format. To use the adiabatic index $\Gamma_1 = (\partial \log P / \partial \log \rho)_{ad}$ calculated from the stellar background then use `0`.

- Line 14 onward: Beginning here and onward, write the mode numbers for each mode you wish to calculate on a separate line. These are specified as $\ell, k$, where $k$ counts the nodes and $\ell$ the angular momentum. Negative $k$ specify g-modes, positive specify p-modes. Note that there is no $1, 0$ mode, so specifying one can cause the program to hang.

The file should be saved in the same file as THRAIN. Assuming THRAIN has been properly compiled and the file called `myinput.txt`, then you can run with

```
./thrain myinput.txt
```

## 2.3 Creating input files for other models

THRAIN is also capable of creating simple stellar models, or reading in stellar models from MESA output. To run a calculation using one of these models, the input file should be structured as follows:

- Line 1: the word `Name:` followed by a keyword to be used when naming files related to this calculation.

- Line 2: the word `Model:` followed by the following specifications, separated by whitespace:

  - the word `newtonian`
  - one of the following model specifications:
    * `polytrope` for a polytrope, followed by a real number for the polytrope index $n$
    * `MESA` to use a MESA file, followed by a string with name of the `dat` file (omiting the ".`dat`"). All MESA files must be in the THRAIN directory.
    * `CHWD` for a Chandrasekhar WD, followed by a real number for initial value $y_o$ (see `ChandrasekharWD`)
  - an integer for the number of grid points to use. For MESA models this is a *suggestion*.

- Line 3: in **polytropes**, the word `Params:` followed by two of the following:

  - `mass` and a real number indicating the total mass of the star
  - `radius` and a real number indicating the total radius of the star
  - `logg` and a real number indicating the $\log_{10} g$ surface gravity of star
  - `zsurf` and a real number indicating the surface redshift

- Line 4: in **all models**, the word `Units:` followed by a keyword specifying:

  - `CGS`, currently the main one working
  - `geo`, for using geometric units as in GR
  - `SI`, for using SI units
  - `astro`, for using "astronomical units", where mass is in $M_\odot$, distance is in km, and time in seconds

  Currently, only `CGS` and `astro` are confirmed to behave properly.

- Line 5: this must be a blank line.

- Line 6: in **all models** the word `Frequencies:` followed by the mode type to use and the adiabatic index to use. Mode type options are:

  - `cowling`
  - `nonradial`

  The adiabatic index $\Gamma_1$ is a number. It may be specified as a simple fraction, as in `5/3` or `4/3`. Other indices must be specified in decimal format. To use the adiabatic index $\Gamma_1 = (\partial \log P / \partial \log \rho)_{ad}$ calculated from the stellar background then use `0`.

- Line 7: Beginning here and onward, write the mode numbers for each mode you wish to calculate on a separate line. These are specified as $\ell, k$, where $k$ counts the nodes and $\ell$ the angular momentum. Negative $k$ specify g-modes, positive specify p-modes. Note that there is no $1, 0$ mode, so specifying one can cause the program to hang.

The file should be saved in the same directory as THRAIN, and is executed the same as WD input files.

## 2.4 Comments in Input Files

Comments may be placed in an input file, subject to the following restrictions:

- Comments must begin with #.

- Comments may only be placed **at the top of the file**.

- Blank lines may be used within the comment section, but not after.

Once the input part of the file begins, neither comments nor line breaks may be used. You can see this examples of comment use within the sample files.

## 2.5 The output

When the calculation begins, THRAIN will create a directory specific to this calculation based on the given name, located in the `output` directory. The first output it makes is an echo of the input file, for re-running the specific calculation. Assuming your calculation was given the name `myname`, this file will be called `myname_in.txt`. The main output file will be called `myname.txt`. This contains data about the star, and summary results for all modes calculated.

The program will also create subdirectories called `modes` and `star`. These contain graphs and data files relevant either to the stellar background or to the modes. The `modes` directory can take up a lot of memory, so if you do not think you will need graphs of every mode, you may want to delete this directory's contents.

## 2.6 Sample Input

There are several sample inputs included in the distribution. They perform the following:

**sampleinput1.txt** Will create results for an $n = 0$ polytrope to compare against Pekeris formula.

**sampleinput4.txt** Will create a table which can be compared to results in Christensen-Dalsgaard & Mullan (1994) paper for polytrope eigenmodes.

**sampleinput5.txt** Shows how to use `MESA` data files.

**sampleinput6.txt** Shows how to use create a simple WD model.

# Part I

# Base Classes

# 3    The `Star` Abstract Class
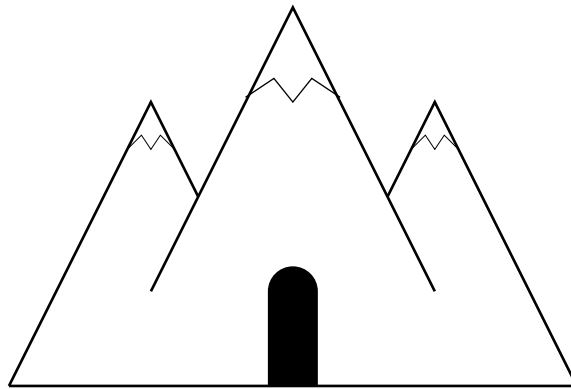
This is an abstract class. In any function that requires a `Star` object, any child of `Star` – such as `Polytrope` or `MESA` – may be used without any additional changes to the program.

## 3.1    The `Star` Interface

The `Star` class has the following methods for external code to interact with:

### 3.1.1    Physics

Global stellar properties:

- `double Radius()`: total radius $R_\star$ of the surface of the star.

- `double Mass()`: total mass $M_\star$ of the star.

- `double Gee()`: value of Newton's constant $G$ in correct units for the star.

- `double light_speed2()`: the squared speed of light in correct units for the star.

Basic stellar properties:

- `double rad(int X)`: radial coordinate $r$ at index `X`.

- `double rho(int X)`: density $\rho(r)$.

- `double drhodr(int X)`: derivative of density $\frac{d\rho}{dr}(r)$.

- `double P(int X)`: pressure $P(r)$.

- `double dPdr(int X)`: derivative of pressure $\frac{dP}{dr}$.

- `double Phi(int X)`: gravitational potential $\Phi(r)$.

- `double dPhidr(int X)`: gravitational field $g = \frac{d\Phi}{dr}(r)$.

- `double mr(int X)`: interior mass $m(r)$.

- `double sound_speed2(int X, double gam1)`: squared sound speed inside star, $v_s^2 = \frac{\texttt{gam1}P}{\rho}$.

Stellar properties used in wave equations:

- `double Schwarzschild_A(int X, double gam1)`: the Schwarzschild discriminant $A$, using $\Gamma_1 = \texttt{gam1}$.

- `double getAstar(int X, double gam1)`: a dimensionless Schwarzschild discriminant $A* = -rA$, using $\Gamma_1 = \texttt{gam1}$.

- `double getU(int X)`: the dimensionless variable $U(r) = \frac{d\ln m(r)}{d\ln r}$.

- `double getVg(int X, double gam1)`: the dimensionless variable $V_g(r) = -\frac{1}{\Gamma_1}\frac{d\ln P}{d\ln r}$ using $\Gamma_1 =$ `gam1`.

- `double getC(int X)`: the dimensionless variable $c_1 = \frac{M_\star}{m(r)}\frac{r^3}{R_\star^3}$.

- `double Gamma1(int X)`: the adiabatic exponent $\Gamma_1 = \left(\frac{\partial P}{\partial \rho}\right)_{\text{ad}}$.

### 3.1.2 Miscellaneous

- `int indexFit`: the index where shoot-to-center algorithms should try to match solutions.

- `int length()`: the maximum index of arrays in the star.

- `double SSR()`: returns a fractional root-mean-square residual of the stellar model when returned to original equilibrium equations.

- `char* name[]`: a C-string holding the star name, used in printing plots and files. Must comply with UNIX directory conventions.

- `void graph_title(char* c)`: replaces `c` with a name to be used in printed graphs of the stellar profile.

- `void writeStar(char *c)`: outputs values of `Star` and plots in gnuplot. The optional argument `c` specifies a directory path where to print the star.

- `void printStar(char *c)`: same as `writeStar()`, but also opens the plot image for easy viewing.

### 3.1.3 Boundary conditions for modes

- `void getXYZCenter(double*, int& maxPow)`: where XYZ could be `Astar` $= A^*$, `U` $= U$, `Vg` $= V_g$, `C1` $= c_1$, with $U, V_g, c_1$ as above, and $A^* = -rA$. Returns coefficients from power-series expansion of XYZ in $x = r/R$ near the center. The highest power desired is `maxPow`; if the stellar model does not implement that power, then `maxPow` will be lowered to match. The coefficients are returned within the array. In the future, this should include methods for $\rho, P, m$.

- `void getXYZSurface(double*, int& maxPow)`: as above, but coefficients from power-series expansion in $t = 1 - r/R$ near the surface.

## 3.2 The `Star` Implementation

This is an abstract class, meaning that (almost) all of its methods are declared `virtual`, so that children classes must specify how all of these functions work. The only methods that `Star` actually defines are:

`writeStar(char* c)` writes out $r, \rho, P, \Phi$ as columns in a tab-separated `.txt` file, then plots $\rho$ and $P$ vs $r$ into a `.png` file with gnuplot. The output files are named according to the model description in `name`.

`printStar()` as above, but additionally opens the plot to the screen. The optional argument `c` specifies a directory path where output should be written.

`SSR()` explained in Sec. 3.2.1.

### 3.2.1 The generic stellar `SSR()`

One method of quantifying the fitness of a numerical solution is by re-substituting the numerical solution into the original equations and calculating the residual. For a general Newtonian star, the hydrostatic conditions are

$$0 = \frac{dP}{dr} + \rho\frac{d\Phi}{dr} \tag{3.1a}$$

$$\nabla^2\Phi = 4\pi G\rho. \tag{3.1b}$$

At each location X, we calculate a scaled residual

$$\text{res}_1[\text{X}] = \frac{|\text{dPdr}(\text{X}) + \text{rho}(\text{X}) \cdot \text{dPhidr}(\text{X})|}{|\text{dPdr}(\text{X})| + |\text{rho}(\text{X}) \cdot \text{dPhidr}(\text{X})|} \tag{3.2a}$$

$$\text{res}_2[\text{X}] = \frac{|4\pi G \cdot \text{rho}(\text{X}) \cdot \text{rad}(\text{X}) - 2\text{dPhidr}(\text{X}) - \text{d2Phi}|}{|4\pi G \cdot \text{rho}(\text{X}) \cdot \text{rad}(\text{X})| + |2\text{dPhidr}(\text{X})| + |\text{d2Phi}|} \tag{3.2b}$$

where $\text{d2Phi} = \frac{d^2\Phi}{dr^2}$ and is computed numerically. The SSR is then

$$SSR = \sqrt{\frac{1}{2 \cdot \text{len}} \sum_{X=0}^{\text{len}-1} [\text{r}_1[\text{X}]^2 + \text{r}_2[\text{X}]^2]}. \tag{3.3}$$

For practicality, the first few points near either border are omitted from the sum. Note that SSR is a misnomer, as this is actually an RMSR (root-mean-square residual).

## 3.3   How to make new `Star` children

If you wish to add a new object representing a stellar model to the code, you need to program the above functions. You need a way of determining $\rho, P, \ldots$, and also $A, V_g, c_1, U$, the total mass and radius $M, R$, *and find power series expansions* for $A^*, V_g, c_1, U$ near the surface and the center. The easiest method is to copy `Polytrope` and change as necessary. If the star model solves different equilibrium equations as those in (3.1) (such as in rotating stars, or 1PN stars), then the `SSR()` method must be overwritten.

## 3.4   References

- Christensen-Dalsgaard, J. and D. J. Mullan (1994), in the appendix, for an example of the process described in 3.1.3.

- Tassoul, Fontaine, Winget (1990), in their equations 40, 41 for examples of the SR method of measuring numerical error described in **??**.

# 4  The `ModeDriver` Abstract Class

This is an abstract class, representing a set of wave equations within a star. The equations are assumed to be in the dimensionless linear form

$$x\frac{d}{dx}\vec{Y} = \mathbf{A}\vec{Y}, \tag{4.1}$$

where $x = r/R$ is a radial variable, $\vec{Y}$ is the normal mode eigenvector and $\mathbf{A} = (a_{ij})$ is a square matrix, so that

$$x\frac{dy_i}{dx} = \sum_j a_{ij}y_j. \tag{4.2}$$

The job of `ModeDriver` objects is to provide the coefficients $a_{ij}$ and boundary conditions to a `Mode<N>` object. This division provides greater flexibility and intercompatibility for different physics within the code.

## 4.1  The `ModeDriver` Interface

The `ModeDriver` definition specifies the following methods for external code to be able to interact with it:

### 4.1.1  Constants

- `const size_t num_var`: a constant integer specifying the number of eigenfunctions in this form of wave equation.

### 4.1.2  Physics

- `double rad(int X)`: the radial coordinate `rad[X]` $= x = r/R_\star$.

- `double Gamma1()`: the adiabatic exponent of the perturbations, $\Gamma_1$, as in $\frac{\delta P}{P} = \Gamma_1 \frac{\delta\rho}{\rho}$. If set to zero, then $\Gamma_1 = (\partial\log P/\partial\log\rho)_{ad}$ from the stellar background is used. This can also be set to a specific number (e.g. $\Gamma_1 = 5/3$), useful with models like polytropes.

- `void getCoeff(double *CC, int X, int b, double w2, int l)`: replaces `CC` $= \mathbf{A}$, the coefficient matrix at the index X and interpolated by spacing 0.5∗`b` for use in an RK4 routine. Uses frequency $\tilde{\omega}^2 =$ `w2` and angular momentum number $\ell$ given. The equation form `Mode<N>` expects is shown in (4.2) (do **not** divide the $a_{ij}$ by the $x$ on the LHS).

### 4.1.3  Calculation

- `int length()`: returns the correct array size for modes.

- `void getBoundaryMatrix(int, double*, double*, double**, int*)`: return a matrix that allows `Mode<N>` to calculate a Wronskian, and an array in integers that allows `Mode<N>` to rescale solutions to match in the center. See Sec. 4.3 below and Sec. 5.2.5 in `Mode<>` for a fuller explanation.

### 4.1.4   Miscellaneous

- `double SSR(double, int, ModeBase*)`: returns a sum-square-residual from back-substitution of mode solution into the original physical equations, normalized by mode amplitude. Used as a check of accuracy of the mode solution.

- `void varnames(std::strings* vn)`: populates a vector of strings `vn` with names to use for each of the eigenfunction variables when forming plots.

### 4.1.5   Constructor

There is only one constructor:

- `ModeDriver(int nv, Star *s)`: initialize the number of variables needed, and set stellar background.

Because this class is **abstract**, the constructor does not actually create an object; it is instead needed in order for daughter classes to initialize the `const` variables `num_var` and `*star`.

### 4.1.6   Boundary conditions for modes

- `setupBoundaries()`: populate arrays containing power series coefficients of relevant stellar properties near center and surface, so that mode eigenfunctions can be calculated at these boundaries.

- `int CentralBC(double **y, double *yo, double w2, int l, int m=0)`: calculate eigenfunction `y[][]` at center, so that outward mode integration begins at the returned integer. Near the center of a spherical star, we can expand each eigenfunction $y_i$ in the form

$$y_i(x) = y_{i0} + y_{i2}x^2 + y_{i4}x^4 + \cdots \tag{4.3}$$

  which allows us to find values near the center using only $y_{i0}, y_{i2}, y_{i4}$. The goal of `CentralBC()` is to find those coefficients given the equation form represented in **A** and the frequency `w2`; this normally takes the form of a recursion algorithm involving series expansions of **A**. We then populate

$$\texttt{y[i][X]} = y_{i0} + y_{i2} \cdot \texttt{rad[X]}^2 + y_{i4} \cdot \texttt{rad[X]}^4 + \cdots.$$

  The maximum position `X` where this procedure is used is the returned value, and normal integration should take over from there.

- `int SurfaceBC(double **y, double *ys, double w2, int l, int m=0)`: calculate eigenfunction `y[][]` at surface, so that outward mode integration begins at the returned integer. Near the surface of a star, we can expand each eigenfunction $y_i$ in the form

$$y_i(x) = y_{i0} + y_{i1}t + y_{i2}t^2 + \cdots \tag{4.4}$$

  for $t = 1 - x$, similarly to the center. `SurfaceBC()` works similarly to its central counterpart, finding these coefficients and populating the eigenfunction near the surface

$$\texttt{y[i][len-X]} = y_{i0} + y_{i1} \cdot \texttt{t} + y_{i2} \cdot \texttt{t}^2 + \cdots.$$

  The miminum position `len-X` where this procedure is used is the returned value, and standard numerical integration takes over from there.

For concrete description, see the included Mathematica notebook `NewtonianBC.nb`.

## 4.2   The `ModeDriver` Implementation

The above section specifies all the ways external code can interact with a `ModeDriver` object. Internally, a `ModeDriver` must also specify:

- `Star* star`: a pointer to the `Star` object to serve as stellar background.

- `int num_var`: the number of eigenfunctions $y_i$ for this mode type.

- `int central_bc_order,surface_bc_order`: the desired order of series expansions in $y_i$ to use at the center and surface.

Because mode types can differ greatly one from the other, `ModeDriver` does not actually define any functions (unlike `Star`). Everything must be implemented by child classes.

## 4.3   Solution method

This class was made to interface with `Mode<N>` to solve for normal modes of stellar oscillation. Because `Mode<N>` is physics-agnostic, all physics is carried by `ModeDriver` objects.

At each grip point and half-grid point, the `ModeDriver` must provide a coefficient matrix $\mathbf{A}$, through the function `getCoeff()`. The exact values of the coefficients in $\mathbf{A}$ will depend on the physics of the waves being implemented.

If there are $N = $ `numvar` eigenfunctions in the wave solution, then there are also $N$ independent solutions. However, there is only one physical solution, which must satisfy all $N$ of the boundary conditions. There are usually $N/2$ BCs at the center, and $N/2$ at the surface; there are also $N/2$ degrees of freedom at the center and $N/2$ at the surface. We can then relate each d.f. to one BC, and to one independent solution. A schematic of this is shown in Fig. 4.1.

It is up to `getBoundaryMatrix()` to explain how to make these independent solutions in each region, by specifying the d.f.s needed tp create each independent solution, the order to create them, and then how to match them at the center. For example, if $N = 4$ and `getBoundaryMatrix()` returns the matrix below, it leads to the following specifications of BCs to create the independent solutions:

$$
\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \longrightarrow
\begin{array}{rcl}
\vec{Y}^{(c1)} &=& (y_1, y_2, 0, 0) \\
\vec{Y}^{(c2)} &=& (0, 0, y_3, y_4) \\
\vec{Y}^{(s1)} &=& (0, y_2, y_3, y_4) \\
\vec{Y}^{(s2)} &=& (y_1, y_2, 0, 0)
\end{array}
\tag{4.5}
$$

where $\vec{Y}^{(c1)}, \vec{Y}^{(c2)}$ are solutions in the center and $\vec{Y}^{(s1)}, Y^{(s2)}$ are solutions at the surface. The further output of `getBoundaryMatrix()` is a list of indices to combine the four solutions to make the physical solution. The process is explained further in Sec. 5.2.5 of `Mode<N>`.

