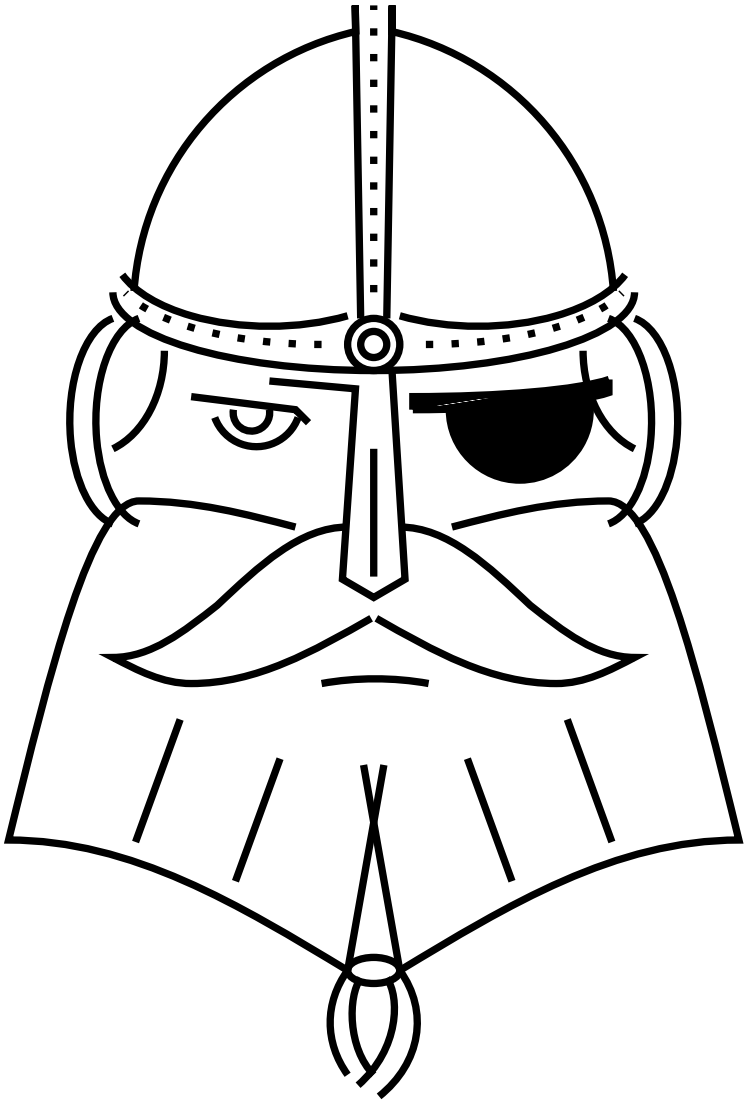
Thrain Documentation v1.0

S. Reece Boston*

January 21, 2022



THE MIGHTY WHITE DWARF CODE

*rboston628@gmail.com

Contents

Chapter 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 (2021)

In bibtex

```
 @author{Boston2022b,
author = "{{Boston}, S.~R.~ and {Clemens}, J.~C.~}",
title = "{The Relativistic Limit of White Dwarf Asteroseismology}",
year = {2021}
}
```

1.2 Version 1.0

This documentation was created for release with version 1.0, which was released with the above paper. Later updates will include functionality for realistic simple WD models, and for asteroseismology in full general relativity.

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 **??**.
- `Mode<N>`: Represents a single klm-mode, described by an `N`-dimensional eigenfunction (e.g. for a Cowling mode `N`=2). Described in **??**.
- `ModeDriver`: Represents a particular form of a stellar wave equation (e.g. Cowling modes, Dziembowski equations, or non-adiabatic waves). Described in **??**.

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 **??**.

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 `Star`s have an method `rho()` in their interface, which must return the density ρ; however, the implementation can vary on how ρ is found, whether by storing ρ in an array, or calculating it from other quantities.

This is illustrated in Figure **??**.