## 4.4   How to make new `ModeDriver` children

If you wish to add a new object representing different stellar waves to the code, you need to add the above functions. If your mode type does not use the same functions as the Dziembowski equation (i.e. $A^*, V_g, c_1, U$), then you will need for `setupBoundaries()` to find a way to extract power series expansions of the functions you use to the desired order: this is usually possible with numerical derivatives of quantities such as $\rho, P, m, g$, which can be found through the equivalent methods in `Star`. You will further need to work out a recursion algorithm that calculates coefficients $y_{i0}, y_{i1}$, etc. near the center and surface based on the form of the equations you are using. The easiest method is to copy `NonradialModeDriver` and change whatever necessary.
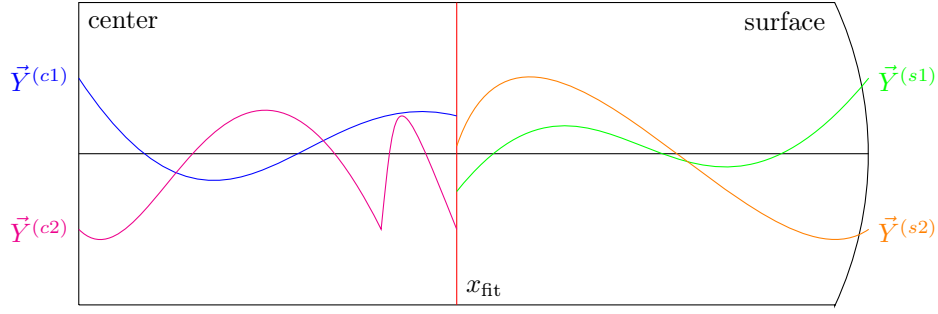
Figure 4.1: A sketch of the solution process for $N = 4$. The matching of the four independent solutions at $x_{\text{fit}}$ allows for finding the eigenvalue $\bar{\omega}^2$ and the physical solution. To create the independent solutions, `getBoundaryMatrix()` gives `Mode<N>` a matrix for assigning boundary values.

## 4.5 References

- Unno et al., 1979

- Dziembowski (1971), for the standard equation set assumed throughout the program, which describes the wave equations using structure variables $A^*, V_g, c_1, U$.

- Cox (1980), *Theory of Stellar Pulsation*, Section 17.6, for a brief of the recursion process of boundary conditions of the wave equations. The coefficients calculated in `setupBoundaries()` are similar to those in Cox's equation 17.63a-e, and the coefficients $y_{in}$ calculate by `CentralBC()` are similar to those in Cox's equation 17.60a-c.

- Christiansen-Dalsgaard & Mullan (1994), for the same boundary recursion process at the surface.

# 5         The Mode<N> Class

This class represents a physical mode of oscillation within a star. Each Mode has num_var=N variables $y_1, \ldots, y_N$, a dimensionless frequency $\bar{\omega}^2 =$ omega2, and mode numbers $k, \ell, m =$ k, l, m, where $k$ is the radial mode order and $\ell, m$ are the angular harmonic numbers. The mode can optionally have $\Gamma_1$ defined separately from the stellar background, as is necessary for polytrope calculations.

Mode<N> expects to be solving mode equations of the form

$$x\frac{d\vec{Y}}{dx} = \mathbf{A}\vec{Y}, \tag{5.1}$$

where $\vec{Y} = \langle y_1, \ldots, y_N \rangle$ is the eigenvector and $\mathbf{A} = (a_{ij})$ is a square N $\times$ N matrix, so that

$$x\frac{dy_i}{dx} = \sum_j a_{ij} y_j. \tag{5.2}$$

It is necessary for ModeDriver objects to provide the coefficients $a_{ij}$, which contain all of the physics of the wave behavior. Note the factor of $x$ on the LHS of the differential equation.

## 5.1    The Mode<N> Interface

The Mode<N> definitions specifies the following methods for external code to be able to interact with it:

### 5.1.1    Mode Properties

Methods related to the mode numbers $k, \ell, m$:

- int modeOrder(): radial mode order, $k$.

- void modeNumbers(int& k, int& l, int& m): replaces arguments with all mode numbers $k, \ell, m$.

Methods related to the frequency:

- double getOmega2(): the square dimensionless frequency $\bar{\omega}^2$ ($\omega^2 = \frac{GM}{R^3}\tilde{\omega}^2$).

- double getFreq(): returns the angular frequency $\omega$ in rad/s.

- double getPeriod(): the period $\Pi = 2\pi/\omega$, in s.

### 5.1.2    Physics

- double getRad(int X): the radial coordinate rad[X] $= x = r/R_\star$.

- double getY(int i, int X): the eigenfunction y[i][X] $= y_i(\text{rad}[X])$.

- double tidal_overlap(): calculates a tidal overlap coefficient.

### 5.1.3 Miscellaneous

- `double SSR()`: returns a root-mean-square-residual from back-substitution of mode solution into the original physical wave equations (e.g. the LAWE), normalized by mode amplitude. Used as a check of accuracy of the mode solution.

- `void writeMode(char *c)`: prints $x, y_1, \ldots, y_N$ to external files and forms plots in gnuplot. The point `xfit` where the double-shooting method is matched is highlighted by a <span style="color:red">red vertical line</span> – be sure . The optional argument `c` specifies a directory path where to print the star.

- `void printMode(char *c)`: writes the mode as above, and then opens the plotted file in an external window.

### 5.1.4 Constructors

There are three constructors for creating modes. They are very similar, and differ only in how they find an initial guess for $\bar{\omega}^2$:

- `Mode<num_var>(int k, int l, int m, ModeDriver*)`: creates a mode with anticipated mode numbers $k, l, m$. The starting value of $\bar{\omega}^2$ is picked based on the formula (5.3) for a uniform-density stellar model.

- `Mode<num_var>(double w2, int l, int m, ModeDriver*)`: creates a mode with initial guess $\bar{\omega}^2 = $ `w2`.

- `Mode<num_var>(double omegMin, double omegMax, int l, int m, ModeDriver*)`: creates a mode where the eigenfrequency $\bar{\omega}^2$ is known to fall in the range $\bar{\omega}^2 \in [\mathtt{omegMin}, \mathtt{omegMax}]$.

In all cases, $\bar{\omega}^2$ is found with a bisection search aided by Newton's method.

## 5.2 The `Mode<N>` Implementation

The above section specifies all the ways external code can interact with a `Mode` object. In this section, we will explain how the details of the calculation are performed.

### 5.2.1 Constants

- `int num_var`: number of variables, `num_var` $=$ `N`, which is specified by `ModeDriver`.

- `int k, int l, int m`: the spherical harmonic quantum numbers $k, l, m$ of the oscillation.

- `double cee2, Gee`: values for $c^2, G$ in appropriate units, for scaling laws.

- `double sig2omeg`: a conversion factor, $\bar{\omega}^2 = \mathtt{sig2omeg} * \omega^2$.

- `int len`: the maximum index of the perturbation arrays.

- `int len_star`: the number of grid points in the stellar background model.

- `int xfit`: the index where shoot-to-center methods at surface and center are to match.

- `double Gamma1`: the adiabatic exponent of the perturbations, $\Gamma_1 = \delta \log P / \delta \log \rho$, as set from the `ModeDriver`.

### 5.2.2 Physics

- `double omega2`: the square of the dimensionless eigenfrequency, $\bar{\omega}^2 = \frac{R^3}{GM}\omega^2$.

- `double *rad`: an array holding the radial variable $x = r/R_\star$.

- `double **y`: 2D array holding the perturbation variables of the star: `y[0][x]` $= y_1(x)$, `y[1][x]` $= y_2(x)$, etc.

- `Star *star`: a pointer to a `Star` object containing the background equilibrium model. Must be the same `Star` object as the related `ModeDriver`.

- `ModeDriver *driver`: a pointer to a `ModeDriver` object, which handles all quantities pertinent to the background stellar model, including implementing the boundary conditions.

### 5.2.3 Calculation

- `void basic_setup()`: a demiurge constructor used by all three constructors to setup `r[]`, `y[][]` arrays, calculate `xfit`, and prepare the boundary matrix.

- `double boundaryMatrix[num_var][num_var]`: a matrix used to specify how to form independent solutions for constructing a Wronskian.

- `int indexOrder[num_var]`: an array of indices for rescaling solutions to match at center.

- `double yCenter[num_var], ySurface[num_var]`: starting values of $\vec{Y}$ at center and surface.

- `void converge()`: the routine to calculate eigenfrequency $\bar{\omega}^2$. Does not need to be called outside of the constructor.

- `bool converged`: a flag, indicates if mode calculation reached the end of convergence process. Deprecated.

- `double RK4out(int xmax, double, double y0[num_var]`: given central values `y0[]`, integrate using RK4 from the center up to index `xmax`.

- `double RK4in(int xmin, double, double ys[num_var]`: given surface values `ys[]`, integrate using RK4 from the surface down to index `xmin`.

- `double RK4center(double, double y0[num_var], double ys[num_var])`: given central and surface values `y0[]`, `ys[]`, calculate a Wronskian $W$ by shooting outward and inward to `xfit`. The Wronskian is the return value.

- `convergeBisect(double tol)`: find $\bar{\omega}^2$ using a bisection search for the zero of Wronskian $W$, stopping when the brackets are within `tol` of one another. Newton's method is used to find the starting brackets.

- `convergeNewton(double tol, int term)`: find $\bar{\omega}^2$ using Newton's method to search for the zero of Wronskian $W$, stopping when $|W| <$ `tol`, or until after `term` steps. Deprecated; use `convergeBisect()` instead.

- `linearMatch(double w2, double y0[num_var], double ys[num_var])`: if frequency w2 leads to a vanishing Wronskian, then this will solve for coefficients of independent solutions needed to cause outward and inward solutions to match at the center.

- `int verifyMode()`: a function which classifies the mode $k$ of the eigenfunction based on its nodes using the Osaki-Sculfaire method. Does not need to be called outside of the constructor.

### 5.2.4 Constructors

All constructors begin with

- `basic_setup()`

which initializes everything necessary except for the frequency. The array `rad[]` is populated with the dimensionless radius $x = r/R$ from the `ModeDriver`. Because `Mode<N>` uses RK4, requiring stellar properties at half-steps, the grid for `Mode<N>` is chosen to be double that of `Star`, so that the stellar properties are defined at the half-steps; this avoids inaccuracies from interpolation.

The index `xfit` for matching the shoot-to-center solutions is chosen from the stellar background. In the output graph, the location of `xfit` is marked by a red vertical line – this can help to diagnose failures to converge.

The three constructors differ only in how they find the eigenfrequency $\bar{\omega}$. They are:

- `Mode<num_var>(int k, int l, int m, ModeDriver*)`: creates a mode with anticipated mode numbers $k, l, m$. The starting value of $\omega^2$ is picked based on the Pekeris formula for uniform stellar frequencies (see Cox, section 17.7), [1]

$$D_k = -2 + \Gamma_1[k(l + k + 5/2) + l + 3/2] \tag{5.3a}$$

$$\bar{\omega}_{k,l}^2 = D_k \pm \sqrt{D_k^2 + l(l+1)}. \tag{5.3b}$$

  The actual $k$ is found at the end from `verifyMode()`, and will not necessarily correspond to the desired `k`; ensuring `k` $= k$ takes place outside the class. [2]

- `Mode<num_var>(double w2, int l, int m, ModeDriver*)`: creates a mode with initial guess $\bar{\omega}^2 = $ `w2`. It will then converge to the eigenfrequency closest to the initial guess. The mode number $k$ is found at the end from `verifyMode()`.

- `Mode<num_var>(double omegMin, double omegMax, int l, int m, ModeDriver*)`: creates a mode where the eigenfrequency $\bar{\omega}^2$ is known to fall in the range $\bar{\omega}^2 \in [$`omegMin`, `omegMax`$]$. A bisection search is then used to find the $\bar{\omega}^2$. Note that this range must bound *exactly* one eigenfrequency or the constructor will fail. If the brackets are *bad* (i.e., they enclose multiple roots, or no root), then this merely calls `converge()` to find $\bar{\omega}^2$ using the midpoint as a starting guess.

### 5.2.5 The solution process

The process of finding the solution is handled by `converge()`, and can be broken into two steps handled in two methods:

1. find the eigenfrequency $\bar{\omega}^2$, handled by `convergeBisect()`.

2. find the physical solution $y_1, \ldots, y_N$ for this $\bar{\omega}^2$, handled by `linearMatch()`.

`Mode<N>` uses a shoot-to-center process to calculate the eigenfunctions. Because the system is of order `N`, there will in general be `N` independent solutions; however, there is only one physical solution which satisfies the boundary conditions. At the center, we must have `N`$/2$ boundary conditions [3], which we can write in the form

$$c_{ij}y_j(0) = 0, \qquad i = 1, \ldots N/2 \tag{5.4}$$

---

[1] Note that this guess tends to overestimate $\bar{\omega}^2$ for non-uniform stars, so that the resulting mode will actually have a radial number $k$ much larger than anticipated. To account for this

$$ks = k - 2l + 4$$

is used instead of `k` in (5.3) to better land at anticipated radial mode $k$ (there is no magic to this, it just seemed to work). For g-modes, where $k < 0$, we use an empirical formula based on the data tables in Christensen-Dalsgaard and Mullan 1994.

[2] The reason it takes place outside the class is that the mode found will be an actual eigenmode solution, just not the one we asked for now. Since we may want it later, outside programs may want to store it before trying again.

[3] This exact split is not mathematically necessary, and the code can be modified to accept different divisions of BCs; this will require `ModeDriver` to specify the division by an integer, and to edit the `numvar/2` lines in `linearMatch()` and `RK4center()`.

and at the surface the other $N/2$ boundary conditions, which we can write in the form

$$s_{ij}y_j(1) = 0, \quad i = N/2 + 1, \ldots N. \tag{5.5}$$

The center BC removes $N/2$ degrees of freedom, leaving $N/2$ remaining; these make $N/2$ independent solutions, $\vec{Y}^{(ci)}$; likewise for the surface BCs, leading to $N/2$ independent solutions $\vec{Y}^{(si)}$. The physical solution will be a combination of all $N$ independent solutions that matches all boundary conditions.

This can be most concretely explained by referring to nonradial pulsation modes, as described by `NonradialModeDriver` 12. This is a 4th-order system, so there must be four boundary conditions – two at the center and two at the surface (See Fig. 5.1). Because the system is fourth order, we need four solutions to form a Wronskian. At the center, two of these are taken to be (see section 3.1 of Christensen-Dalsgaard's notes distributed with the `adipls` program, or chapter 6 of his lecture notes)

$$y_1(0) = 0, \qquad y_3(0) = \texttt{y0[2]} \rightarrow \vec{Y}^{(s1)} = (y_1^{(c1)}, y_2^{(c1)}, y_3^{(c1)}, y_4^{(c1)}), \tag{5.6a}$$

$$y_1(0) = \texttt{y0[0]}, y_3(0) = 0 \qquad \rightarrow \vec{Y}^{(s2)} = (y_1^{(c2)}, y_2^{(c2)}, y_3^{(c2)}, y_4^{(c2)}) \tag{5.6b}$$

and at the surface

$$y_1(1) = 0, \qquad y_3(1) = \texttt{ys[2]} \rightarrow \vec{Y}^{(s1)} = (y_1^{(s1)}, y_2^{(s1)}, y_3^{(s1)}, y_4^{(s1)}), \tag{5.6c}$$

$$y_1(1) = \texttt{ys[0]}, y_3(1) = 0 \qquad \rightarrow \vec{Y}^{(s2)} = (y_1^{(s2)}, y_2^{(s2)}, y_3^{(s2)}, y_4^{(s2)}) \tag{5.6d}$$

The condition for these to match is that

$$a\vec{Y}^{(c1)}[\texttt{xfit}] + b\vec{Y}^{(c2)}[\texttt{xfit}] = c\vec{Y}^{(s1)}[\texttt{xfit}] + \vec{Y}^{(s2)}[\texttt{xfit}] \tag{5.7}$$

for some variables $a, b, c$ at the fitting point (the scale of $\vec{Y}^{s2}$ is arbitrarily set to 1). It can be shown that this occurs when the Wronskian $W$ vanishes, where $W$ is calculated as

$$W = \begin{vmatrix} y_1^{(c1)}[\texttt{xfit}] & y_2^{(c1)}[\texttt{xfit}] & y_3^{(c1)}[\texttt{xfit}] & y_4^{(c1)}[\texttt{xfit}] \\ y_1^{(c2)}[\texttt{xfit}] & y_2^{(c2)}[\texttt{xfit}] & y_3^{(c2)}[\texttt{xfit}] & y_4^{(c2)}[\texttt{xfit}] \\ y_1^{(s1)}[\texttt{xfit}] & y_2^{(s1)}[\texttt{xfit}] & y_3^{(s1)}[\texttt{xfit}] & y_4^{(s1)}[\texttt{xfit}] \\ y_1^{(s2)}[\texttt{xfit}] & y_2^{(s2)}[\texttt{xfit}] & y_3^{(s2)}[\texttt{xfit}] & y_4^{(s2)}[\texttt{xfit}] \end{vmatrix}. \tag{5.8}$$

The solution is considered converged when $W = 0$, which means inward and outward solutions are linearly dependent on one another. When $W = 0$, it is *not* the case that inward and outward solutions have the same value at `xfit`, just that the four sets of solutions can be written in terms of each other. For the purposes of finding $\bar{\omega}^2$, it is sufficient to use 0 or 1 for the initial values. It is only necessary to give these specific values when matching the inward and outward solutions for mode counting.

Each constructor specifies an initial guess for $\bar{\omega}^2$. The method `convergeBisect()` then uses bisection to find a value $\bar{\omega}^2$ so that the Wronskian of (5.8) vanishes. The brackets for the bisection search are determined using Newton's method. If $W = 0$, then $\bar{\omega}^2$ is an eigenfrequency.

Once an eigenfrequency $\bar{\omega}^2$ has been discovered, we must find the physical solution with `linearMatch()`. Because $W$ vanishes, we know that (5.7) is satisfied for some choice of $a, b, c$. To find these values, write the homogeneous linear system of (5.7) as a matrix equation,

$$\begin{bmatrix} y_1^{(c1)} & y_1^{(c2)} & -y_1^{(s1)} & -y_1^{(s2)} \\ y_2^{(c1)} & y_2^{(c2)} & -y_2^{(s1)} & -y_2^{(s2)} \\ y_3^{(c1)} & y_3^{(c2)} & -y_3^{(s1)} & -y_3^{(s2)} \\ y_4^{(c1)} & y_4^{(c2)} & -y_4^{(s1)} & -y_4^{(s2)} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{5.9}$$

which can be solved with Gauss-Jordan elimination. Then the values

$$y_1(0) = a \cdot \texttt{y0[0]}, \quad y_3(0) = b \cdot \texttt{y0[2]}, \quad y_1(1) = \texttt{ys[0]}, \quad y_3(1) = c \cdot \texttt{ys[2]} \tag{5.10}$$

are the correct boundary values to use to find the physical solution. Other values are determined from the boundary conditions, (5.4),(5.5). In order for this to work, we *must* have a vanishing Wronskian. If mode solutions are behaving poorly, check the Wronskian values to ensure they are nearly zero.

We have illustrated this for a nonradial Newtonian mode with N=4. For different modes, the process is analogous, but with different boundary conditions and different choices of independent solutions.
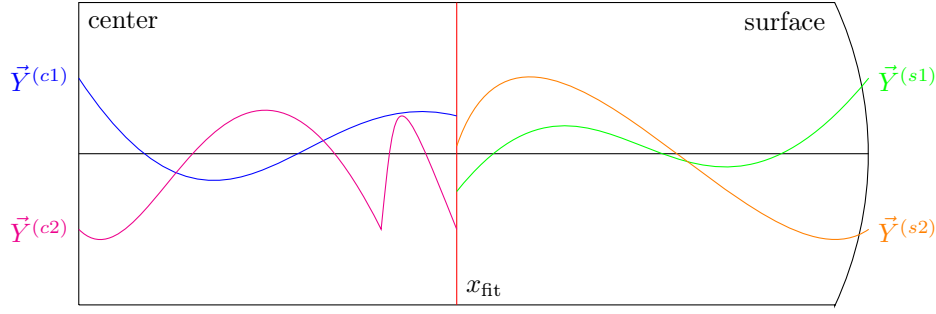
Figure 5.1: A sketch of the solution process for nonradial modes with $\mathtt{N} = 4$ . The two BCs at each boundary give the two solutions in each region, and the values of these four solutions at $x_{\text{fit}}$ are used in the Wronskian for find $\bar{\omega}^2$, then fit together to create the physical solution.

## 5.3   How to extend `Mode<N>`

The class `Mode<N>` is actually a child class of the class `ModeBase` (found in `ModeDriver.h`). To make use of polymorphism with a template class, C++ syntax requires the non-template class `ModeBase` to be the parent. The solution method for `Mode<N>` is deliberately old-fashioned, relying on RK4 integration and other basic methods. To use a more sophisticated method to solve for modes, such as the Henyey method, it is recommended to make a new mode type inheriting directly from `ModeBase` instead of from `Mode<N>`.

## 5.4   A note about editing `Mode<N>`

Because `Mode<N>` is a template class, C++ does not allow a true separation of header/source files, so that the `Mode.h` file includes the `Mode.cpp` file. This means object files which include `Mode.h` will also include the definitions in `Mode.cpp`. The net of this is that whenever you make edits to `Mode.cpp`, you must `make clean` before recompiling.

## 5.5   References

- Scuflaire, R. (1974) and Osaki, Y. (1975); these two papers jointly describe the method to classify mode order and distinguish p and g-modes.

- Christensen-Dalsgaard, J. and D. J. Mullan (1994), for the table of polytrope g-modes used to predict g-mode frequencies.

- Christensen-Dalsgaard, J. (2008), in *Astrophysics and Space Science* for the official publication describing `adipls`. It contains a (much less complete) description of the Wronskian process.

- Christensen-Dalsgaard lecture notes

# Part II

# The White Dwarf Models

# 6      The `SimpleWD` Class
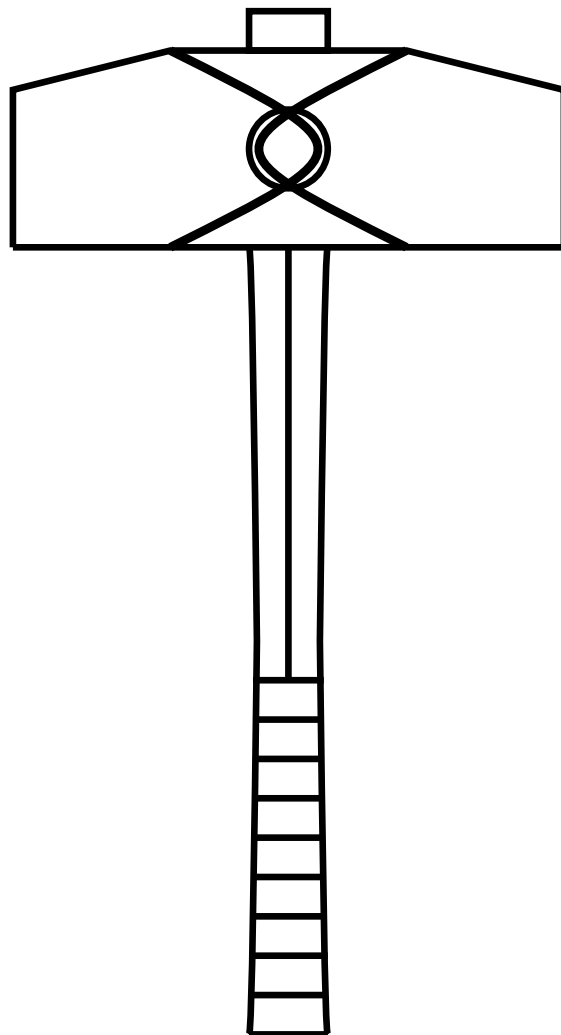
This class fulfills the central purpose of THRAIN, which is the creation of simple, analytic, parametric white dwarf models. The models were first introduced in Boston et al. (2022b) , and then used in Boston et al. (2022c) to form a new WD classification scheme.

The equation of stellar structure can be written

$$\frac{dP}{dr} = -\frac{Gm\rho}{r^2}, \qquad \frac{dm}{dr} = 4\pi r^2 \rho, \qquad \frac{dT}{dr} = -\frac{Gm\rho}{r^2}\frac{T}{P}\nabla, \qquad \frac{dL}{dr} = 4\pi r^2 \rho \epsilon, \qquad (6.1a)$$

where $\nabla$ (often read "del") is a function describing the process of heat transport, $\epsilon$ is a function describing energy production (e.g. through nuclear fusion), and $\rho$ is the density, related to pressure and temperature. All three functions can be written in the form

$$\rho = \rho * P, T, \{X_i\}), \quad \nabla = \nabla(P, T, \{X_i\}), \quad \epsilon = \epsilon(P, T, \{X_i\}), \qquad (6.2a)$$

where $\{X_i\} = \{X, Y, Z_C, Z_O \ldots\}$ is the set of element abundances for elements H, He, C, and O. These equations can be re-written in a dimensionless-logarithmic form due to Schwarzschild (1958),

$$r = R_\star x, \quad m = M_\star \hat{m} \quad L = L_\star \hat{L} \quad P = \frac{GM_\star^2}{4\pi R_\star^4}\hat{P}, \quad T = \mu_S \frac{m_H}{k_B}\frac{GM_\star}{R_\star}\hat{T}, \quad \rho = \frac{M_\star}{4\pi R_\star^3}\hat{\rho}, \quad \epsilon = \frac{L_\star}{M_\star}\hat{\epsilon}, \quad (6.3)$$

so that

$$\frac{d\log\hat{P}}{d\log\hat{m}} = - \quad \exp\{2\log\hat{m} - \log\hat{P} - \log x\} \qquad (6.4a)$$

$$\frac{d\log\hat{T}}{d\log\hat{m}} = -\nabla\exp\{2\log\hat{m} - \log\hat{P} - \log x\} \qquad (6.4b)$$

$$\frac{d\log x}{d\log\hat{m}} = \quad \exp\{\log\hat{m} - \log\hat{\rho} - 3\log x\} \qquad (6.4c)$$

$$\frac{d\log\hat{L}}{d\log\hat{m}} = \quad \epsilon\exp\{\log\hat{m} - \log\hat{L}\}. \qquad (6.4d)$$

This is the form of equation that THRAIN integrates, using a fixed grid in $\hat{m}$, and approximating $\hat{\epsilon} \approx const.$ It is up to the `SimpleWD` class to indicate $\rho, \nabla$ and $\{X_i\}$.

## 6.1    The `SimpleWD` Interface

`SimpleWD` is a child of `Star`, which means it must implement the `Star` interface – i.e., the functions returning $\rho, P, \Phi$, the `printStar()` method, etc. as in 3. It does not define anything beyond `Star`.

### 6.1.1    Constructors

There is only one constructor:

- `SimpleWD(double M, double T, int N)`: will create a WD star with $M_\star = $ `M`, $T_{\text{eff}} = $ `T`, using $N_{\text{grid}} = $ `N`. It is unspoken, but a file `swd.txt` located in the `THRAIN` home directory will be read to provide important parameters to the EOS and element distribution. This file is automatically populated by THRAIN, but must be manually corrected if using `SimpleWD` apart from THRAIN.

## 6.2 The `SimpleWD` Implementation

This class relies on the `stellar` library described in Chapter 7. In particular, it makes use of objects of type

- `Abundance`: represents a set $\{X_i\} = \{X, Y, Z_C, Z_O\}$ of relative elemental abundances.
- `StellarVar`: represents a set $\rho, r, P, m, T, L$ of stellar variables.
- `EOS`: represents a function $P = P(\rho, T, \{X_i\})$ defined within some region of the star.

The `SimpleWD` definition specifies the following class data and methods used in the implementation:

### 6.2.1 Constants

- `int Ntot`: The number of grid points in the WD model.
- `int Ncore`: The number of grid points within the WD 'core.'
- `double Mtot`, `Msolar`: the mass of the star in CGS units, or solar units.
- `double Ltot`, `Lsolar`: the surface luminosity of the star in CGS units, or solar units.
- `double Rtot`, `Rstar`: the radius of the star in CGS units. or in Earth radii.
- `double Teff`: the effective temperature $T_{\text{eff}}$ of the star.
- `double Dscale,Pscale,Tscale`: the scales for density, pressure, temperature, as in (6.3).
- `StellarVar Yscale`, `logYscale`: all scale values in a single `StellarVar` object.
- `StellarVar Ystar`, `Ysolar`: all surface values in a single `StellarVar` object, in either CGS or solar units.
- `zy,zc,zo`: parameters for location of transition (in $z = -\log(1 - \hat{m})$) from He→H, C→He, O→C.
- `by,bc,bo`: parameters for width of transition region from He→H, C→He, O→C.
- `my,mc,mo`: parameters for the maximum value of abundance for He, C, O.

### 6.2.2 Solution Variables

- `double *logQ`: the independent variable, $\log \hat{m}$, which is rendered on a fixed grid.
- `StellarVar *logY`: an array of `StellarVar` objects holding the log-dimensionless solution variables at each grid point.
- `StellarVar *dlogY`: an array of `StellarVar` objects holding the derivatives $\frac{d\log Y}{d\log x}$ at each grid point, which is useful for calculating $U, V_g, A^*$ and other derivatives.
- `Abundance *Xelem`: an array of `Abundance` objects holding the elemental abundances of H,He,C,O at each grid point.
- `Abundance *dXelem`: an array of `Abundance` objects holding the derivatives $\frac{d\log \texttt{Xelem}}{d\log P}$, which is used in calculating the Brünt-Väisälä frequency.
- `Abundance *Xtot`: a `Abundance` object with the total relative abundances of H,He,C,O throughout the star.

### 6.2.3 Physics

The functions below might take $Y = \{\rho, r, P, m, T, L\}$, or $\hat{Y} == \{\hat{\rho}, x, \hat{P}, \hat{m}, \hat{T}, \hat{L}\}$, or log versions, depending on the function. The required form is indicated for each function.

- `EOS core_pressure`: a `EOS` object representing the core EOS function $P = P(\rho, T, \{X_i\})$, illustrated in Figure 6.1. This is used in the majority of the WD.

- `EOS atm_pressure`: a `EOS` object representing the EOS function $P = P(\rho, T, \{X_i\})$ in the outer atmosphere.

- `EOS* getEOS(StellarVar&, Abundance&)`: will determine whether to use the core or atmospheric EOS. Input is $Y$.

- `double equationOfState(StellarVar&, Abundance&)`: will evaluate the EOS, returning $\rho = \rho(P, T, \{X_i\})$ bu inverting the usual EOS in terms of $P(\rho, T, \{X_i\})$. Serious errors can occur at this stage if the temperature does not fall off fast enough, leading to $P_{\rm rad} > P_{\rm total}$, leading to negative pressures for the other components. Input is $\log \hat{Y}$.
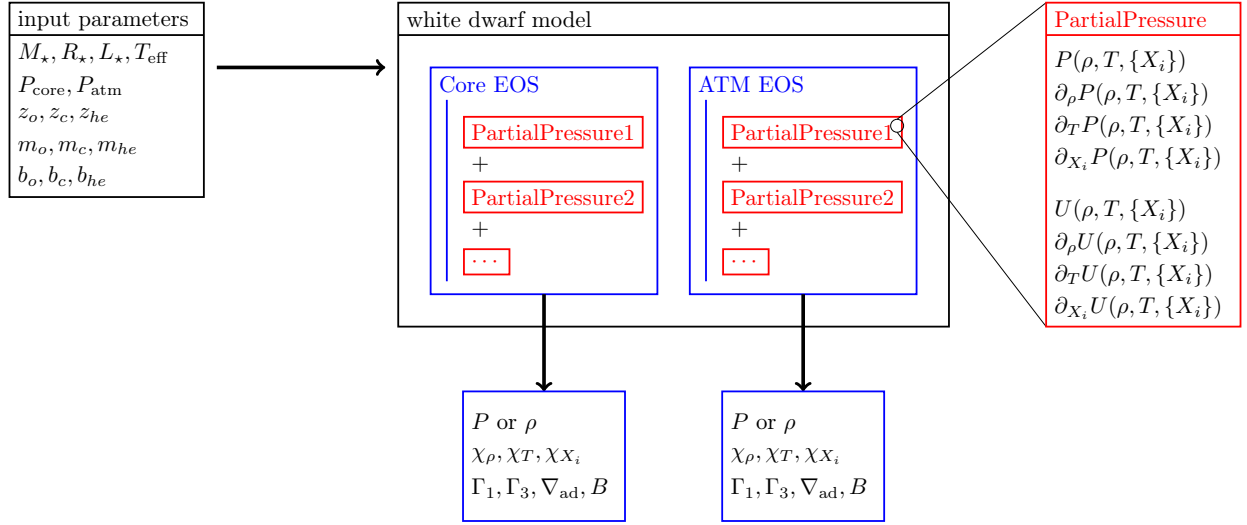


Figure 6.1: A schematic representation of the EOS. The input parameters include the observable surface quantities $(M_\star, R_\star, L_\star, T_{\rm eff})$, the element distribution parameters $(z_i, m_i, b_i)$, and a listing of the pressure terms to use in the core and atmosphere EOS. The two EOS objects (blue boxes) are composed of multiple partial pressure terms (red boxes), which are themselves composed of 8 functions specifying $P, U$ and their derivatives. The EOS uses all partial pressures to compute $P, \rho$, and the thermodynamic derivatives.

- `double energyTransport(StellarVar&, Abundance&)`: returns $\nabla$. First calculates both $\nabla_{\rm rad}$ and $\nabla_{\rm ad}$, and returns the smaller of the two. $\nabla_{\rm ad}$ is computed assuming effective convection. Input is $\log \hat{Y}$.

- `double opacity(StellarVar&, Abundance&)`: the opacity $\kappa = \kappa(P, T, \{X_i\})$, which is used to compute $\nabla_{\rm rad}$. Input is $\log Y$.

- `double energyProduction(StellarVar&, Abundance&)`: the energy production rate $\epsilon = \epsilon(P, T, \{X_i\})$. For the WD models here, this is a constant 1. Input is $\log \hat{Y}$.

- `Abundance findAbundance(double logm, double logr, Abundance& dX)`:
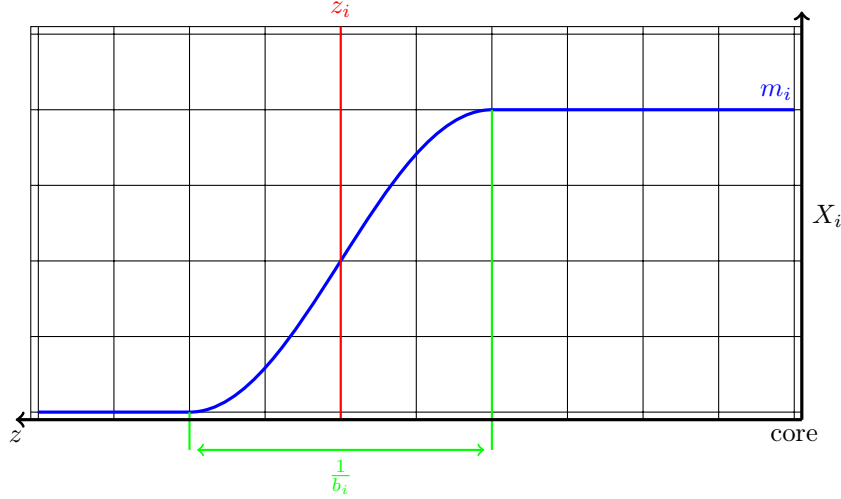
Figure 6.2: An explanation of the parameters in the elemental abundances. For a generic element denoted x (which might be c,o,y), then mx is the maximum value attained, zx is the midway transition point where $X_x = \frac{1}{2}$mx, and bx indicates the breadth of the transition. Both zx,bx are defined in terms of $-\log(1 - \hat{m})$. When four of these curves are combined, they can form much more complicated shapes.

### 6.2.4 Constructor

There is only one constructor:

- `SimpleWD(double M, double T, int N)`

The arguments for $M_\star, T_{\text{eff}}$ must be given in CGS units, and will be converted by `SimpleWD` into the log-dimensionless for in (6.3). For setup, the constructor makes use of the following functions:

- `void setup()`: reads from the `swd.txt` file, which should be in the current directory (i.e. '.').

An example of how to write it is shown at right. The example is also the default set of parameters, which will be set if a valid `swd.txt` file is not found. The keywords following `equation of state` indicate the EOS in the core and the outer atmosphere, which are used to initialize the `core_pressure` and `atm_pressure` objects. The numbers following `chemical parameters` correspond to the constants `zy, zc`, etc. used to describe the chemical composition.

```
# equation of state
core:
rad ideal coul deg_zero
atm:
rad ideal

# chemical parameters
he 10.0 3.0 1.0
c 3.0 3.0 1.0
o 2.0 2.0 0.6
```

- `void initFromChandrasekhar()`: to make sure the convergence routine begins close to a solution, this will create a `ChandrasekharWD` object with the desired mass $M_\star = $ M. The radius of this Chandrasekhar WD is used as a first guess for the radius $R \approx R_\star$, and the central density and pressure are slightly scaled up to be used as initial guesses. Because $R_\star$ must be determined by the mass-radius relation *after* the calculation is finished, we must instead use $R$ as the scale variable in, e.g. (6.3). The variables will be re-scaled by the true $R_\star$ at the end of the calculation when it is known.