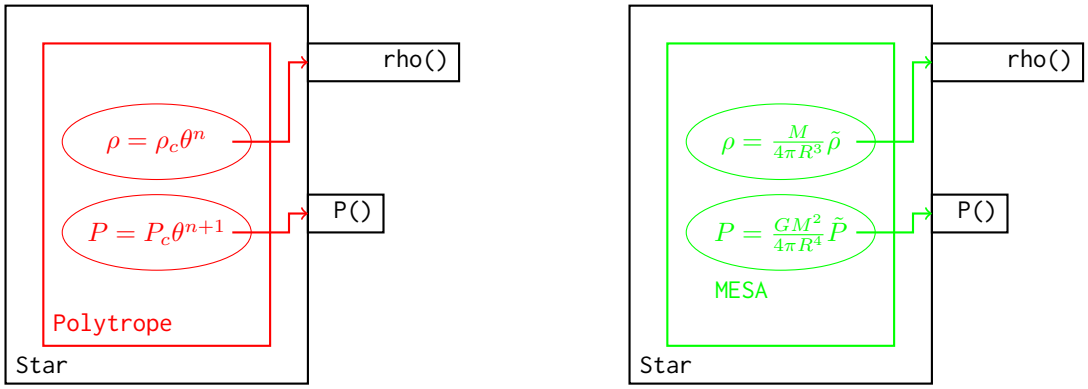


Figure 1.1: An illustration of a class (the boxes labeled `Star`). The interface (the "pipes" `rho()`, `P()`) can connect to external code needing a star's density and pressure. The internal logic of the implementation (the circles) is illustrated for two `Star` daughters: the `Polytrope` class (red)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

1.4 Diagram of Program Structure

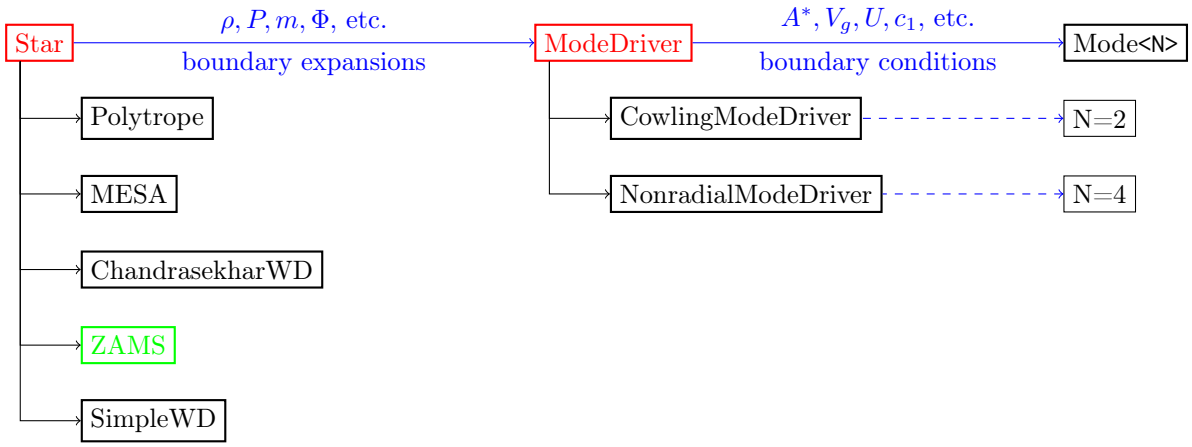


Figure 1.2: Structure of program, showing hierarchies. Red boxes are abstract classes, which provide rules for child classes to follow. Blue lines show friendship. Green planned but unimplemented. Any figure lower in the hierarchy (a child) can be used in place of a parent or grandparent class. Therefore, a `ChandrasekharWD` can be used anywhere the program calls for a `Star`.

Chapter 2

Using the command line program

2.1 Installation and Set Up

This program was developed in macOS and originally compiled with `gcc`. 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.

To compile, first `cd` to the `lib` directory and type

```
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 so but there are still problems, please email me.

2.2 Creating an input file

THRAINis 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:
  - a keyword for the *regime* of physics:
    - `newtonian` – Newtonian physics
    - `1pn` – the first post-Newtonian approximation
    - `gr` – Einstein's general relativity
  - a keyword for the stellar background model:
    - `polytrope` – a polytrope (works in all regimes)
    - `MESA` – a wrapper for a `MESA` model (only in `newtonian`)
    - `CHWD` – a Chandrasekhar WD (only in `newtonian`, `1pn`)
  - model-specific parameters. See Sec **??** below.
- Line 3: in **some models**, the word `Params:` followed by two of the following:
  - the word `mass` and a real number indicating the total mass of the star
  - the word `radius` and a real number indicating the total radius of the star
  - the word `logg` and a real number indicating the $\log_{10} g$ surface gravity of star

  - the word `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` (only in `newtonian` or `gr`)
  - `nonradial`

  The adiabatic index is a number. The special cases $\Gamma_1 = 5/3$ and $\Gamma_1 = 4/3$ can be specified by writing the fractions `5/3` and `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 ℓ, k, where k counts the nodes and ℓ 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 `GRPulse`. Assuming `GRPulse` has been properly compiled and the file called `myinput.txt`, then you can run with