- `void setupGrid(double Qcore, int Ncenter)`: creates a grid in $\hat{m}$. Within the "core" (taken to be the first `Ncenter` steps and containing up to $\hat{m} = $ Qcore), the step-size is chosen so that $\Delta x \approx const$ at the center. The outer grid uses a geometric grid, where the step size is chosen to exponentially decrease using the geometric series so that the terminal grid point occurs at $\hat{m} = 1$. This method is adapted

30

from the `ZAMS.f` code by Hansen, Kawaler & Trimble (2004) and distributed with their textbook. A similar grid is also used by `GYRE`.

Once the above setup is performed, the convergence begins. There are three free parameters: the central dimensionless pressure $\hat{P}_0$, the central dimensionless temperature $\hat{T}_0$, and $\delta x = (R_\star - R)/R$ which gives the total stellar radius $R_\star$ in units of $R$. The initial value of $\hat{P}_0$ is chosen to be slightly larger than the initializing Chandrasekhar WD (to account for radiation and gas pressures), the initial value of temperature $\hat{T}_0$ is chosen to be a thousand times larger than $\hat{T}_{\text{eff}}$, and the initial value of $\delta x$ is chosen to be $10^{-3}$. The stellar interior is integrated using (6.4), double-shooting from the center and the surface and matching at `Ncore`, defined as where $\hat{m} = 0.99$. A vector Newton method attempts to zero the mis-match, and ends when the inward and outward solutions agree to 9 digits.

We now call

- `void rescaleR()`: before, all quantities were determined using $R$ form the initializing Chandrasekhar WD as the scale radius. Now that we know the true stellar radius $R_\star$, we rescale all variables so that $\hat{R}_\star = 1$. This requires rescaling all of $\hat{Y}$.

- `void expandGrid(int Ntrue)`: the grid is calculated in $\hat{m}$, but the pulsation code requires a grid in $x$ with points defined at a halfway point between `x[i]` and `x[i+1]`. We therefore compute the above stellar model on a grid half the size as `Ntrue`, and at this stage we expand it to the true size `Ntrue` with the new points `x[i]` $= 0.5($`x[i+1]`$+$`x[i-1]`$)$. We find the values of $\log \hat{Y}$ at the halfway points by integrating to `x[i]` from `x[i+1]` and `x[i-1]` with simple Euler steps, then averaging the two results. See Fig. 11.1.



Figure 6.3: The circles labeled represent the initial grid points, which are unevenly spaced in $x$. The squares labeled are the half-grid points needed by the modes' RK4 algorithms. These new points are chosen to lie at the midpoint in $x$ between two circle points.

The final step is to calculate the Brünt-Väisälä frequency $N^2$ and other important thermodynamic properties. This is handles by many of the methods in `EOS`.

## 6.2.5   Calculation

- `int numv=3`: the number of variables in the convergence routine ($\hat{P}_0, \hat{T}_0, \delta x$).

- `void joinAtCenter(double x[numv], oduble f[numv], double &F)`: calls the below methods to integrate from the center and the surface to the matching point, using values $\hat{P}_0 =$ `x[0]`, $\hat{T}_0 =$ `x[1]`, $\delta x =$ `x[2]`. The differences in $\hat{P}, \hat{T}$, and $x$ at the matching point are stored in `f[]`, which the convergence routine tries to zero. The input `F` is use to return `F` $= \frac{1}{2} \sum_{\texttt{i}}^{\texttt{numv}}$ `f[i]`$^2$, which is used to throttle the convergence routine.

- `double calculateCore(double x[numv], int Nmax)`: integrates (6.4) from the center outward to the matching point, using standard RK4 and $\rho, \nabla, \epsilon$ as defined above.

- `double calculateAtmosphere(double x[numv], int Nmin)`: integrates (6.4) from the surface inward to the matching point, using standard RK4 and $\rho, \nabla, \epsilon$ as defined above.

- `int firstCoreStep(double x[numv], double& rholast, int Nmax)`: at the very center, (6.4) is singular. To avoid pathological behavior, we integrate the very first step in variable $x$ using a dimensionless version of (6.1), with $\Delta x$ chosen to correspond to $\Delta \hat{m} = \frac{1}{3}\rho_0 \Delta x^3$.

- `int firstAtmosphereStep(double x[numv], double& rholast)`: the very first step from the stellar surface inward, and perhaps the most complex part of the calculation.

The functions

- `dYdR()`

- `dlogYdlogR()`

- `dYdM()`

- `dlogYdlogM()`

implement different versions of the equations of stellar equilibrium, as indicated by their names. These derivatives are returned as a `StellarVar` object.

### 6.2.6 The Atmosphere

Defining exactly what is meant by "the surface" of a real star is a sensitive issue. If the surface is defined as the location where $\rho = 0$, then the surface of our Sun technically extends well beyond the usual bound of the solar system. This is probably not what we meant by "surface."

We choose to define the surface as the photosphere, the location where $T(r) = T_{\text{eff}}$. The photosphere is also called the surface of last scatter, as it is the location where photons trapped within the star are finally able to freely escape into space. This corresponds to the visible border of the Sun.

### 6.2.7 Boundary conditions for modes

In order to implement the boundary conditions of modes, it is necessary to have power series expressions for $A^*, U, c_1, V_g$. This is more difficult to do for the complicated physical processes taking place inside a white dwarf star.

- `setupCenter()`: prepares coefficients $\theta_k$ for an approximate polytrope method, based on $n = \frac{1}{\Gamma_1 - 1}$ with $\Gamma_1 = $ `Gam1(0)`.

- `setupSurface()`: en empty method. Only for conformity.

- `getXYZCenter(double* xyz, int& maxPow)`: where XYZ is `Astar`, `Vg`, `U`, `C1` Returns terms of the power series expansion in $x = r/R$ near the center, which have been borrowed from `Polytrope` (see 9). The coefficients are stored in the `xyz[]` array to be returned. If more are requested with `maxPow`, then we set `maxPow` $= 4$ so that the `ModeDriver` only implements the BC to order $x^4$.

- `getXYZSurface(double* xyz, int& maxPow)`: as above, but near the surface expanded in powers of $t = 1 - r/R$, using the surface condition for an $n = 3/2$ polytrope (see 9).

Within the directory `output/calc_name/star/wave_coefficient`, there are several files for testing the goodness of these power series expansions. The coefficients of the expansion are printed to files `center.txt,surface.txt`. The first few points of $\rho, P, A^*, U, V_g, c_1$ are printed, from the stellar data and using the expansion function, for comparison, and the errors printed in the corresponding `.png` files for easier viewing.

## 6.3 Limitations

The `SSR()` method used for modes assumes a uniform grid spacing when calculating derivatives; for this reason, mode error should be estimated using $c_0$.

For hot WD stars in the instability strip, for some reason many modes will fail to calculate, and errors are high. Cold stars below the instability strip seem to calculate well. These numerical problems with DAVs were a primary motivation for the `SimpleWD` class.

It is suggested to double-check the graphs of each mode. The central red vertical line shows the fitting point of the double-shooting method; ensure regular behavior around this line. The initial guesses used in constructing g-mode frequencies within `Mode<N>`, based on the core g-modes of polytropes, is probably very unsuited to the surface g-modes of DAVs.

## 6.4 Internal Logic Behind the Interface

With the above variables, we can now define how the following `Star` quantities are computed:

- `double rad(int X)`: $r = \texttt{Rstar} * \exp\{\texttt{logY[X][radi]}\}$.

- `double rho(int X)`: $\rho = \texttt{Dscale} * \exp\{\texttt{logY[X][dens]}\}$.

- `double drhodr(int X)`: $\frac{d\rho}{dr} = \texttt{Dscale/Rstar} * \exp\{\texttt{logY[X][dens]} - \texttt{logY[X][radi]}\} * \texttt{dlogY[X][dens]}$.

- `double P(int X)`: $P = \texttt{Pscale} * \exp\{\texttt{logY[X][pres]}\}$.

- `double dPdr(int X)`: $\frac{dP}{dr} = \texttt{Pscale/Rstar} * \exp\{\texttt{logY[X][pres]} - \texttt{logY[X][radi]}\} * \texttt{dlogY[X][pres]}$

- `double Phi(int X)`: <span style="color:red">Not yet implemented (01/22/2022) .</span>

- `double dPhidr(int X)`: $g = \frac{d\Phi}{dr} = G\texttt{Mstar/Rstar}^2 * \exp\{\texttt{logY[X][mass]} - 2\texttt{logY[X][radi]}\}$.

- `double mr(int X)`: $m(r) = \texttt{Mtot} * \texttt{mass(X)}$.

- `double Schwarzschild_A(int X, double GamPert)`: If `GamPert=0`, then

$$A = -\frac{N^2}{g} = -\hat{N}^2 \exp\{2\texttt{logY[X][radi]} - \texttt{logY[X][mass]}\}/\texttt{Rstar}.$$

  Otherwise,

$$A = (\texttt{dlogY[X][dens]} - \texttt{dlogY[X][pres]/GamPert}) * \exp\{-\texttt{logY[X][radi]}\}/\texttt{Rstar}.$$

  (Note units! $A^*$ is dimensionless, but Schwarzschild discriminant is not.)

- `double getAstar(int X, double GamPert)`: If `GamPert=0`, then

$$A^* = -\hat{N}^2 \exp\{3\texttt{logY[X][radi]} - \texttt{logY[X][mass]}\}.$$

  Otherwise,

$$A^* = \texttt{dlogY[X][pres]/GamPert} - \texttt{dlogY[X][dens]}.$$

- `double getVg(int X, double GamPert)`: If `GamPert=0`, then

$$V_g = \texttt{logY[X][pres]}/\Gamma_1.$$

  Otherwise

$$V_g = \texttt{logY[X][pres]/GamPert}.$$

- `double getU(int X)`: $U = \texttt{dlogY[X][mass]}$.

- `double getC(int X)`: $c_1(r) = \exp\{3\texttt{logY[X][radi]} - \texttt{logY[X][mass]}\}$.

- `double sound_speed2(int X, double GamPert)`: $v_s^2 = \frac{\Gamma_1 P}{\rho}$. If `GamPert=0`,

$$v_s^2 = \texttt{Gamma1(X)*pres(X)} * \exp\{\texttt{logY[X][pres]} - \texttt{logY[X][dens]}\} * \texttt{Pscale/Dscale}.$$

  Otherwise,

$$v_s^2 = \texttt{GamPert*pres(X)} * \exp\{\texttt{logY[X][pres]} - \texttt{logY[X][dens]}\} * \texttt{Pscale/Dscale}.$$

The other methods, `length()`, `Radius()`, etc., are trivially defined.

## 6.5   References

- Paxton et al. (2010), the original paper introducing `MESA` to the world, describing the physics and numerical methods used within the code.

- Schwarzschild (1958), *Structure and Evolution of the Stars*, available from Dover, as a reference for the dimensionless variables chosen here and also some of the equations for the microphysics.

- Fuller & Lai (2011), for an explanation of the $c_0$ column (and $Q_\alpha$ column) of `tidal_overlap.txt`. See especially their Table 1.

- Lamb & van Horn (1975)

- Matt Wood (1990)

- Kim & Montgomery (2018)

- Hansen, Kawaler & Trimble (2004)

- Cox & Giuli (1980)

- Kippenhahn, Weigert & Hofmeister (1967), *Meth. Comp. Phys.*, 7,129

# 7      The Stellar Library

stellar is a library file, which is essential to creating the WD models of SimpleWD. It will also be useful generally to other realistic stellar model types in future updates to this program. This chapter documents the elements of stellar and how they are used by THRAIN.

## 7.1   chemical

This is a namespace containing important functions when dealing with element abundances and this impact on other formulae.

- enum elem: a list of the elements used in models. There are four: hydrogen h, helium he, carbon c, and oxygen o. These values, accessed as, e.g. chemical::h, are used as replacement indices for chemical values. They are otherwise identical to integers 0 through 3.

- double Z[]: the number of protons in each element

- double A[]: the mass number of each element. Standard isotopes are assumed.

This namespace also defines functions for partial derivatives of atomic average quantities which appear in certain pressure terms. When a pressure term depends on the elemental abundance, it will depend on it through some averaged quantity $\mu$, so that $P(\rho, T, \{X\}) = P(\rho, T, \mu(\{X\}))$. Therefore

$$\frac{\partial P}{\partial X_{\mathtt{i}}} = \frac{\partial P}{\partial \mu} \frac{\partial \mu}{\partial X_{\mathtt{i}}}. \tag{7.1}$$

For any average quantity $\mu$, due to the constraint $\sum X_i = 1$ we can isolate one element e to be excluded from the summation, and the average can be defined as

$$\mu(\{X\}) = g\left(\sum f(i)X_i\right) = g\left(\sum_{i \neq \mathtt{e}} [f(i) - f(\mathtt{e})]\, X_i + f(\mathtt{e})\right), \tag{7.2}$$

where $f(i)$ is an atomic property being averaged over and $g$ is some possible function of the average of $f(i)$. Therefore, for any average $\mu$ and element i,

$$\frac{\partial \mu}{\partial X_{\mathtt{i}}} = [f(\mathtt{i}) - f(\mathtt{e})] \cdot g'\left(\sum f(i)X_i\right). \tag{7.3}$$

With this in mind, we can now define the following functions which appear in chemical:

- partial_mean_A(elem i, Abundance X): computes $\frac{\partial \langle A \rangle}{\partial X_{\mathtt{i}}}$, where $f(i) = 1/A_i$ and $g(f) = 1/f$. Therefore

$$\frac{\partial \langle A \rangle}{\partial X_{\mathtt{i}}} = \left[\frac{1}{A_{\mathtt{i}}} - \frac{1}{A_{\mathtt{e}}}\right]\left(-\langle A \rangle^2\right). \tag{7.4}$$

- `partial_mean_Z(elem i, Abundance X)`: computes $\frac{\partial \langle Z \rangle}{\partial X_{\text{i}}}$, where $f(i) = Z_i$ and $g(f) = f$. Therefore

$$\frac{\partial \langle Z \rangle}{\partial X_{\text{i}}} = Z_{\text{i}} - Z_{\text{e}}. \tag{7.5}$$

- `partial_mu_e(elem i, Abundance X)`: computes $\frac{\partial \mu_e}{\partial X_{\text{i}}}$, where $f(i) = Z_i/A_i$ and $g(f) = 1/f$. Therefore

$$\frac{\partial \mu_e}{\partial X_{\text{i}}} = \left[ \frac{Z_{\text{i}}}{A_{\text{i}}} - \frac{Z_{\text{e}}}{A_{\text{e}}} \right] \left( -\mu_e^2 \right). \tag{7.6}$$

- `partial_mean_coul(elem i, Abundance X)`: computes $\frac{\partial \langle Z^2/A^{1/3} \rangle}{\partial X_{\text{i}}}$, where $f(i) = Z_i/A_i^{1/3}$ and $g(f) = f$. Therefore

$$\frac{\partial \langle Z^2/A^{1/3} \rangle}{\partial X_{\text{i}}} = \frac{Z_{\text{i}}^2}{A_{\text{i}}^{1/3}} - \frac{Z_{\text{e}}^2}{A_{\text{e}}^{1/3}}. \tag{7.7}$$

## 7.2 Abundance

This is a class (technically a structure) represent an elemental abundance (in a given region). It could be used for the star as a whole giving the total relative abundances, or at each radius $r$ within the star giving a local abundance.

- `double H1, He4, C12, O16`: these hold the relative abundance of each of the elements: hydrogen, helium, carbon, and oxygen. For a `Abundance` called `X`, these values can be accessed in the following ways:

  - `X.H1` to get hydrogen, `X.He4` to get helium, etc.
  - `X[0]` to get hydrogen, `X[1]` to get helium, etc.
  - `X[chemical::h]` to get hydrogen, `X[chemical::he]` to get helium, etc.

- `int e`: an element to exclude from summations over all elements. This is here due to the constraint $\sum_i X_i = 1$ which fixes ones element, so that that element may be dropped in certain situations.

- `double mean_A()`: the mean atomic mass number $\langle A \rangle$, which is computed with a geometric mean,

$$\frac{1}{\langle A \rangle} = \sum \frac{X_i}{A_i}. \tag{7.8}$$

  In the literature, this is sometimes also represented as $\langle A \rangle = \mu_A$ or $\mu_i$.

- `double mean_Z()`: the mean charge number $\langle Z \rangle$, which is computed as an arithmetic mean,

$$\langle Z \rangle = \sum X_i Z_i. \tag{7.9}$$

- `double mu_e()`: the mean atomic mass per electron $\mu_e$, which is computed as an arithmetic mean,

$$\frac{1}{\mu_e} = \sum X_i \frac{A_i}{Z_i}. \tag{7.10}$$

  The optional argument `y` indicates the ionization fraction, otherwise assumed to be 1 (full ionization).

- `mean_coulomb()`: an average which appears in calculations of the Coulomb pressure,

$$\left\langle \frac{Z^2}{A^{1/3}} \right\rangle = \sum X_i \frac{Z_i^2}{A_i^{1/3}}. \tag{7.11}$$

- `Abundance operator*`: multiple all abundances by a scalar quantity. The abundance must appear first in the order of operations.

- `Abundance operator+`: add two abundances together, element-wise.

- `Abundance operator-`: subtract two abundances, element-wise.

- `void enforce()`: force elemental abundances to obey constraints, $X_i > 0$ and $\sum X_i = 1$. Any negatives are set to zero, then all abundances are rescaled to sum to 1.

## 7.3  `StellarVar`

This is a class, similar to `Abundance`, but instead representing a set of stellar variables $\{\rho, r, P, m, T, L\}$ which may be defined at some region in a star. It could be used for the star as a whole, for instance to give observable values of $R_\star, M_\star, L_\star, T_{\text{eff}}$ in a single object; or it could be specified at each radius $r$ giving local values; or it can be used to hold scale values for each quantity. These values could be dimensionful, could be dimensionless analogs (i.e. $x, \hat{m}$, etc.), or could be log variables (i.e. $\log x, \log \hat{m}$, etc.).

- `enum VarName`: a list of the stellar variables used in the models. They are $\text{dens} = \rho$, $\text{radi} = r$, $\text{pres} = P$, $\text{mass} = m$, $\text{temp} = T$, $\text{lumi} = L$. The last element `num_var` gives the total count of elements, and is useful in `for`-loops or other situations.

- `double var[num_var]`: the array holding the actual values of the variables. Assuming a `StellarVar` named `Y`, these can be accessed as either:

    - `Y[0]` for $\rho$, `Y[1]` for $r$, `Y[2]` for $P$, etc.
    - `Y[dens]` for $\rho$, `Y[radi]` for $r$, `Y[pres]` for $P$, etc.

    The accessor methods are defined in this class.

- `operator*(double)`: scalar multiplication, multiply each variable by the given amount.

- `operator/(double)`: scalar division, divide each variable by the given number.

- `operator+(StellarVar)`: add two `StellarVar` objects together, element-wise. Useful in numerical integration methods.

- `operator-(StellarVar)`: subtraction two `StellarVar` objects, element-wise.

- `operator*(StellarVar)`: multiply two `StellarVar` objects, element-wise. Useful for rescaling variables.

- `operator/(StellarVar)`: divide two `StellarVar` objects, element-wise. Useful for rescaling variables.

Outside of this class are also defined exponential and logarithm methods for taking exponentials, so that `StellarVar` objects can be easily shifted from logarithmic to linear versions.

## 7.4  EOS

This class represent an equation of state within a region. It consists of a vector of partial pressure objects, and operations for computing needed quantities based on the equation of state.

## 7.5  `PartialPressure`

A single pressure term representing an equation of the form $P = P(\rho, T, \{X_i\})$. Associated with each partial pressure is a partial internal energy $U = U(\rho, T, \{X_i\})$. Each `PartialPressure` object consists of function pointers to functions describing these equations, and also needed partial derivatives. They are:

`funcptr P` : the function $P = P(\rho, T, \{X_i\})$, which can be simply accessed as, e.g, `myPressure(rho,T,X)`, with `rho` a double, `T` a double, and `X` an `Abundance` object.

`funcptr partialRho` : the partial derivative $\partial P/\partial \rho$

`funcptr partialT` : the partial derivative $\partial P/\partial T$

`(*partialX)(chemical::elem e, double,double,Abundance)` : the partial derivative $\partial P/\partial X_{\mathrm{e}}$, relative to the particular element $X_{\mathrm{e}}$, either hydrogen, helium, carbon, or oxygen.

The same operations are then repeated for $U$. Note that only pressure may be simply accessed with `()` on the `PartialPressure` object, and others must be called explicitly. The type `funcptr` is defined as a pointer to any function accepting `(double, double, Abundance)`, where the first double is pressure, then temperature.

## 7.6 The chandra library

The electron degeneracy pressures (and also the `ChandrasekharWD` models) sometimes depend on functions or constants defined within the `chandra` library. There are two namespaces here, `Chandrasekhar` and `FermiDirac`.

The `Chandrasekhar` namespace contains the following:

`double A0` : constant factor $A = \frac{\pi m_e^4 c^5}{3h^3}$ which appears in degenerate pressure equation. The value listed here is the 1939 value used by Chandrasekhar to make tabulations.

`double B0` : constant factor $B = \frac{8\pi m_H}{3}\left(\frac{m_e c}{h}\right)^3$ which appears in the degenerate density equation. In the literature this value is often defined to include $\mu_e$; we do not do this, as $\mu_e$ will change within the flkuid. The value listed here is the 1939 value used by Chandrasekhar to make tabulations, not the modern value.

`double factor_f(double x)` : the function $f$ appearing in $P_{\mathrm{deg}} = Af(x)$, defined

`double factor_g(double x)` : the function $g$ appearing in $U_{\mathrm{deg}} = Ag(x)$.

The two functions are defined as

$$f(x) = (2x^3 - 3x)\sqrt{1 + x^2} + 3\sinh^{-1}x, \tag{7.12a}$$

$$g(x) = 8x^3(\sqrt{x^2 + 1} - 1) - f(x). \tag{7.12b}$$

The `FermiDirac` namespace is largely aimed at evaluating the Fermi-Dirac functions

$$F_k(\eta, \beta) = \int_0^\infty \frac{x^k}{1 + e^{x-\eta}}\sqrt{1 + \tfrac{1}{2}\beta x}, \tag{7.13}$$

using various approximations or numerical techniques.

`static const double xFermiDiracXHalf[]` : one of several sets of numerical constants for use in calculating Fermi-Dirac fuctions $F_{\mathrm{X}/2}$, with $\mathrm{X} = 1, 3, 5$. These values are abscissae used in a 24-point Guass quadrature method for integrating the $F_{\mathrm{X}/2}$ functions. The values are due for Sagar (1991).

`static const double wFermiDiracXHalf[]` : as above, but the weights used to calculate $F_{\mathrm{X}/2}$.

`double FermiDiracXHalf(double eta, double beta)` : quadrature integration of $F_{\mathrm{X}/2}$, defined by Sagar (1991) as

$$F_{\mathrm{X}/2}(\eta, \beta) = \sum_k w_k \frac{1 + e^{x_k}}{1 + e^{x_k - \eta}}\sqrt{1 + \tfrac{1}{2}\beta x_k}, \tag{7.14}$$

for $\mathrm{X} = 1, 3, 5$.

`double FermiDirac(double k, double eta, double beta)` : directly integrates the Fermi-Dirac functions $F_k(\eta, \beta)$ using trapezoidal integration.

The other functions define various approximations to $F_k(\eta, \beta)$ in different limits. See Cox & Giuli (1980) or Pichon (1989).

# 8       The Partial Pressures

Within `stellar` are defined several partial pressure with their partial derivatives as `PartialPressure` objects. Here we will list out and explain the available partial pressures.

## 8.1   `PartialPressure`

A single pressure term representing an equation of the form $P = P(\rho, T, \{X_i\})$. Associated with each partial pressure is a partial internal energy $U = U(\rho, T, \{X_i\})$. Each `PartialPressure` object consists of function pointers to functions describing these equations, and also needed partial derivatives. They are:

`funcptr P` : the function $P = P(\rho, T, \{X_i\})$, which can be simply accessed as, e.g, `myPressure(rho,T,X)`, with `rho` a double, `T` a double, and `X` an `Abundance` object.

`funcptr partialRho` : the partial derivative $\partial P / \partial \rho$

`funcptr partialT` : the partial derivative $\partial P / \partial T$

`(*partialX)(chemical::elem e, double,double,Abundance)` : the partial derivative $\partial P / \partial X_\mathrm{e}$, relative to the particular element $X_\mathrm{e}$, either hydrogen, helium, carbon, or oxygen.

The same operations are then repeated for $U$. Note that only pressure may be simply accessed with `()` on the `PartialPressure` object, and others must be called explicitly. The type `funcptr` is defined as a pointer to any function accepting `(double, double, Abundance)`, where the first double is pressure, then temperature.

## 8.2   How to Define New Pressures

## 8.3   Ideal Gas Pressure, `ideal`

This partial pressure term follows the ideal gas law. The pressure is

$$P_\mathrm{ideal} = \frac{N_A k_B}{\langle A \rangle} \rho T, \tag{8.1}$$

where $\langle A \rangle = $ `X.mean_A()`, $N_A$ is Avogadro's number, and $k_B$ is the Boltzmann constant. For an ideal gas, the pressure and internal energy are related by

$$U_\mathrm{ideal} = \frac{3}{2} P_\mathrm{ideal}. \tag{8.2}$$

The partial derivatives with respect to $\rho, T$ are obvious. With respect to $X_i$, the partial derivative is

$$\frac{\partial P_\mathrm{ideal}}{\partial X_\mathrm{i}} = -\frac{N_A k_B}{\langle A \rangle^2} \frac{\partial \langle A \rangle}{\partial X_\mathrm{i}}, \tag{8.3}$$

using

$$\frac{\partial \langle A \rangle}{\partial X_\mathrm{i}} = \texttt{chemical::partial\_mean\_A(i, X)}. \tag{8.4}$$

The partial derivatives of $U$ are simply related to partial derivatives of $P$ by the same factor $\frac{3}{2}$.

## 8.4 Radiation Pressure, `rad_gas`

This follows the radiation pressure law. The pressure is

$$P_{\mathrm{rad}} = \frac{a}{3}T^4, \tag{8.5}$$

where $a = 7.5657 \times 10^{-15} \mathrm{erg} \cdot \mathrm{cm}^{-3} \mathrm{K}^{-4}$ is the radiation constant, which is related to the Stefan-Boltzmann constant $\sigma$ by $a = 4\sigma/c$. For a photon gas, the pressure and internal energy are related by

$$U_{\mathrm{rad}} = 3P_{\mathrm{rad}}. \tag{8.6}$$

Neither $P_{\mathrm{rad}}$ nor $U_{\mathrm{rad}}$ depends on $\rho$ or $X_{\mathrm{i}}$, and partial derivatives with respect to $T$ are obvious.

## 8.5 Coulomb Repulsion, `coul`

When the gas is ionized, there will result a pressure resulting from ion-ion repulsion and ion-electron attraction due to the Coulomb interaction. Modeling this can be complicated. We will use an approximate formula due to Koester (1976, eq. 16; see also Tassoul et al. 1990, eq. 18),

$$P_{\mathrm{coul}} = \frac{-0.3\Gamma_C^{3/2}}{\Gamma_C^{1/2} + 1.03921} \times P_{\mathrm{ideal}} = f_C P_{\mathrm{ideal}}, \tag{8.7}$$

with $P_{\mathrm{ideal}}$ as above, and with $\Gamma_C$ defined

$$\Gamma_C(\rho, T, \{X\}) = 2.275 \times 10^5 \left\langle \frac{Z^2}{A^{1/3}} \right\rangle \frac{\rho^{1/3}}{T}. \tag{8.8}$$

Any partial derivatives pf $P_{\mathrm{coul}}$ will require partially differentiating $\Gamma_C$. The partial derivatives are

$$\frac{\partial P_{\mathrm{coul}}}{\partial \rho} = f_C \frac{\partial P_{\mathrm{ideal}}}{\partial \rho} + P_{\mathrm{ideal}} \frac{df_C}{d\Gamma_C} \frac{\partial \Gamma_C}{\partial \rho}, \tag{8.9a}$$

$$\frac{\partial P_{\mathrm{coul}}}{\partial T} = f_C \frac{\partial P_{\mathrm{ideal}}}{\partial T} + P_{\mathrm{ideal}} \frac{df_C}{d\Gamma_C} \frac{\partial \Gamma_C}{\partial T}, \tag{8.9b}$$

$$\frac{\partial P_{\mathrm{coul}}}{\partial X_{\mathrm{i}}} = f_C \frac{\partial P_{\mathrm{ideal}}}{\partial X_{\mathrm{i}}} + P_{\mathrm{ideal}} \frac{df_C}{d\Gamma_C} \frac{\partial \Gamma_C}{\partial X_{\mathrm{i}}}, \tag{8.9c}$$

and

$$\frac{df_C}{d\Gamma_C} = \frac{-0.45\Gamma_C^{1/2}}{\Gamma_C^{1/2} + 1.03921} + \frac{0.15\Gamma_C}{(\Gamma_C^{3/2} + 1.03921)^2} \tag{8.10a}$$

$$\frac{\partial \Gamma_C}{\partial \rho} = 2.275 \times 10^5 \left\langle \frac{Z^2}{A^{1/3}} \right\rangle \frac{1}{3} \frac{\rho^{-2/3}}{T} \tag{8.10b}$$

$$\frac{\partial \Gamma_C}{\partial T} = 2.275 \times 10^5 \left\langle \frac{Z^2}{A^{1/3}} \right\rangle \left( -\frac{\rho^{1/3}}{T^2} \right) \tag{8.10c}$$

$$\frac{\partial \Gamma_C}{\partial X_{\mathrm{i}}} = 2.275 \times 10^5 \frac{1}{3} \frac{\rho^{-2/3}}{T} \texttt{chemical::partial\_mean\_coul(i,X)}. \tag{8.10d}$$

Because this force is due to the Coulomb interaction as radiation pressure, it also has $U_{\mathrm{coul}} = 3P_{\mathrm{coul}}$, and the same scaling holds for all partial derivatives of $U_{\mathrm{coul}}$.

## 8.6 Fully Degenerate ($T = 0$) Electron Pressure, `deg_zero`

Under extreme gravitation, the gas is prevented from further collapse by the fact that only two electrons may occupy the same box of phase space. In the limit of zero temperature, we approximate all electrons

as occupying the lowest available quantum levels up to some maximum defined by Fermi momentum $p_F$, or a dimensionless version $x = p_F/m_e c$. Using statistical mechanics, we find the electron degeneracy pressure and baryon density to be (see Chandrasekhar 1935)

$$\rho_{\text{baryons}} = B\mu_e x^3, \tag{8.11a}$$

$$P_{\text{deg},0} = \frac{\pi m_e^4 c^5}{3h^3} \left[ (2x^3 - 3x)\sqrt{1 + x^2} + 3\sinh^{-1} x \right] = Af(x). \tag{8.11b}$$

where $\mu_e$ relates the atomic mass in the baryon density to the number of electrons. Given a density $\rho$, we find $x = (\rho/B/\mu_e)^{1/3}$, which is then used to compute $P_{\text{deg},0}$.

Unlike the previous pressure terms, there is no simple relation between $P_{\text{deg},0}$ and $U_{\text{deg},0}$. Instead, $U_{\text{deg},0}$ must be calculated using $U = Ag(x)$ as defined in the chandra library.
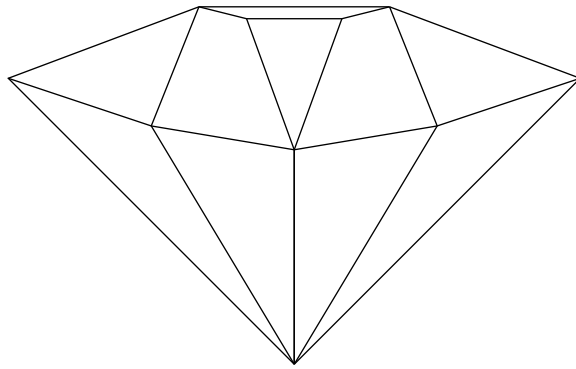
## 8.7   Partially Degenerate Electron Pressure, deg_finite

## 8.8   Arbitrarily Degenerate Electron Pressure, deg_trap

This uses trapezoidal integration to directly integrate $F_k(\eta, \beta) = $ FermiDirac(k,eta,beta).

# Part III

# Simple Stellar Models

# 9      The Polytrope Class

This class represents a pure polytropic star with given index $n$. Polytropes are described by the Lane-Emden equation,

$$\frac{1}{\xi^2}\frac{d}{d\xi}\left(\xi^2\frac{d\theta}{d\xi}\right) = -\theta^n, \tag{9.1}$$

where the function $\theta(x)$[1] is related to density by

$$\rho(r) = \rho_c\theta(r)^n \tag{9.2}$$

and where

$$\xi = r\sqrt{\frac{4\pi G\rho_c^2}{(n+1)P_c}}. \tag{9.3}$$

The boundary conditions are that $\theta = 1$ at the center $\xi = 0$, and that $\theta = 0$ at the surface $\xi = \xi_1$.

## 9.1    The `Polytrope` Interface

`Polytrope` is a child of `Star`, which means it must implement the `Star` interface – i.e., the functions returning $\rho, P, \Phi$, the `printStar()` method, etc. as in 3.

In addition, `Polytrope` offers the following methods accessible to external code:

- `double getX(int k)`: return position variable $\xi = $ `x[k]`.

- `double getY(int k)`: return solution $\theta(\xi[k]) = $ `y[k]`.

- `double getYderiv(int k)`: return derivative $\frac{d\theta}{d\xi}|_{\xi[k]} = $ `z[k]`.

These are mostly used for testing purposes, such as comparing to known analytic equations or to relativistic polytropes.

### 9.1.1    Constructors

There are three constructors available for a `Polytrope`:

- `Polytrope(double M, double R, double n, int len)`: creates a polytrope of specific grid size (length) `len`, of polytropic index $n = $ `n`, with total mass $M_\star = $ `M` and $R_\star = $ `R`. The step size `dx` needed to reach the surface $(y = 0)$ in exactly `len` steps is found by bisection.

- `Polytrope(double n, int len)`: as above, but maintaining dimensionless variables where $G = \rho_c = P_c = 1$.

- `Polytrope(double n, int len, double dx)`: as above, but specifies the step size to use as `dx`. There is no guarantee of reaching the "true" surface. This is used for testing scaling relations. Do not use for production runs.

---

[1]Within the code we use ASCII-friendly characters $\xi = x$ and $\theta = y$

## 9.2 The `Polytrope` Implementation

The `Polytrope` definition specifies the following class data and methods used in the implementation:

### 9.2.1 Constants

- `int len`: the length of arrays for this calculation, which is returned by `length()` function.

- `double n`: the polytropic index $n$.

- `double Gamma`: the adiabatic exponent $\Gamma$ of the polytrope, $\Gamma = 1 + \frac{1}{n}$, where $P \sim \rho^{\Gamma}$.

- `double rho0`: the central density $\rho_c$.

- `double P0`: the central pressure $P_c$.

- `double Rn`: the scale factor $R_n = \sqrt{\frac{(n+1)P_c}{4\pi G \rho_c^2}}$ with $r = R_n \xi$.

### 9.2.2 Solution Variables

- `double **Y`: an array holding the solution to (9.1) in the form $(\xi, \theta, \frac{d\theta}{d\xi})$. The first index is the grid point `X`.

- `enum VarName {x=0, y, z, numvar}`: indices for solution variables within `**Y`, where $x \to \xi, y \to \theta, z \to \frac{d\theta}{d\xi}$, and where `numvar` is used to indicate the number of variables.

- `double *mass`: an array containing the interior mass $m(r)$ as a function of radius, $m(r) = \int_0^r 4\pi r^2 \rho(r) dr$, or removing dimensional factors, `mass[X]` $= -4\pi \xi^2 \frac{d\theta}{d\xi} = -4\pi$`Y[X][x]`$^2$`Y[X][z]`.

- `double *base`: an array used in calculation of stellar variables. Defined as `base[X]` $=$ `Y[X][y]`$^{n-1}$. Allows for fewer calls to `pow()` in calculation.

### 9.2.3 Constructors

There are three constructors:

- `Polytrope(double M, double R, double n, int len)`

- `Polytrope(double n, int len)`

- `Polytrope(double n, int len, double dx)`

In all, `n` is the polytropic index and `len` is the number of grid points to use. In the third, `dx` is the stepsize to use; this latter method is intended for testing errors when the grid size is changed.

`Polytrope(double M, double R, doublen ,int len)` begins by doing a check on $n$ to be sure it corresponds to a finite star, so that $0 \le n < 5$. If not, the constructor sends an error message and quits – future work should provide better error handling here. This constructor is intended for real stars, and will not construct $n = 5$ models.

The `name[]` for polytropes will be prefixed by `polytrope`, and appended with the index $n$ rounded to the first decimal place. So for $n = 3$, the `name[]` will be `polytrope.3.0`.

The first step of the constructor is to find the appropriate step size `dx` so the stellar surface occurs within exactly `len` steps. This is done using bisection with the variable $\theta_1 =$ `Y[len-1][y]`, beginning with an initial guess of `dx` $= \frac{\sqrt{6}}{len-1}$ (which is the correct value for an $n = 0$ polytrope). The bisection search will stop either when a `dx` is found so that `Y[len-1][y]`=0, or when the brackets have stopped moving.

Once the surface is found, we pick `indexFit` to be the place where $\theta(\xi) = 0.5$; because the stellar grid is twice the mode grid, this index must then be divided in half. Shooting methods in the `Mode` class will

match solutions at this index value. At the same time, we also calculate `base[X]` $= \theta^{n-1} = $ `Y[X][y]`$^{n-1}$ and `mass[X]`. For `mass[]`, we can use (9.1) to simplify the integration, noting

$$m(r) = \int_0^r 4\pi r^2 \rho(r) dr = \int_0^x 4\pi R_n^3 \xi^2 \rho_c \theta^n d\xi = 4\pi \rho_c R_n^3 \int_0^\xi \frac{d}{d\xi} \left( -\xi^2 \frac{d\theta}{d\xi} \right) = -4\pi R_n^3 \rho_c \xi^2 \frac{d\theta}{d\xi}; \qquad (9.4)$$

thus we can calculate `mass` directly without needing to use numerical integration techniques.