```
./GRPulse myinput.txt
```

2.2.1 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.

You can see this examples of comment use within the sample files.

2.3 Parameters for stellar models

For the following model, on the same line as the model keyword, give the following parameters in order, separated by a space.

- `polytrope`
  1. real number for polytrope index n
  2. integer for number of grid points

  The next line should be `Params:` as above.

- `CHWD`

1. real number for initial value y_o (see `ChandrasekharWD`)
2. integer for number of grid points

Do **not** include `Params:` line.

- `MESA`

1. string with name of dat file (omitting the "`.dat`")
2. integer for number of grid points to use – this number is a *suggestion*

All `MESA` data files must be moved to the `GRPulse` directory. Do **not** include the `Params:` line.

2.4 The output

When the calculation begins, `GRPulse` 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.

2.5 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.

sampleinput2.txt Will create the post-newtonian results for $n = 1$ in Boston (2021)

sampleinput3.txt Will create the Newtonian results for $n = 2$ in Boston (2021)

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.

Part I

Base Classes

Chapter 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 $\Gamma_1 =$ `gam1`.

- `double getAstar(int X, double gam1)`: a dimensionless Schwarzschild discriminant $A* = -rA$, using $\Gamma_1 =$ `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 $\Gamma_1 = \left(\frac{\partial P}{\partial \rho}\right)_{\rm ad}$.

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 ρ, 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, ρ, P, Φ as columns in a tab-separated `.txt` file, then plots ρ 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 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

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

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

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

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

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

Chapter 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, Γ_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 ℓ given. The equation form `Mode<N>` expects is shown in (**??**) (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. **??** below and Sec. **??** 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 $\mathbf{A}$ and the frequency `w2`; this normally takes the form of a recursion algorithm involving series expansions of $\mathbf{A}$. 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. **??**.

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. **??** 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 ρ, 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.

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.

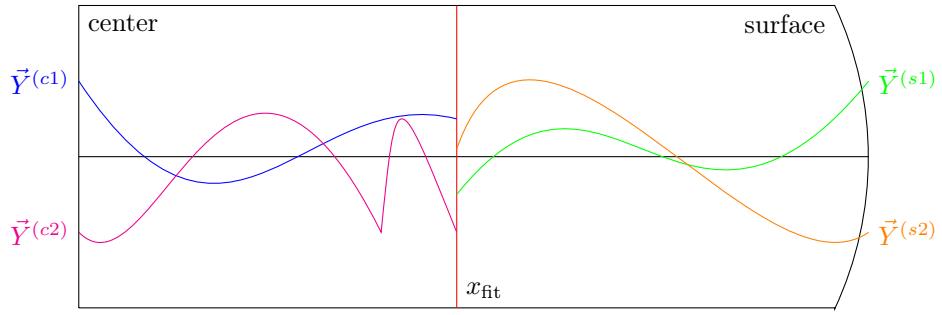


Figure 4.1: A sketch of the solution process for $N = 4$. The matching of the four independent solutions at x_{fit} allows for finding the eigenvalue $\bar{\omega}^2$ and the physical solution. To create the independent solutions, `getBoundaryMatrix()` gives `Mode<N>` a matrix for assigning boundary values.

Chapter 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 ℓ, m are the angular harmonic numbers. The mode can optionally have Γ_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 × 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, ℓ, m:

- int modeOrder(): radial mode order, k.
- void modeNumbers(int& k, int& l, int& m): replaces arguments with all mode numbers k, ℓ, m.

Methods related to the frequency:

- double getOmega2(): the square dimensionless frequency $\bar{\omega}^2$ ($\omega^2 = \frac{GM}{R^3}\bar{\omega}^2$).