The scale of the star (including central values of $P(0), \rho(0)$ and radius) are fixed so that the total mass and radius correspond to `M,R`.

`Polytrope(double n,int len)` works the same, except that instead of using homology to fix mass and radius, everything is left dimensionless, where $G = P_c = \rho_c = 1$.

This is capable of handling the $n = 5$ case. In that case there is no surface, so the bisection method would never finish. We use the dreaded `goto` to skip over the bisection process, and use `dx` from our first guess. This index cannot model a realistic star, so is only useful when testing against the $n = 5$ analytic solution, $\theta_5(\xi) = \left( 1 + \frac{1}{3}\xi^2 \right)^{-1/2}$.

`Polytrope(double n, int len,double dx)` does not have a bisection search for `dx`, which is specified as a parameter. The fitting index is also chosen to be an exact power of 2, for easier comparisons. This constructor should only be used when testing errors or how quantities scale with changes in the grid size.

### 9.2.4 Calculation

- `void centerInit(double y[numvar])`: set initial values $\xi = $ `Y[0][x]` $= 0, \theta = $ `Y[0][y]` $= 1, \frac{d\theta}{d\xi} = $ `Y[0][z]` $= 0$.

- `void RK4step(double dx, double yin[numvar[, double yout[numvar])`: performs a single RK4 step of the Lane-Emden equations, with step size $\Delta\xi = $ `dx` from `yin[]` and ending at `yout[]`. Because the equations call for $\theta^n$, the program can produce `NaN`s at the surface if $\theta < 0$ and $n$ is not an integer. To prevent `NaN` errors, we use a complex variable `YCN` $= \theta^n$, and take only the real part of `YCN` in equations.

- `double RK4integrate(int Len, double dx)`: integrate from the center out to position `Len` using `RK4step()` above. Returns the value $\theta$`[Len-1]` at the end of the integration, to be used in bisection search of `dx`. Since only $\theta \geq 0$ is physical, the integration stops when $\theta$`[X+1]` $< 0$ and returns $\theta[X + 1]$).

- `int RK4integrate(int Len, double dx, int g)`: as above, but all the way to the end of `Len`; it does not stop if $\theta < 0$. The return value is an integer, `Len`. The integer `g` now mainly serves the role of distinguishing this method from the other method. Better naming would also work.

### 9.2.5 Boundary conditions for modes

The boundary conditions of the modes require series expressions for $A^*, U, c_1, V_g$. Because of the very simple mathematical structure of a polytrope it is possible to derive analytic formulas for these coefficients.

- `double ac[4]`: coefficients of power series for $\theta$ near the center, in terms of dimensionless $x = r/R_\star = \xi/\xi_1$:
  $$\theta(x) \approx \theta_0 + \theta_2 x^2 + \theta_4 x^4 + \theta_6 (r/R)^6 = \text{ac[0]} + \text{ac[1]}x^2 + \text{ac[2]}x^4 + \text{ac[3]}x^6.$$

- `double as[6]`: coefficients of power series for $\theta$ near the surface ( with $t = 1 - r/R_\star = 1 - \xi/\xi_1$)
  $$\theta(t) \approx \theta_0 + \theta_1 t + \theta_2 t^2 + \cdots = \text{as[0]} + \text{as[1]}t + \text{as[2]}t^2 + \cdots.$$

- `setupCenter()`: initialize the coefficients above. See `NewtonianBC.nb` for further information.

- `setupSurface()`: initialize the coefficients above. See `NewtonianBC.nb` for further information.

- `getXYZCenter(double* xyz, int& maxPow)`: where XYZ is `Astar`, `Vg`, `U`, `C1` Returns terms of the power series expansion in $x$ near the center. For a polytrope with index $n$, these are pretty easy to calculate, and are given up to fourth order. See the Mathematica notebook `NewtonianBC.nb` for detailed explanation. The coefficients are:

$$A^*(r) = 0 + A_V \left( \frac{n+1}{3\Gamma_1} \right) \xi_1^2 x^2 + A_V \left( \frac{2(n+1)}{36\Gamma_1} - \frac{2n(n+1)}{60\Gamma_1} \right) \xi_1^4 x^4 + \cdots \tag{9.5}$$

$$U(r) = 3 + \left( -\frac{n}{5} \right) \xi_1^2 x^2 + 54n \left( \frac{7n}{8100} - \frac{1}{1296} + \frac{5-8n}{15120} \right) \xi_1^4 x^4 + \cdots \tag{9.6}$$

$$c_1(r) = \left( -\frac{3z_1}{x_1} \right) + \left( \frac{3nz_1}{10x_1} \right) \xi_1^2 x^2 + \left( \frac{n(2n+25)z_1}{1400x_1} \right) \xi_1^4 x^4 + \dots \tag{9.7}$$

$$V_g(r) = 0 + \left( \frac{n+1}{3\Gamma_1} \right) \xi_1^2 * x^2 + \left( \frac{2(n+1)}{36\Gamma_1} - \frac{2n(n+1)}{60\Gamma_1} \right) \xi_1^4 x^4 + \cdots . \tag{9.8}$$

  Here $\xi_1 = $ `Y[len-1][x]` and $z_1 = \frac{d\theta}{d\xi}|_{\xi_1} = $ `Y[len-1][z]`. The coefficients are stored in the input `xyz[]` array to be returned. We have found terms up to order $x^4$; if more are requested with `maxPow`, then we set `maxPow` $= 4$ so that the `ModeDriver` only implements the BC to order $x^4$.

- `getXYZSurface(double* xyz, int& maxPow)`: as above, but near the surface expanded in powers of $t = 1 - x$. Unlike the central boundary, there is no requirement that only even powers appear. Two of the stellar variables, $A^*, V_g$ have a pole at $t = 0$, so require a $t^{-1}$ term. If $n$ is not an integer, there are also terms $t^n$ appearing in the expansions (I have included the $t^n$ term in the past, but it is not currently set up - need a method to include at least the first non-integer part of an expansion).

  Due to the nature of the recursion relation at the surface, to implement the BCs to 4th order, we need $A^*, V_g$ to order $t^3$ (which includes the $t^{-1}$ term), we need $c_1$ to $t^3$, and we need $U$ to order $t^4$. The values of the $U$ terms will depend nontrivially on the polytropic index $n$; for certain $n$, many terms will simply vanish. The details of the calculations are too tedious to write out here, but can be found in the Mathematica notebook `NewtonianBC.nb`.

In both the center and the surface, the BC is given to **fourth order**.

## 9.3 Internal Logic Behind the Interface

With the above variables, we can now define how the following `Star` quantities are computed:

- `double rad(int X)`: $r = R_n \xi$.

- `double rho(int X)`: $\rho = \rho_c \theta^n = $ `rho0*base[X]*Y[X][y]`.

- `double drhodr(int X)`: $\frac{d\rho}{dr} = \frac{n\rho_c}{R_n} * \theta^{n-1} \frac{d\theta}{d\xi} = $ `n*rho0*base[X]*Y[X][z]/Rn`.

- `double P(int X)`: $P = P_c \theta^{n+1} = $ `P0*base[X]*Y[X][y]*Y[X][y]`.

- `double dPdr(int X)`: $\frac{dP}{dr} = \frac{(n+1)}{R_n} \theta^n \frac{d\theta}{d\xi} = (n+1)*rho0*base[X]*Y[X][y]*Y[X][z]/Rn$.

- `double Phi(int X)`: $\Phi$ which has been matched so that $\Phi(R_\star) = -\frac{GM_\star}{R_\star}$. Returns $\Phi = -\frac{(n+1)P_c}{\rho_c}\theta + \Phi_c = $ `(n+1)*P0/rho0*(Y[len-1][x]*Y[len-1][z]-Y[X][y])`.

- `double dPhidr(int X)`: $g = \frac{d\Phi}{dr} = -\frac{(n+1)P_c}{\rho_c} \frac{d\theta}{d\xi} = $ `-(n+1)*P0/rho0*Y[X][z]/Rn`.

- `double mr(int X)`: $m(r) = -4\pi R_n^3 \rho_c \xi^2 \frac{d\theta}{d\xi} = $ `Rn^3*rho0*mass[X]`.

- `double Schwarzschild_A(int X, double GamPert)`: the Schwarzschild discriminant

$$A = \frac{d\theta/d\xi}{\theta R_n} \left( n - \frac{n+1}{\Gamma_1} \right) = \frac{Y[X][z]}{Y[X][y]R_n} \left( n - \frac{n+1}{GamPert} \right) .$$

  If `GamPert` $= 0$ is given, uses $\Gamma_1 = \Gamma$ (which will result in $A = 0$).

- double getAstar(int X, double GamPert:): dimensionless Schwarzschild discriminant

$$A^* = \frac{\xi}{\theta}\frac{d\theta}{d\xi}\left(\frac{n+1}{\Gamma_1} - n\right) = \frac{\texttt{Y[X][z]*Y[X][x]}}{\texttt{Y[X][y]}}\left(\frac{n+1}{\texttt{GamPert}} - n\right).$$

If `GamPert` $= 0$ is given, uses $\Gamma_1 = \Gamma$ (which will result in $A^* = 0$).

- double getU(int X): $U = \frac{4\pi\rho r}{d\Phi/dr} = -\frac{\xi \cdot \theta^n}{d\theta/d\xi} = -\frac{\texttt{Y[X][x]*Y[X][y]*base[X]}}{\texttt{Y[X][z]}}$. Special care must be take at X $= 0$, where $U = 3$.

- double getVg(int X, double GamPert): returns $V_g(r) = \frac{rg}{v_s^2} = -\frac{n+1}{\Gamma_1} * \frac{\xi}{\theta}\frac{d\theta}{d\xi} = -\frac{n+1}{\texttt{GamPert}}\frac{\texttt{Y[X][x]}}{\texttt{Y[X][y]}}\frac{\texttt{Y[X][z]}}{}$. If `GamPert` $= 0$, uses $\Gamma_1 = \Gamma$

- double getC(int X): returns

$$c_1(r) = \frac{M_\star}{m(r)}\frac{r^3}{R_\star^3} = \frac{z_1}{\xi_1} * \frac{\xi}{d\theta/d\xi} = \frac{\texttt{Y[len-1][z]*Y[X][x]}}{\texttt{Y[len-1][x]*Y[X][z]}}.$$

Special care must be take at X $= 0$, where $c_1 = -\frac{3z_1}{\xi_1}$.

- double sound_speed2(int X, double GamPert): returns $v_s^2 = \frac{\Gamma_1 P}{\rho} = \frac{\Gamma_1 P_c}{\rho_c} * \theta(x) = \frac{\Gamma_1 P_c}{\rho_c}\texttt{Y[X][y]}$, where $\Gamma_1 = $ `GamPert`. If no $\Gamma_1$ is specified, uses $\Gamma_1 = \Gamma$

- double Gamma1(int X): returns $\Gamma = 1 + \frac{1}{n}$.

These values are all calculated at $r = $ `rad(X)`. Since `base[X]` $= \theta^{n-1}$, this is used in the code to avoid calls to `pow()`. The other methods, `length()`, `Radius()`, etc., are trivially defined.

## 9.4   References

- Lane, J. H. (1870) for the first publication of Eqn. (9.1).

- Hansen, Kewaler, and Trimble (2004), in particular chapter 7 for a discussion of using RK4 method to calculate polytropic stellar models.

- Christensen-Dalsgaard and Mullan (1994) in their appendix, for a description of a similar recursive boundary process as that described in **??** (though there for a different form of the equations).

- Cox (1980), *Theory of Stellar Pulsation*, Section 17.6, for a brief of the recursion process of boundary conditions of the wave equations.

# 10    The ChandrasekharWD Class

This class represents a zero-temperature white dwarf star, held up entirely by electron degeneracy pressure, as first described by Chandrasekhar (1935).

Treated as a homogeneous gas, the exclusion principle indicates that each cell of phase space (quantal volume $d^3x d^3p = h^3$) can accommodate at most two electrons (two to account for the two spin states). In the limit of cold matter where $T \to 0$, all electrons will settle to the lowest possible unfilled state and the number density of electrons becomes

$$n_e = 2\frac{\int d^3p}{h^3} = \int_0^{p_F} \frac{8\pi\, p^2}{h^3} dp = \frac{8\pi\, p_F^3}{3h^3}, \tag{10.1}$$

where here $p_F$ is defined from the energy

$$E_F = \sqrt{p_F^2 c^2 + m_e^2 c^4}. \tag{10.2}$$

Define the parameters $x, \theta$ by

$$x = \frac{p_F}{m_e c}, \qquad p = m_e c \sinh\theta. \tag{10.3}$$

In terms of this the electron number density becomes

$$n_e = \frac{8\pi}{3h^3} m_e^3 c^3 x^3 = \frac{8\pi}{3}\left(\frac{m_e c}{h}\right)^3 x^3. \tag{10.4}$$

We can use kinetic theory to calculate the electron pressure

$$P_e = \frac{1}{3}\int_0^{p_F} \frac{p^2 c^2}{E} \frac{8\pi\, p^2}{h^3} dp = \frac{8\pi m_e^4 c^5}{3h^3}\int_0^{\theta_F} \sinh^4\theta\, d\theta. \tag{10.5}$$

Most of the density is going to come from the baryons. Define a parameter $\mu_e$, the mean atomic mass per electron. This number is determined by the chemical composition: for pure hydrogen, $\mu_e = 1$, as there is a single nucleon for each electron; for helium-4, $\mu_e = 2$; likewise for carbon-12, $\mu_e = 2$. In general,

$$\frac{1}{\mu_e} = \sum_i X_i \frac{Z_i}{A_i}, \tag{10.6}$$

where the sum is over all elements inside the star and $X_i$ is the relative mass abundance. In terms of $\mu_e$, we can express the baryon density as

$$\rho_B = \mu_e m_H n_e = \mu_e m_H \frac{8\pi}{3}\left(\frac{m_e c}{h}\right)^3 x^3 = Bx^3, \tag{10.7}$$

where $m_H$ is the mass of a hydrogen atom.

Define

$$f(x) = 8\int_0^{\theta_F} \sinh^4\theta\, d\theta = \left[(2x^3 - 3x)\sqrt{1 + x^2} + \sinh^{-1} x\right]. \tag{10.8}$$

The electron pressure becomes

$$P_e = \frac{\pi m_e^4 c^5}{3h^3} \left[ (2x^3 - 3x)\sqrt{1+x^2} + \sinh^{-1} x \right] = Af(x). \tag{10.9}$$

Note that

$$\frac{df(x)}{dx} = 8\frac{d\theta}{dx}\frac{d}{d\theta} \int_0^{\theta_F} \sinh^4 \theta \, d\theta = \frac{8\sinh^4 \theta_F}{\cosh \theta_F} = \frac{8x^4}{\sqrt{1+x^2}}. \tag{10.10}$$

Consider the equations of hydrostatic equilibrium:

$$\frac{d\Phi}{dr} = -\frac{1}{\rho}\frac{dP}{dr}, \qquad \frac{1}{r^2}\frac{d}{dr}\left( r^2 \frac{d\Phi}{dr} \right) = 4\pi G\rho. \tag{10.11}$$

Here, the only contribution to pressure we are considering is the partial pressure from electrons $P_e$, and the only density we are considering is the baryon mass density $\rho_B$. Therefore, combining these,

$$\frac{1}{r^2}\frac{d}{dr}\left( \frac{Ar^2}{Bx^3}\frac{df(x)}{dr} \right) = \frac{1}{r^2}\frac{d}{dr}\left( \frac{Ar^2}{Bx^3}\frac{8x^4}{\sqrt{1+x^2}}\frac{dx}{dr} \right) = -4\pi GBx^3. \tag{10.12}$$

Define a new radius $\xi$ by $r = R_n \xi$, analogous to in the Lane-Emden equation, with

$$R_n = \sqrt{\frac{2A}{\pi GB^2}}. \tag{10.13}$$

Further define the new variable $y$ by

$$y^2 = 1 + x^2. \tag{10.14}$$

Note

$$\frac{dy}{d\xi} = \frac{d}{d\xi}\sqrt{1+x^2} = \frac{x}{\sqrt{1+x^2}}\frac{dx}{d\xi}. \tag{10.15}$$

Then at last we have the equation

$$\frac{1}{\xi^2}\frac{d}{d\xi}\left( \xi^2 \frac{dy}{d\xi} \right) = -(y^2 - 1)^{3/2}. \tag{10.16}$$

This is the equation for Chandrasekhar's white dwarf. It is very similar to the Lane-Emden equation (9.1), and can be easily integrated numerically using the same techniques. At the center of the star the central value $y_o = y(\xi = 0)$ provides a free parameter; the surface will occur when $P_e = 0$, which happens when $f(x) = 0$, or when $x = 0$, which is to say when $y_1 = y(\xi_1) = 1$. Since $y$ will decrease, this means $y_o$ must be larger than 1.

## 10.1 The ChandrasekharWD Interface

ChandrasekharWD is a child of Star and must offer the same interface. In addition, ChandrasekharWD offers the following methods accessible to external code:

- double getXi(int k): return position vriable $xi = $ xi[k].

- double getX(int k): return Fermi-momentum variable $x = $ x[k].

- double getY(int k): return solution $y(\xi[k]) = $ y[k].

- double getYderiv(int k): return derivative $\frac{dy}{d\xi}|_{\xi[k]} = $ z[k].

These are used for testing against other solutions, such as tabulated values or relativistic solutions.

### 10.1.1 Constructors

There are two constructors available for a `ChandrasekharWD`:

- `ChandrasekharWD(double y0, double mue, int len)`: creates a WD of specific grid size (length) `len` with an initial value $y(0) = $ `y0`, and with mean baryons-per-electron ($\mu_e$) given by `mue`. To reproduce Chandrasekhar's famous result of the limiting mass, use `mue = 2`. The step size `dx` needed to reach the surface ($y = 0$) in exactly `len` steps is found by bisection.

- `ChandrasekharWD(double y0, int len, double dx)`: as above, but specifies the step size to use as `dx`. There is no guarantee of reaching the "true" surface. This is used for testing scaling relations. Do not use for production runs.

## 10.2 The `ChandrasekharWD` Implementation

The `ChandrasekharWD` definition specifies the following class data and methods used in the implementation. Because the mathematical structure is very similar to a polytrope, the implementation is also very similar to the `Polytrope`.

### 10.2.1 Constants

- `int len`: the maximum index of the polytrope, which is returned by `length()` function.

- `double Y0, Y02`: the central value $y(0)$ and its square.

- `double X0, X02`: the central value $x(0) = \sqrt{y(0)^2 - 1}$ and its square.

- `double A0`: the central pressure. This is set from physical variables described in Chandrasekhar 1939; to get better SSR, set to 1.

- `double B0`: the central density. This is set from physical variables described in Chandrasekhar 1939; to get better SSR, set to 1.

- `double Rn`: the scale factor $R_n = \sqrt{\frac{2A}{\pi G B^2}}$ with $r = R_n \xi$, where $\xi$ is the independent variable of equation (10.16).

### 10.2.2 Solution Variables

- `double *xi`: an array with the position variable $\xi$ from (10.16).

- `double *x`: an array with the relativistic parameter $x = p_F/mc = \sqrt{y^2 - 1}$.

- `double *y`: an array with the solution $y$ from (10.16).

- `double *z`: an array containing $z = \frac{dy}{d\xi}$, which is used in the integration method, and also allows the calculation of e.g. $\frac{d\rho}{dr}$ without the need for finite differencing.

- `double *f`: an array containing $f(x) = (2x^3 - 3x)\sqrt{1 + x^2} + 3\sinh^{-1}(x)$.

- `double *mass`: an array containing the interior mass $m(r)$ as a function of radius, $m(r) = \int_0^r 4\pi r^2 \rho(r)dr$, or removing dimensional factors, `mass[X]` $= -4\pi$`xi[X]`$^2$`*z[X]`.

### 10.2.3 Constructors

There are two constructors:

- ChandrasekharWD(double Y0, double mu_electron, int len)

- ChandrasekharWD(double Y0, int len, double dx)

In both, Y0 is the initial value $y_o$ = Y0 and len is the number of grid points to use. The parameter mu_electron = $\mu_e$, and dx is the stepsize to use. Tee second method is intended for testing errors when the grid size is changed.

Excepting the equations appearing in the integration, the internal logic of the constructors is identical to Polytrope. See Sec. **??** for more info.

### 10.2.4 Calculation

- double RK4integrate(int Len, double dx): integrate equation (10.16) from the center out to position Len using RK4. Returns the value y[Len-1] at the end of the integration, to be used in disection search of dx. Because the equations call for $(y^2 - 1)^{3/2}$, the program can produce NaNs if $y < 1$. To prevent NaN errors, we use a complex variable YCN = $(y^2 - 1)^{3/2}$, and consider the real part of YCN. Since only $y \geq 1$ is physical, the integration stops when y[X+1] < 1 and returns y[X+1] instead.

- int RK4integrate(int Len, double dx, int g): This method integrates as described above, all the way to the end of Len; it does not stop if $y < 1$. The return value is an integer, Len. The integer g now mainly serves the role of distinguishing this method from the other method. Better naming would also work.

### 10.2.5 Boundary conditions for modes

In order to implement the boundary conditions of modes, it is necessary to have power series expressions for $A^*, U, c_1, V_g$. For the case of a Chandrasekhar WD, because of the very simple mathematical structure it is possible to derive analytic formulas for these coefficients.

- double yc[4]: coefficients of power series for $y$ near the center, in terms of dimensionless $s = r/R_\star = \xi/\xi_1$:
$$y(\xi) \approx y_0 + y_2 s^2 + y_4 s^4 + y_6 s^6 = \text{yc[0]} + \text{yc[1]}s^2 + \text{yc[2]}s^4 + \text{yc[3]}s^6.$$

- double xc[2]: coefficients of
$$x(\xi) \approx x_0 + x_2 s^2 = \text{xc[0]} + \text{xc[1]}s^2.$$

- double fc[2]: coefficients of
$$f(x(\xi)) \approx f_0 + f_2 s^2 = \text{fc[0]} + \text{fc[1]}s^2.$$

- setupCenter(): initialize the coefficients above. See NewtonianBC.nb for further information.

- setupSurface(): an empty method. Only for conformity.

- getXYZCenter(double* xyz, int& maxPow): where XYZ is Astar, Vg, U, C1. Returns terms of the power series expansion in $s = r/R_\star$ near the center. The coefficients are stored in the xyz[] array to be returned. We have found terms up to order $s^4$; if more are requested with maxPow, then we set maxPow = 4 so that the ModeDriver only implements the BC to order $s^4$. The details of the calculations can be found in the Mathematica notebook NewtonianBC.nb.

- `getXYZSurface(double* xyz, int& maxPow)`: as above, but near the surface expanded in powers of $t = 1 - r/R_\star = 1 - s$. Unlike the central boundary, there is no requirement that only even powers appear. The stellar variable $A^*$ has a pole at $t = 0$, so requires a $t^{-1}$ term.

  Due to the nature of the recursion relation at the surface, to implement the BCs to 4th order, we need $A^*, V_g, c_1$ to $t^3$ (which includes the $t^{-1}$ term of $A^*$). All $U$ terms will simply vanish. The details of the calculations can be found in the Mathematica notebook `NewtonianBC.nb`.

In both the center and the surface, the BC is given to **fourth order**.

## 10.3   Internal Logic Behind the Interface

With the above variables, we can now define how the following `Star` quantities are computed:

- `double rad(int X)`: $r = R_n \xi$.

- `double rho(int X)`: $\rho = Bx^3$.

- `double drhodr(int X)`: $\frac{d\rho}{dr} = \frac{3B}{R_n} x^2 \frac{dx}{d\xi} = \frac{3B}{R_n} x^2 \frac{y}{x} \frac{dy}{d\xi} = \frac{3B}{R_n} xy \frac{dy}{d\xi} = $ `3B0*x[X]*y[X]*z[X]/Rn`.

- `double P(int X)`: $P = A * f(x) = $ `A0*f[X]`.

- `double dPdr(int X)`: $\frac{dP}{dr} = \frac{8A}{R_n} * x^3 \frac{dy}{d\xi}$, see eqn. 9 in Chandrasekhar 1935.

- `double Phi(int X)`: $\Phi$ which has been matched so that $\Phi(R) = -\frac{GM}{R}$.

- `double dPhidr(int X)`: $g = \frac{d\Phi}{dr} = -\frac{1}{\rho} \frac{dP}{dr} = -\frac{8A}{BR_n} * \frac{dy}{d\xi}$.

- `double mr(int X)`: $m(r) = -4\pi R_n^3 B * x^2 \frac{dy}{d\xi} = $ `B0*Rn`$^3$`*mass[X]`.

- `double Schwarzschild_A(int X, double GamPert)`:

$$A = -\frac{dy/d\xi}{R_n} \left[ \frac{8x^3}{\Gamma_1 f(x)} - \frac{3y}{x^2} \right]$$

  If `GamPert`$=0$, uses $\Gamma_1$ from the equation of state (which will result in $A^* = 0$).

- `double getAstar(int X, double GamPert)`:

$$A^* = \xi \frac{dy}{d\xi} \left( \frac{8x^3}{f(x)\Gamma_1} - \frac{3y}{x^2} \right).$$

- `double getU(int X)`: $U = \frac{4\pi\rho r}{d\Phi/dr} = -\frac{\xi x^3}{dy/d\xi}$. Special care must be taken at `X` $= 0$.

- `double getVg(int X, double GamPert)`: returns $V_g(r) = \frac{rg}{v_s^2} = -\frac{8\xi x^3}{\Gamma_1 f(x)} \frac{dy}{d\xi}$. where $\Gamma_1 = $ `GamPert`. If no $\Gamma_1$ is specified, uses $\Gamma_1 = $ `Gamma1(X)`, leading to $V_g(r) = -\frac{3\xi y}{x^2} \frac{dy}{d\xi}$.

- `double getC(int X)`: returns $c_1(r) = \frac{M}{m(r)} \frac{r^3}{R^3} = \frac{z_1}{\xi_1} * \frac{\xi}{dy/d\xi}$. Special care must be take at `X` $= 0$.

- `double sound_speed2(int X, double GamPert)`: $v_s^2 = \frac{\Gamma_1 P}{\rho} = \frac{\Gamma_1 A}{B} * \frac{f(x)}{x^3}$, where $\Gamma_1 = $ `GamPert`. If no $\Gamma_1$ is specified, then $v_s^2 = \frac{8A}{3B} * \frac{x^2}{y}$.

- `double Gamma1(int X)`: $\Gamma = \frac{8}{3} * \frac{x^5}{yf(x)}$.

These values are all calculated at $r = $ `rad(X)`. The other methods, `length()`, `Radius()`, etc., are trivially defined.

## 10.4   References

- Chandrasekhar (1935), MNRAS **95** for the first publication of (10.16).

- Chandrasekhar (1939), *An Introduction to the Study of Stellar Structure*, a classic text available from Dover, which includes any tabulated values as a means of testing.

# 11        The MESA Class

This class is used to read in data from MESA output to be used with our mode-calculation routines. We will assume we have calculated a stellar profile with MESA which has been saved with name mymesa.dat. The code is assuming output given by MESA v 1.01 in the .dat file – output columns explained here.

THRAIN is distributed with two sample profiles calculated with MESA, one for a cold and another for a hot WD model. Both of these were calculated by simply modifying the wd_cool_0.6M example in the standard MESA test suite that comes with the MESA distribution. The sample input file sampleinput5.txt will calculate several modes for a MESA model.

## 11.1    The MESA Interface

MESA is a child of Star, which means it must implement the Star interface. This is a minimal stellar model, as all work has been done for us by MESA.

### 11.1.1    Constructors

There is only one constructor:

- MESA(const char* file, int L): opens and reads in the MESA output stored in the file, whose address is given by file; the final model is to be interpolated to have a specific grid size (length) *close to* L.

## 11.2    The MESA Implementation

The MESA definition specifies the following class data and methods used in the implementation.

Many of the variables below are objects called Splinors, which are spline interpolators. These all use the radial coordinate as the independent variable ($x$), and different columns of the MESA file (or calculated quantities) as the dependent variable ($y$).

These Splinors can be called like a function, taking the argument $x = r/R_\star$ as in radi[]. Splinors also have a derivative method (deriv()), which returns the derivative of the interpolating spline at the given $x$.

### 11.2.1    Constants

- int len: The desired gridsize of the model to be used in pulsations.

- int subgrid: The number of subgrid points needed to achieve the closest proximity to len. Due to the nature of integer division, and the need for half-steps, the actual gridsize will not be exactly as desired.

- int Ntot: The original number of grid points in the first line of MESA file.

- int Mtot: the mass of the star $M_\star$ in CGS units, read from first line of MESA file.

- int Rtot: the radius of the star $R_\star$ in CGS units, read from first line of MESA file.

- `double Dscale`: density scale $\frac{M_\star}{4\pi R_\star^3}$.

- `double Pscale`: pressure scale $\frac{GM_\star^2}{4\pi R_\star^4}$.

- `double Gscale`: gravitational scale $GM_\star/R_\star^2$.

## 11.2.2 Solution Variables

- `double *radi`: The increased radial grid, $x = r/R_\star$, of length `len`. Takes initial data from MESA file, column 2; augmented with subgrids between points.

- `Splinor *dens`: density from MESA column 7, quantity $\hat{\rho} = \rho/$`Dscale`.

- `Splinor *pres`: pressure from MESA column 5, quantity $\hat{P} = P/$`Pscale`.

- `Splinor *mass`: mass from MESA column 3, quantity $\hat{m} = m/$`Mtot`.

- `Splinor *grav`: gravitational field strength calculated as $\hat{g} = \hat{m}/x^2$.

- `Splinor *BVfq`: Brunt-Väiäsäla frequency from MESA column 9, quantity $\hat{N}^2 = N^2 * R_\star^2/(GM_\star)$.

- `Splinor *Gam1`: adiabatic index $\Gamma_1$ from MESA column 10.

## 11.2.3 Stellar Properties for Waves

- `Splinor *aSpline`: stellar variable $A^* = -\hat{N}^2 x/\hat{g}$.

- `Splinor *vSpline`: stellar variable $V = \hat{g}x\hat{\rho}/\hat{p}$ (note: this is *not* $V_g$).

- `Splinor *cSpline`: stellar variable $c_1 = x^3/\hat{m}$.

- `Splinor *uSpline`: stellar variable $U = \hat{\rho}x^3/\hat{m}$.

## 11.2.4 Constructor

There is only one constructor:

- `MESA(char *file, int len)`

It begins by opening the file specified by `file`, and reads in `Ntot`, which MESA prints in the 1st line. The data contains $r, m(r), \rho, P, N^2, \Gamma_1$ in CGS units, and we calculate $g = Gm/r^2$, along with $A, U, c_1, V$.

While CGS units are great for scales of a tablespoon of water at STP, they are not so great when talking about literally astronomical pressures, temperatures and volumes. For this reason, all quantities are converted into units where $M_\star = R_\star = G = 1$, which perform better numerically. We define the Schwarzschild dimensionless units, represented by a hat, as (see Schwarzschild 1958)

$$r = R_\star x, \qquad m = M_\star \hat{m}, \qquad P = \frac{GM_\star^2}{4\pi R_\star^4}\hat{P}, \qquad \rho = \frac{M_\star}{4\pi R_\star^3}\hat{\rho}, \qquad g = \frac{GM_\star}{R_\star^2}\hat{g}. \tag{11.1}$$

We now augment the grid so that there are *at least* `len` points. The simplest way to do this is to add evenly spaced subgrid points between each pair of data points. We choose to divide each subgrid into a binary fraction, to ensure there are half-steps for use in the mode solution. This means you can not exactly specify how many grid points you can use, and you must use twice as many as are available in the MESA output.

Once the grid is formed, we initialize all `Splinor` quantities.

We next set `indexFit`, the fitting index for mode calculations, as the location where $\rho = 0.5$`dens[0]`.

### 11.2.5 Calculation

The only calculation to be performed is the spline-fitting, which is handled automatically by `Splinor`. Everything else has been handled by `MESA` to generate the `.dat` file.

This class exists only to allow mode calculations with data computed by `MESA`. The mode routines we have defined rely on simple RK4, which requires evaluation of the derivatives at half-grid points (i.e. for `x[1]`, requires an evaluated at `x[1]` + $\frac{1}{2}$(`x[2]` − `x[1]`)). It is highly unlikely that the grid chosen by `MESA` will give these values. Therefore, in this program we form our own grid, which must at minimum define the half-grid points. See Figure 11.1.
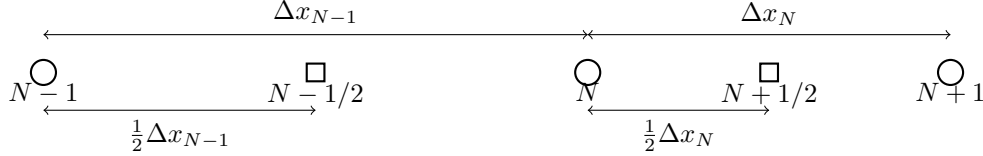


Figure 11.1: The circles labeled represent the grid as computed by `MESA`. These are drawn to be unevenly spaced, as they are in most `MESA` data files. The squares labeled are the half-grid points needed by the modes' RK4 algorithms. These must be determined by spline-fitting.

### 11.2.6 Boundary conditions for modes

In order to implement the boundary conditions of modes, it is necessary to have power series expressions for $A^*, U, c_1, V_g$. This is more difficult to do for the complicated physical processes taking place inside a white dwarf star.

- `setupCenter()`: In the center of the star, suppose we have a general equation of state $\hat{P} = F(\hat{\rho}, \hat{T})$ in the center, with $\hat{T}$ a dimensionless temperature. We can then find expansions for $\hat{\rho}, \hat{P}$ in terms of $x = r/R_\star$,

$$\hat{T} = \hat{T}_0 + \hat{T}_2 x^2 + \hat{T}_4 x^4 + \cdots \tag{11.2a}$$

$$\hat{\rho} = \hat{\rho}_0 + \hat{\rho}_2 x^2 + \hat{\rho}_4 x^4 + \cdots \tag{11.2b}$$

$$= \hat{\rho}(0) + \left(\frac{\hat{P}_2 - \hat{T}_2 F_T}{F_\rho}\right) x^2 + \left(\frac{2\hat{P}_4 - 2\hat{T}_4 F_T - \hat{T}_2^2 F_{TT} - 2\hat{T}_2 \hat{\rho}_2 F_{T\rho} - \hat{\rho}_2^2 F_{\rho\rho}}{2F_\rho}\right) x^4 \tag{11.2c}$$

$$\hat{P} = \hat{P}_0 + \hat{P}_2 x^2 + \hat{P}_4 x^4 + \cdots \tag{11.2d}$$

$$= \hat{P}(0) + \left(-\frac{1}{6}\hat{\rho}_0^2\right) x^2 + \left(-\frac{2}{15}\hat{\rho}_0\hat{\rho}_2\right) x^4 + \cdots, \tag{11.2e}$$

in terms of which

$$c_1 = \frac{3}{\hat{\rho}_0} - \frac{9}{5}\frac{\hat{\rho}_1}{\hat{\rho}_0^2} x^2 + \left(\frac{27}{25}\frac{\hat{\rho}_2^2}{\hat{\rho}_0^3} - \frac{9}{7}\frac{\hat{\rho}_4}{\hat{\rho}_0^2}\right) x^4 + \cdots \tag{11.3a}$$

$$U = 3 + \frac{6}{5}\frac{\hat{\rho}_2}{\hat{\rho}_0} x^2 + \left(\frac{12}{7}\frac{\hat{\rho}_4}{\hat{\rho}_0} - \frac{18}{25}\frac{\hat{\rho}_2^2}{\hat{\rho}_0^2}\right) x^4 + \cdots \tag{11.3b}$$

$$V_g = -\frac{2\hat{P}_2}{\hat{P}_0\Gamma_1} x^2 + \left(\frac{2\hat{P}_2^2}{\hat{P}_0^2\Gamma)1} - \frac{4\hat{P}_4}{\hat{P}_0\Gamma_1}\right) x^4 + \cdots \tag{11.3c}$$

$$A^* = \left(\frac{2\hat{P}_2}{\hat{P}_0\Gamma_1} - \frac{2\hat{\rho}_2}{\hat{\rho}_0\Gamma_1}\right) x^2 + \left(\frac{4\hat{P}_4}{\hat{P}_0^2\Gamma_1} - \frac{2\hat{P}_2^2}{\hat{P}_0^2\Gamma_1} - \frac{4\hat{\rho}_4}{\hat{\rho}_0} + \frac{2\hat{\rho}_2^2}{\hat{\rho}_0^2}\right) x^4 + \cdots \tag{11.3d}$$

These are general. For `MESA`, we approximate the equation of state in the center as a polytrope with $n = \frac{1}{\Gamma_1(0)-1}$. We can then find series coefficients for $\theta$ around $x$ (see Sec. 9.2.5)

$$\theta = \theta_0 + \theta_2 x^2 + \theta_4 x^4 + \cdots \tag{11.4}$$

in terms of which $\hat{\rho}_2 = n\hat{\rho}_0\theta_2$ and $\hat{\rho}_4 = \frac{1}{2}\hat{\rho}_0(2n\theta_4 + n(n-1)\theta_2^2)$. The rest then follows.

- `getXYZCenter(double* xyz, int& maxPow)`: where XYZ is `Astar`, `Vg`, `U`, `C1` Returns terms of the power series expansion as above. The coefficients are replaced into the `xyz[]` array to be returned. If more are requested with `maxPow`, then we set `maxPow` = 4 so that the `ModeDriver` only implements the BC to order $x^4$.

- `setupSurface()`: Follows a similar method as for the center above, except at the surface expanded in powers of $t = 1 - r/R_\star$. The EOS near the surface is assumed to be an ideal gas plus radiation pressure. See the `NewtonianBCs.nb` notebook for more details.