- double getFreq(): returns the angular frequency ω 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] $= y_i$(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 $x, y_1, \ldots, y_{\mathtt{N}}$ to external files and forms plots in gnuplot. The point `xfit` where the double-shooting method is matched is highlighted by a red vertical line – 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 (**??**) 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 $\bar{\omega}^2 \in$ [`omegMin`, `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, $\bar{\omega}^2 =$ `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 $\bar{\omega}^2$ is picked based on the Pekeris formula for uniform stellar frequencies (see Cox, section 17.7), [1]

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

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

  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 $y_1, \ldots, y_{\mathsf{N}}$ 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

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

[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

$$\mathsf{ks} = \mathsf{k} - 2\mathsf{l} + 4$$

is used instead of `k` in (**??**) 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

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

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` **??**. This is a 4th-order system, so there must be four boundary conditions – two at the center and two at the surface (See Fig. **??**). 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)

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

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

and at the surface

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

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

The condition for these to match is that

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

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

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

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 (**??**) 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 (**??**) is satisfied for some choice of a, b, c. To find these values, write the homogeneous linear system of (**??**) 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

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

are the correct boundary values to use to find the physical solution. Other values are determined from the boundary conditions, (**??**),(**??**). 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.

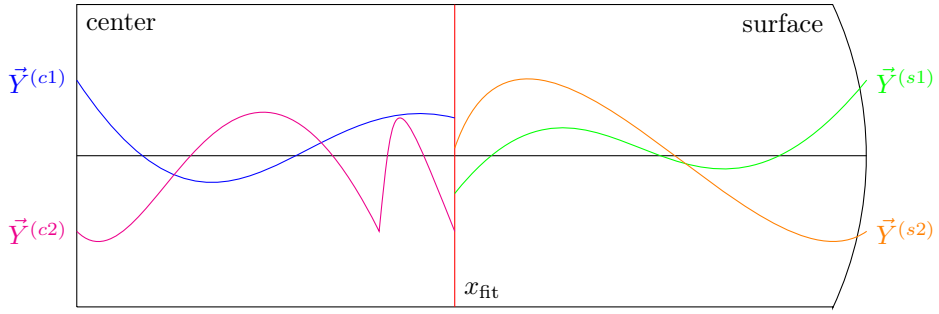


Figure 5.1: A sketch of the solution process for nonradial modes with `N` = 4 . The two BCs at each boundary give the two solutions in each region, and the values of these four solutions at $x_{\rm fit}$ are used in the Wronskian for find $\bar{\omega}^2$, then fit together to create the physical solution.

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

Stellar Models

Chapter 6

The `Polytrope` Class

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

$$\frac{1}{\xi^2}\frac{d}{d\xi}\left(\xi^2\frac{d\theta}{d\xi}\right) = -\theta^n, \tag{6.1}$$

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

$$\rho(r) = \rho_c\theta(r)^n \tag{6.2}$$

and where

$$\xi = r\sqrt{\frac{4\pi G\rho_c^2}{(n+1)P_c}}. \tag{6.3}$$

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

6.1 The `Polytrope` Interface

`Polytrope` is a child of `Star`, which means it must implement the `Star` interface – i.e., the functions returning ρ, P, Φ, the `printStar()` method, etc. as in **??**.

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

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

6.1.1 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 $G = \rho_c = P_c = 1$.

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

- `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.

6.2 The `Polytrope` Implementation

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

6.2.1 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 Γ of the polytrope, $\Gamma = 1 + \frac{1}{n}$, where $P \sim \rho^\Gamma$.
- `double rho0`: the central density ρ_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 $r = R_n \xi$.

6.2.2 Solution Variables

- `double **Y`: an array holding the solution to (**??**) 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` $\rightarrow \xi$, `y` $\rightarrow \theta$, `z` $\rightarrow \frac{d\theta}{d\xi}$, 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.

6.2.3 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 $0 \leq n < 5$. 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 $= \frac{\sqrt{6}}{\text{len}-1}$ (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 (??) to simplify the integration, noting

$$m(r) = \int_0^r 4\pi r^2 \rho(r) dr = \int_0^x 4\pi R_n^3 \xi^2 \rho_c \theta^n d\xi = 4\pi \rho_c R_n^3 \int_0^\xi \frac{d}{d\xi}\left(-\xi^2 \frac{d\theta}{d\xi}\right) = -4\pi R_n^3 \rho_c \xi^2 \frac{d\theta}{d\xi}; \tag{6.4}$$

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.

6.2.4 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 θ^n, the program can produce NaNs 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 θ[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 θ[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.

6.2.5 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 θ 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 θ 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:

$$A^*(r) = 0 + A_V\left(\frac{n+1}{3\Gamma_1}\right)\xi_1^2 x^2 + A_V\left(\frac{2(n+1)}{36\Gamma_1} - \frac{2n(n+1)}{60\Gamma_1}\right)\xi_1^4 x^4 + \cdots \tag{6.5}$$

$$U(r) = 3 + \left(-\frac{n}{5}\right)\xi_1^2 x^2 + 54n\left(\frac{7n}{8100} - \frac{1}{1296} + \frac{5-8n}{15120}\right)\xi_1^4 x^4 + \cdots \tag{6.6}$$

$$c_1(r) = \left(-\frac{3z_1}{x_1}\right) + \left(\frac{3nz_1}{10x_1}\right)\xi_1^2 x^2 + \left(\frac{n(2n+25)z_1}{1400x_1}\right)\xi_1^4 x^4 + \ldots \tag{6.7}$$

$$V_g(r) = 0 + \left(\frac{n+1}{3\Gamma_1}\right)\xi_1^2 * x^2 + \left(\frac{2(n+1)}{36\Gamma_1} - \frac{2n(n+1)}{60\Gamma_1}\right)\xi_1^4 x^4 + \cdots . \tag{6.8}$$

  Here ξ_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**.

6.3 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)`: Φ 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

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

  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 $V_g(r) = \frac{rg}{v_s^2} = -\frac{n+1}{\Gamma_1} * \frac{\xi}{\theta}\frac{d\theta}{d\xi} = -\frac{n+1}{\texttt{GamPert}}\frac{\texttt{Y[X][x] Y[X][z]}}{\texttt{Y[X][y]}}$. 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 $v_s^2 = \frac{\Gamma_1 P}{\rho} = \frac{\Gamma_1 P_c}{\rho_c} * \theta(x) = \frac{\Gamma_1 P_c}{\rho_c}$`Y[X][y]`, where $\Gamma_1 =$ `GamPert`. If no Γ_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.

6.4 References

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

Chapter 7

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

$$n_e = 2\frac{\int d^3p}{h^3} = \int_0^{p_F} \frac{8\pi\, p^2}{h^3} dp = \frac{8\pi\, p_F^3}{3h^3}, \tag{7.1}$$

where here p_F is defined from the energy

$$E_F = \sqrt{p_F^2 c^2 + m_e^2 c^4}. \tag{7.2}$$

Define the parameters x, θ by

$$x = \frac{p_F}{m_e c}, \qquad p = m_e c \sinh\theta. \tag{7.3}$$

In terms of this the electron number density becomes

$$n_e = \frac{8\pi}{3h^3} m_e^3 c^3 x^3 = \frac{8\pi}{3}\left(\frac{m_e c}{h}\right)^3 x^3. \tag{7.4}$$

We can use kinetic theory to calculate the electron pressure

$$P_e = \frac{1}{3}\int_0^{p_F} \frac{p^2 c^2}{E}\frac{8\pi\, p^2}{h^3} dp = \frac{8\pi m_e^4 c^5}{3h^3}\int_0^{\theta_F} \sinh^4\theta\, d\theta. \tag{7.5}$$

Most of the density is going to come from the baryons. Define a parameter μ_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,

$$\frac{1}{\mu_e} = \sum_i X_i \frac{Z_i}{A_i}, \tag{7.6}$$

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

$$\rho_B = \mu_e m_H n_e = \mu_e m_H \frac{8\pi}{3}\left(\frac{m_e c}{h}\right)^3 x^3 = B x^3, \tag{7.7}$$

where m_H is the mass of a hydrogen atom.

Define