- `getXYZSurface(double* xyz, int& maxPow)`: as above, but near the surface expanded in powers of $t = 1 - r/R$.

Within the directory `output/calc_name/star/wave_coefficient`, there are several files for testing the goodness of these power series expansions. The coefficients of the expansion are printed to files `center.txt,surface.txt`. The first few points of $\rho, P, A^*, U, V_g, c_1$ are printed, from the stellar data and using the expansion function, for comparison, and the errors printed in the corresponding `.png` files for easier viewing.

## 11.3 Internal Logic Behind the Implementation

With the above variables, we can now define how the following `Star` quantities are computed:

- `double rad(int X)`: $r = \text{Rtot} * \text{radi[X]}$.

- `double rho(int X)`: $\rho = \text{Dscale} * \text{dens(X)}$.

- `double drhodr(int X)`: $\frac{d\rho}{dr} = \text{Dscale/Rtot} * \text{dens'(X)}$.

- `double P(int X)`: $P = \text{Pscale} * \text{pres(X)}$.

- `double dPdr(int X)`: $\frac{dP}{dr} = \text{Pscale/Rtot} * \text{pres'(X)}$.

- `double Phi(int X)`: <span style="color:red">Not yet implemented (01/22/2022) .</span>

- `double dPhidr(int X)`: $g = \frac{d\Phi}{dr} = \text{Gscale} * \text{grav(X)}$.

- `double mr(int X)`: $m(r) = \text{Mtot} * \text{mass(X)}$.

- `double Schwarzschild_A(int X, double GamPert)`: If `GamPert`=0, $A = \text{aSpline(X)/radi[X]/Rtot}$. Otherwise,
$$A = \frac{\text{dens'(X)}}{\text{dens(X)*Rtot}} - \frac{\text{pres'(X)}}{\text{Gam1(X)*pres(X)*Rtot}}.$$
(Note units! $A^*$ is dimensionless, but Schwarzschild discriminant is not.)

- `double getAstar(int X)`: If `GamPert`=0, $A^* = \text{aSpline(X)}$. Otherwise
$$A = \frac{\text{radi[X]pres'(X)}}{\text{Gam1(X)*pres(X)}} - \frac{\text{radi[X]dens'(X)}}{\text{dens(X)}}.$$

- `double getU(int X)`: $U = \text{uSpline(X)}$.

- `double getVg(int X, double GamPert)`: $V_g = \text{vSpline(X)/GamPert}$. If `GamPert`=0, $V_g = \text{vSpline(X)/Gam1(X)}$.

- double getU(int X): $U = $ uSpline(X).

- double getC(int X): $c_1(r) = $ cSpline(X).

- double sound_speed2(int X, double GamPert): $v_s^2 = \frac{\Gamma_1 P}{\rho}$. If GamPert=0, $v_s^2 = \frac{\text{Gam1(X)*pres(X)}}{\text{dens(X)}}$. Otherwise, $v_s^2 = \frac{\text{GamPert*pres(X)}}{\text{dens(X)}}$.

For brevity's sake, all calls to interpolation are written as func(X); in C++ syntax, they must actually be written func->interp(radi[X]). All calls to interpolated derivatives are written as func'(X), instead of func->deriv(radi[X]).

## 11.4   Limitations

The SSR() method used for modes assumes a uniform grid spacing when calculating derivatives; for this reason, the printed mode SSR in output from this model will be higher than expected, due to numerical errors introduced in the derivatives. For modes with $l \geq 2$, I suggest using the $c_0$ column (the last one) from the tidal_overlap.txt file as a better measurement of error.

For hot WD stars in the instability strip, for some reason many modes will fail to calculate, and errors are high. Cold stars below the instability strip seem to calculate well. When referring to eigenmodes calculated from MESA, check the graph of the mode. The central red vertical line shows the fitting point of the double-shooting method; ensure regular behavior around this line.

## 11.5   References

- Paxton et al. (2010), the original paper introducing MESA to the world, describing the physics and numerical methods used within the code.

- Paxton et al. (2013), describes extensions to MESA, including new capabilities for small planets.

- Paxton et al. (2015), further extends MESA, including modeling supernovae.

- Schwarzschild (1958), *Structure and Evolution of the Stars*, available from Dover, as a reference for the dimensionless variables chosen here.

- Fuller & Lai (2011), for an explanation of the $c_0$ column (and $Q_\alpha$ column) of tidal_overlap.txt. See especially their Table 1.

# Part IV

# Mode Drivers

# 12  The NonradialModeDriver Class

This class controls the nonradial, $k, \ell, m$ oscillation modes calculated for the full fourth-order Newtonian system, which consists of the LAWE and the perturbed Poisson equation:

$$\omega^2 \boldsymbol{\xi} = -\nabla \chi + \frac{\vec{A}}{v_s^2}(\nabla \cdot \boldsymbol{\xi}) \tag{12.1a}$$

$$\nabla^2 \Delta \Phi = 4\pi G \Delta \rho. \tag{12.1b}$$

With some algebra this can be put in the form of four first-order equations

$$x\frac{dy_1}{dx} = [V_g - 3 + (2 - \ell)]\, y_1 + \left[\frac{\ell(\ell+1)}{c_1 \bar{\omega}^2} - V_g\right] y_2 + V_g y_3 \tag{12.2a}$$

$$x\frac{dy_2}{dx} = \left[c_1 \bar{\omega}^2 - A^*\right] y_1 + [1 + A^* - U + (2 - l)]\, y_2 - A^* y_3 \tag{12.2b}$$

$$x\frac{dy_3}{dx} = [1 - U + (2 - \ell)]y_3 + y_4 \tag{12.2c}$$

$$x\frac{dy_4}{dx} = A^* U y_1 + U V_g y_2 + [\ell(\ell+1) - U V_g]\, y_3 + [(2 - \ell) - U]y_4 \tag{12.2d}$$

where

$$y_1 = \frac{1}{x^{\ell-2}}\frac{\xi^r}{r}, \quad y_2 = \frac{1}{x^{\ell-2}}\frac{\chi}{gr}, \quad y_3 = \frac{1}{x^{\ell-2}}\frac{\Delta\Phi}{gr}, \quad y_4 = \frac{1}{x^{\ell-2}}\frac{1}{g}\frac{d\Delta\Phi}{dr}, \tag{12.3}$$

with $x = r/R$ and where $U, V_g, A^*, c_1$ are unitless stellar variables defined as

$$U = \frac{d\log g}{d\log r} + 2, \quad V_g = -\frac{1}{\Gamma_1}\frac{d\log P}{d\log r}, \quad A^* = \frac{1}{\Gamma_1}\frac{d\log P}{d\log r} - \frac{d\log\rho}{d\log r}, \quad c_1 = \frac{M}{m(r)}\left(\frac{r}{R}\right)^3. \tag{12.4}$$

We could also write this set of equations in the form

$$x\frac{d}{dx}\vec{Y} = x\frac{d}{dx}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{13} & 0 \\ 0 & 0 & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \mathbf{A}\vec{Y}. \tag{12.5}$$

using matrix notation. Notice that the LHS of (12.2) is in terms of $x\frac{d}{dx}$, and not merely $\frac{d}{dx}$. This is to preserve unitlessness of the equations, and preserves stability near the center.

## 12.1  The NonradialModeDriver Interface

NonradialModeDriver is a child of ModeDriver, which means it must uphold the ModeDriver contract. It does not define anything beyond that.

### 12.1.1 Constructor

There is only one constructor:

- `NonradialModeDriver(Star *s, double Gamma1)`

This single constructor initializes all arrays and sets up the boundaries.

## 12.2 The `NonradialModeDriver` Implementation

The `NonradialModeDriver` definition specifies the following class data and methods used in the implementation:

### 12.2.1 Constants

- `int len`: the number of gridpoints of the `Mode<N>` calculation.

- `int len_star`: the number of gridpoints of the underlying `Star` model.

- `double adiabatic_index`: the adiabatic exponent $\Gamma_1$: if set to zero uses the background star's value of $\Gamma_1$.

- `int BC_C, BC_S = 4`: the order of the power series expansions we would like for the central, surface boundary conditions. The actual order will depend on the background stellar model.

### 12.2.2 Physics

We need the variables $A^*, U, c_1, V_g$ defined in (12.4):

- `double r[len_star]`: the dimensionless radius $r[X] = x = r/R$.

- `double A[len_star]`: the dimensionless Schwarzschild discriminant $A[X] = A^* = \frac{1}{\Gamma_1} \frac{d \log P}{d \log r} - \frac{d \log \rho}{d \log r}$.

- `double U[len_star]`: the variable $U[X] = U = \frac{d \log m}{d \log r}$.

- `double C[len_star]`: the variable $C[X] = c_1 = \frac{M}{R^3} \frac{r^3}{m}$.

- `double V[len_star]`: the variable $V[X] = V_g = -\frac{1}{\Gamma_1} \frac{d \log P}{d \log r}$.

These variables are called from the stellar model through `*star`.

### 12.2.3 Calculation

- `void initializeArrays()`: fill out values of $A[], U[], C[], V[]$ from `*star`.

- `void getCoeff(double *CCI, int X, int b, double omeg2, int L)`: return the matrix $\mathbf{A} = \text{CCI}$ shown in (13.3).

- `void getBoundaryMatrix(int nv, double* y0, double* ys, double** y, int* indexOrder)`: the boundary matrix y specifies how the `Mode<N>`'s shoot-to-center algorithm described in Sec. 5.2.5 specifies values at each boundary as in (5.6). For the fourth-order nonradial mode, we pick

$$\vec{Y}^{(c1)} = (1, y_2, 0, 0), \vec{Y}^{(c2)} = (0, 0, 1, y_4), \vec{Y}^{(s1)} = (0, y_2, 1, y_4), \vec{Y}^{(s2)} = (1, y_2, 0, 0), \quad (12.6)$$

which leads to the matrix y returned. The array `indexOrder = {y1,y3,y3,y1}` indicates the first solution comes by scaling $y_1$, the second by scaling $y_3$, the third by scaling $y_3$, and the final by $y_1$, so that these BCs can be rescaled to make the physical solution.
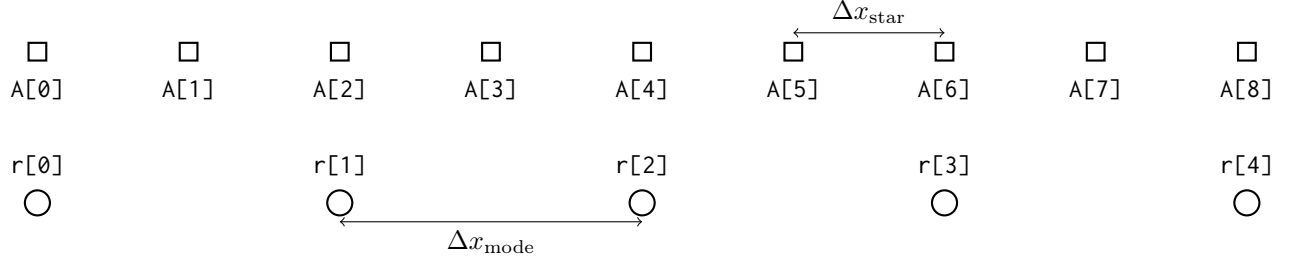
Figure 12.1: The squares are stellar grid points with spacing $\Delta x_{\mathrm{star}}$ containing values such as $A^* = $ A[N]. The circles are mode grid points with spacing $\Delta x_{\mathrm{mode}}$, and contain values such as $r = $ r[N] or mode values $y_1(\mathsf{r[N]}) = $ yy[y1][N].

- double SSR(double omeg2, int l, ModeBase* mode): calculate the RMSR in equations (12.1), where the variables in (12.1) double are calculated from the background star and the dimensionless variables in (12.2). This requires numerical differentiation to evaluate quantities such as $\frac{d^2}{dr^2}\Delta\Phi$. To make the derivatives as accurate as possible, this uses a 7-point stencil for the differences; however, **this requires an assumption of a uniform grid**. For modes calculated for background stars without a uniform grid, the RMSR will record additional numerical error, making these values larger. This is important for modes calculated with MESA data.

- double innerproduct(ModeBase* mode1, ModeBase* mode2): calculates an inner product between two nonradial modes, defined by $\langle \vec{\xi_1} | \vec{\xi_2} \rangle = \int d^3 r \rho \vec{\xi_1^*} \cdot \vec{\xi_2}$. Must first transform to dimensionless $y_i$ into physical perturbations $\xi^r, \chi, \Delta\Phi$, then uses trapezoidal integration. See Press & Teukolsky (1977).

- double tidal_overlap(ModeBase* mode): as above, but calculates an overlap coefficient with an external tidal force. See Press & Teukolsky (1977).

### 12.2.4 Constructor

The only constructor does not do much more than initialize the arrays A[],U[],C[],V[] described above and which appear in (12.2), and then set up the boundary conditions described below. The only point of it worth mentioning is the selection of the grid. As mentioned above, there are two grids, one of length len and the other of length len_star.

In the RK4 integration, it is necessary to have values corresponding to midpoints, e.g. x[n] + 0.5dx. To achieve this without the error inherent in spline fitting, the mode calculation uses half the number of grid points as the stellar background code (see Fig 12.1).

It is necessary to avoid fence-post issues at the boundaries. If the background star has an odd number of points, then we can evenly pick the mode grid so that there is always a stellar grid point on the half-grid,
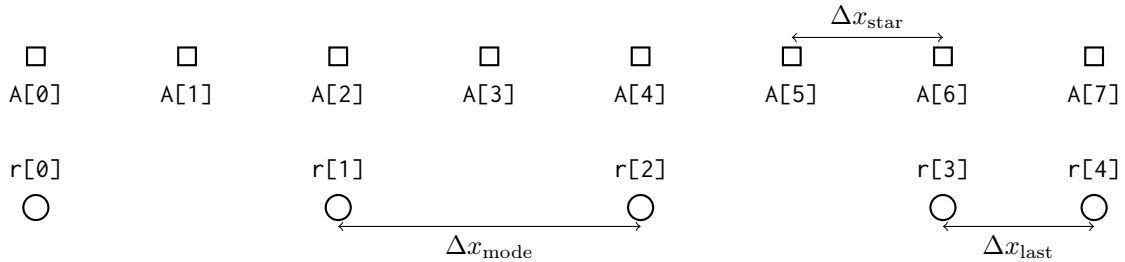


Figure 12.2: As in Fig. 12.1 the stellar grid is odd and can't be evenly divided. To fix this, the final grid spacing is $\Delta x_{\mathrm{last}} = \frac{1}{2}\Delta x_{\mathrm{mode}}$. Because of the power series solutions used there, this does not impact the accuracy of the calculation.

by picking the grid with total length `len` = (`len_star` + 1)/2. If the background star has an even number of points, the situation is a bit more complicated (see Fig. 12.2). The mode grid cannot be chosen so that there is always a stellar grid point on the half-grid; there has to be one point left over. We make the leftover point the last grid point, and the spacing there will be half the spacing elsewhere. To achieve this, choose `len` = `len_star`/2 + 1.

### 12.2.5 Miscellaneous

- `enum VarNames {y1=0, y2, y3, y4}`: the conventional names for the variables are $y_1, \ldots, y_4$; however, C++ is zero-indexed. To avoid calling, e.g. `yy[1]` (being $y_2$) when $y_1$ is intended, one can instead call `yy[y1]`.

- `void varnames(std::string *names)`: allows for `Mode<N>` to know what labels to assign the mode variables when it is printing graphs. The input array of `string`s will be set with the names described here.

### 12.2.6 Boundary conditions

- `Ac[3], Vc[3], cc[3], Uc[3]`: coefficients from power series expansions of $A^*, V_g, c_1, U$ to order $x^4 = (r/R)^4$ at the center of the star. Will come from the particular `Star` model.

- `As[5], Vs[5], cs[5], Us[5]`: coefficients from power series expansions of $A^*, V_g, c_1, U$ to order $t^4 = (1 - r/R)^4$ at the surface of the star. Will come from the particular `Star` model.

- `cProdc[3], cProds[5]`: coefficients in expansion of `cProd` = $\frac{1}{c_1}$ at the center and surface of the star.

- `setupBoundaries()`: initialize the above arrays at both boundaries.

- `CenterBC(double** ymode, double* y0, double omeg2, int l, int m)`: Initialize the first few steps of the eigenmode using a power series expansion. The array `y0[]` specifies the values of $y_i$ at the very center. Once the coefficients are found, the first `start` values are calculated in `ymode[][]` using the series expansion. Then `start` is returned, indicating where the usual integration method of `Mode<N>` begins.

- `SurfaceBC(double** ymode, double* y0, double omeg2, int l, int m)`: As above, but at the surface.

## 12.3 References

- Dziembowski, 1971, for the first publication of the equations in (12.2).

- Unno et al., *Nonradial Oscillations of Stars*, 1979, for detailed description of the equations (12.2) and their boundary conditions.

- Cox, *Theory of stellar pulsation*, 1980, for a detailed discussion of nonradial oscillations.

- Press & Teukolsky, 1977, for a better description of the inner product and a definition of the overlap coefficients.

# 13 The `CowlingModeDriver` Class

This class represents $n, l, m$ oscillation modes calculated in the Cowling approximation, where $\Delta\Phi \to 0$. In this approximation the LAWE (12.1) simplifies to the equations of motion for an oscillation mode

$$\frac{d\xi^r}{dr} = \left[\frac{g}{v_s^2} - \frac{2}{r}\right]\xi^r + \left[\frac{\ell(\ell+1)}{r^2\sigma^2} - \frac{1}{v_s^2}\right]\chi \tag{13.1a}$$

$$\frac{d\chi}{dr} = \left[\sigma^2 - N^2\right]\xi^r - A\chi \tag{13.1b}$$

which we rewrite in the form

$$x\frac{dy_1}{dx} = [V_g - 3 + (2-\ell)]y_1 + \left[\frac{\ell(\ell+1)}{c_1\bar{\omega}^2} - V_g\right]y_2 \tag{13.2a}$$

$$x\frac{dy_2}{dx} = [c_1\bar{\omega}^2]y_1 + [1 + A^* - U + (2-\ell)]y_3 \tag{13.2b}$$

or in matrix form

$$x\frac{d}{dx}\vec{Y} = x\frac{d}{dx}\left[\begin{array}{c} y_1 \\ y_2 \end{array}\right] = \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array}\right]\left[\begin{array}{c} y_1 \\ y_2 \end{array}\right] = \mathbf{A}\vec{Y}. \tag{13.3}$$

These variables have the same meaning as in (12.2).

This class works exactly as the `NonradialModeDriver` class, except that it uses `num_var`=2. The boundary conditions are the same, but cut off to two variables.

Because there are only two variables, there are only two BCs, one at each boundary. This means that the outward and inward solutions are the only ones in their regions, which we can label

$$\vec{Y}^{(c)} = (1, y_2), \vec{Y}^{(s)} = (1, y_3). \tag{13.4}$$

These two solutions can be used in a Wronskian, which will vanish when the correct $\bar{\omega}^2$ is found, and can be rescaled to match at the fitting point, as described in Sec. 5.2.5.

## 13.1 References

Same as for `NonradialModeDriver`.