$$f(x) = 8\int_0^{\theta_F} \sinh^4\theta d\theta = \left[(2x^3 - 3x)\sqrt{1+x^2} + \sinh^{-1} x\right]. \tag{7.8}$$

The electron pressure becomes

$$P_e = \frac{\pi m_e^4 c^5}{3h^3}\left[(2x^3 - 3x)\sqrt{1+x^2} + \sinh^{-1} x\right] = Af(x). \tag{7.9}$$

Note that

$$\frac{df(x)}{dx} = 8\frac{d\theta}{dx}\frac{d}{d\theta}\int_0^{\theta_F} \sinh^4\theta\, d\theta = \frac{8\sinh^4\theta_F}{\cosh\theta_F} = \frac{8x^4}{\sqrt{1+x^2}}. \tag{7.10}$$

Consider the equations of hydrostatic equilibrium:

$$\frac{d\Phi}{dr} = -\frac{1}{\rho}\frac{dP}{dr}, \qquad \frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = 4\pi G\rho. \tag{7.11}$$

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 ρ_B. Therefore, combining these,

$$\frac{1}{r^2}\frac{d}{dr}\left(\frac{Ar^2}{Bx^3}\frac{df(x)}{dr}\right) = \frac{1}{r^2}\frac{d}{dr}\left(\frac{Ar^2}{Bx^3}\frac{8x^4}{\sqrt{1+x^2}}\frac{dx}{dr}\right) = -4\pi GBx^3. \tag{7.12}$$

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

$$R_n = \sqrt{\frac{2A}{\pi GB^2}}. \tag{7.13}$$

Further define the new variable y by

$$y^2 = 1 + x^2. \tag{7.14}$$

Note

$$\frac{dy}{d\xi} = \frac{d}{d\xi}\sqrt{1+x^2} = \frac{x}{\sqrt{1+x^2}}\frac{dx}{d\xi}. \tag{7.15}$$

Then at last we have the equation

$$\frac{1}{\xi^2}\frac{d}{d\xi}\left(\xi^2\frac{dy}{d\xi}\right) = -(y^2-1)^{3/2}. \tag{7.16}$$

This is the equation for Chandrasekhar's white dwarf. It is very similar to the Lane-Emden equation (**??**), 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.

7.1 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 $y($`ξ[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.

7.1.1 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 (μ_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.

7.2 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`.

7.2.1 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 ξ is the independent variable of equation (**??**).

7.2.2 Solution Variables

- `double *xi`: an array with the position variable ξ from (**??**).
- `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 (**??**).
- `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]`.

7.2.3 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.

7.2.4 Calculation

- `double RK4integrate(int Len, double dx)`: integrate equation (**??**) 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 `NaN`s 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.

7.2.5 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 $s = r/R_\star = \xi/\xi_1$:
$$y(\xi) \approx y_0 + y_2 s^2 + y_4 s^4 + y_6 s^6 = \texttt{yc[0]} + \texttt{yc[1]}s^2 + \texttt{yc[2]}s^4 + \texttt{yc[3]}s^6.$$
- `double xc[2]`: coefficients of
$$x(\xi) \approx x_0 + x_2 s^2 = \texttt{xc[0]} + \texttt{xc[1]}s^2.$$
- `double fc[2]`: coefficients of
$$f(x(\xi)) \approx f_0 + f_2 s^2 = \texttt{fc[0]} + \texttt{fc[1]}s^2.$$
- `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**.

7.3 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)`: Φ 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 Γ_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 Γ_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 Γ_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.

7.4 References

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

Chapter 8

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.

GRPulse 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.

8.1 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.

8.1.1 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.

8.2 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.

8.2.1 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 $GM_\star/R_\star^2$.

8.2.2 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 Γ_1 from `MESA` column 10.

8.2.3 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 $U = \hat{\rho}x^3/\hat{m}$.

8.2.4 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)

$$r = R_\star x, \qquad m = M_\star \hat{m}, \qquad P = \frac{GM_\star^2}{4\pi R_\star^4}\hat{P}, \qquad \rho = \frac{M_\star}{4\pi R_\star^3}\hat{\rho}, \qquad g = \frac{GM_\star}{R_\star^2}\hat{g}. \tag{8.1}$$

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]`.

8.2.5 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 **??**.

Figure 8.1: 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.

8.2.6 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 $x = r/R_\star$,

$$\hat{T} = \hat{T}_0 + \hat{T}_2 x^2 + \hat{T}_4 x^4 + \cdots \tag{8.2a}$$

$$\hat{\rho} = \hat{\rho}_0 + \hat{\rho}_2 x^2 + \hat{\rho}_4 x^4 + \cdots \tag{8.2b}$$

$$= \hat{\rho}(0) + \left(\frac{\hat{P}_2 - \hat{T}_2 F_T}{F_\rho}\right) x^2 + \left(\frac{2\hat{P}_4 - 2\hat{T}_4 F_T - \hat{T}_2^2 F_{TT} - 2\hat{T}_2\hat{\rho}_2 F_{T\rho} - \hat{\rho}_2^2 F_{\rho\rho}}{2F_\rho}\right) x^4 \tag{8.2c}$$

$$\hat{P} = \hat{P}_0 + \hat{P}_2 x^2 + \hat{P}_4 x^4 + \cdots \tag{8.2d}$$

$$= \hat{P}(0) + \left(-\frac{1}{6}\hat{\rho}_0^2\right) x^2 + \left(-\frac{2}{15}\hat{\rho}_0\hat{\rho}_2\right) x^4 + \cdots, \tag{8.2e}$$

in terms of which

$$c_1 = \frac{3}{\hat{\rho}_0} - \frac{9}{5}\frac{\hat{\rho}_1}{\hat{\rho}_0^2} x^2 + \left(\frac{27}{25}\frac{\hat{\rho}_2^2}{\hat{\rho}_0^3} - \frac{9}{7}\frac{\hat{\rho}_4}{\hat{\rho}_0^2}\right) x^4 + \cdots \tag{8.3a}$$

$$U = 3 + \frac{6}{5}\frac{\hat{\rho}_2}{\hat{\rho}_0} x^2 + \left(\frac{12}{7}\frac{\hat{\rho}_4}{\hat{\rho}_0} - \frac{18}{25}\frac{\hat{\rho}_2^2}{\hat{\rho}_0^2}\right) x^4 + \cdots \tag{8.3b}$$

$$V_g = -\frac{2\hat{P}_2}{\hat{P}_0\Gamma_1} x^2 + \left(\frac{2\hat{P}_2^2}{\hat{P}_0^2\Gamma)1} - \frac{4\hat{P}_4}{\hat{P}_0\Gamma_1}\right) x^4 + \cdots \tag{8.3c}$$

$$A^* = \left(\frac{2\hat{P}_2}{\hat{P}_0\Gamma_1} - \frac{2\hat{\rho}_2}{\hat{\rho}_0\Gamma_1}\right) x^2 + \left(\frac{4\hat{P}_4}{\hat{P}_0^2\Gamma_1} - \frac{2\hat{P}_2^2}{\hat{P}_0^2\Gamma_1} - \frac{4\hat{\rho}_4}{\hat{\rho}_0} + \frac{2\hat{\rho}_2^2}{\hat{\rho}_0^2}\right) x^4 + \cdots \tag{8.3d}$$

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 θ around x (see Sec. **??**)

$$\theta = \theta_0 + \theta_2 x^2 + \theta_4 x^4 + \cdots \tag{8.4}$$

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.

8.3 Internal Logic Behind the Implementation

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

- `double rad(int X)`: $r =$ `Rtot` $*$ `radi[X]`.
- `double rho(int X)`: $\rho =$ `Dscale` $*$ `dens(X)`.
- `double drhodr(int X)`: $\frac{d\rho}{dr} =$ `Dscale`/`Rtot` $*$ `dens'(X)`.
- `double P(int X)`: $P =$ `Pscale` $*$ `pres(X)`.
- `double dPdr(int X)`: $\frac{dP}{dr} =$ `Pscale`/`Rtot` $*$ `pres'(X)`.
- `double Phi(int X)`: Not yet implemented (07/15/2021) .
- `double dPhidr(int X)`: $g = \frac{d\Phi}{dr} =$ `Gscale` $*$ `grav(X)`.
- `double mr(int X)`: $m(r) =$ `Mtot` $*$ `mass(X)`.
- `double Schwarzschild_A(int X, double GamPert)`: If `GamPert`$=0$, $A =$ `aSpline(X)/radi[X]/Rtot`. Otherwise,
$$A = \frac{\texttt{dens'(X)}}{\texttt{dens(X)*Rtot}} - \frac{\texttt{pres'(X)}}{\texttt{Gam1(X)*pres(X)*Rtot}}.$$
(Note units! A^* is dimensionless, but Schwarzschild discriminant is not.)
- `double getAstar(int X)`: If `GamPert`$=0$, $A^* =$ `aSpline(X)`. Otherwise
$$A = \frac{\texttt{radi[X]pres'(X)}}{\texttt{Gam1(X)*pres(X)}} - \frac{\texttt{radi[X]dens'(X)}}{\texttt{dens(X)}}.$$
- `double getU(int X)`: $U =$ `uSpline(X)`.
- `double getVg(int X, double GamPert)`: $V_g =$ `vSpline(X)/GamPert`. If `GamPert`$=0$, $V_g =$ `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, $v_s^2 = \frac{\texttt{Gam1(X)*pres(X)}}{\texttt{dens(X)}}$. Otherwise, $v_s^2 = \frac{\texttt{GamPert*pres(X)}}{\texttt{dens(X)}}$.

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])`.

8.4 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.

8.5 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_α column) of `tidal_overlap.txt`. See especially their Table 1.

Part III

Mode Drivers

Chapter 9

The NonradialModeDriver Class

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

$$\omega^2 \boldsymbol{\xi} = -\nabla\chi + \frac{\vec{A}}{v_s^2}(\nabla\cdot\boldsymbol{\xi}) \tag{9.1a}$$

$$\nabla^2\Delta\Phi = 4\pi G\Delta\rho. \tag{9.1b}$$

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

$$x\frac{dy_1}{dx} = [V_g - 3 + (2-\ell)]\,y_1 + \left[\frac{\ell(\ell+1)}{c_1\bar{\omega}^2} - V_g\right] y_2 + V_g y_3 \tag{9.2a}$$

$$x\frac{dy_2}{dx} = \left[c_1\bar{\omega}^2 - A^*\right] y_1 + [1 + A^* - U + (2-l)]\,y_2 - A^* y_3 \tag{9.2b}$$

$$x\frac{dy_3}{dx} = [1 - U + (2-\ell)]y_3 + y_4 \tag{9.2c}$$

$$x\frac{dy_4}{dx} = A^*U y_1 + UV_g y_2 + [\ell(\ell+1) - UV_g]\,y_3 + [(2-\ell) - U]y_4 \tag{9.2d}$$

where

$$y_1 = \frac{1}{x^{\ell-2}}\frac{\xi^r}{r}, \quad y_2 = \frac{1}{x^{\ell-2}}\frac{\chi}{gr}, \quad y_3 = \frac{1}{x^{\ell-2}}\frac{\Delta\Phi}{gr}, \quad y_4 = \frac{1}{x^{\ell-2}}\frac{1}{g}\frac{d\Delta\Phi}{dr}, \tag{9.3}$$

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

$$U = \frac{d\log g}{d\log r} + 2, \quad V_g = -\frac{1}{\Gamma_1}\frac{d\log P}{d\log r}, \quad A^* = \frac{1}{\Gamma_1}\frac{d\log P}{d\log r} - \frac{d\log\rho}{d\log r}, \quad c_1 = \frac{M}{m(r)}\left(\frac{r}{R}\right)^3. \tag{9.4}$$

We could also write this set of equations in the form

$$x\frac{d}{dx}\vec{Y} = x\frac{d}{dx}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{13} & 0 \\ 0 & 0 & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \mathbf{A}\vec{Y}. \tag{9.5}$$

using matrix notation. Notice that the LHS of (**??**) 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.

9.1 The NonradialModeDriver Interface

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

9.1.1 Constructor

There is only one constructor:

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

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

9.2 The `NonradialModeDriver` Implementation

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

9.2.1 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 Γ_1: if set to zero uses the background star's value of Γ_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.

9.2.2 Physics

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

- `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`.

9.2.3 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 $\mathbf{A}$ = `CCI` shown in (**??**).
- `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. **??** specifies values at each boundary as in (**??**). For the fourth-order nonradial mode, we pick

$$\vec{Y}^{(c1)} = (1, y_2, 0, 0), \vec{Y}^{(c2)} = (0, 0, 1, y_4), \vec{Y}^{(s1)} = (0, y_2, 1, y_4), \vec{Y}^{(s2)} = (1, y_2, 0, 0), \tag{9.6}$$

  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.

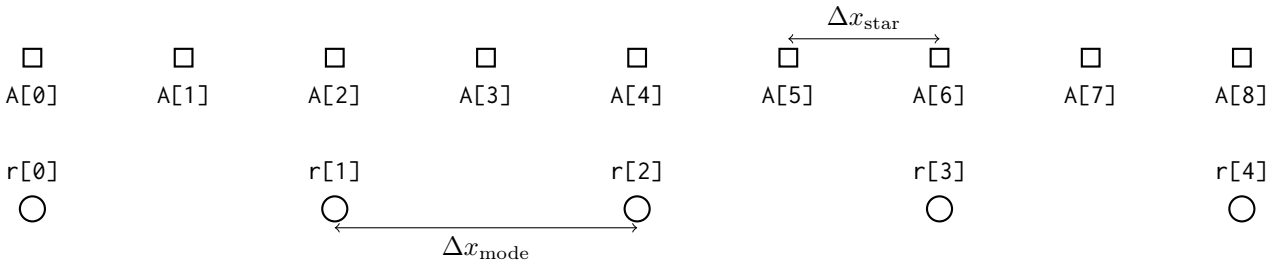


Figure 9.1: The squares are stellar grid points with spacing $\Delta x_{\rm star}$ containing values such as $A^* =$ `A[N]`. The circles are mode grid points with spacing $\Delta x_{\rm mode}$, and contain values such as $r =$ `r[N]` or mode values $y_1($`r[N]`$) =$ `yy[y1][N]`.

- `double SSR(double omeg2, int l, ModeBase* mode)`: calculate the RMSR in equations (**??**), where the variables in (**??**) double are calculated from the background star and the dimensionless variables in (**??**). 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 $\langle\vec{\xi}_1|\vec{\xi}_2\rangle = \int d^3r\rho\vec{\xi}_1^*\cdot\vec{\xi}_2$. 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).

9.2.4 Constructor

The only constructor does not do much more than initialize the arrays `A[]`,`U[]`,`C[]`,`V[]` described above and which appear in (**??**), 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]` $+ 0.5$`dx`. 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 **??**).

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,

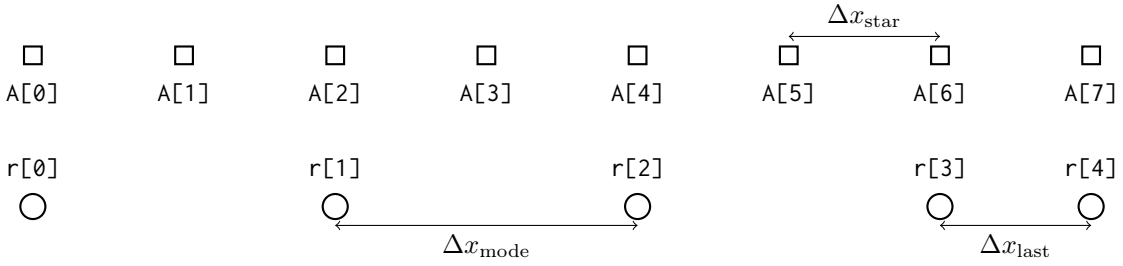


Figure 9.2: As in Fig. **??** the stellar grid is odd and can't be evenly divided. To fix this, the final grid spacing is $\Delta x_{\rm last} = \frac{1}{2}\Delta x_{\rm mode}$. Because of the power series solutions used there, this does not impact the accuracy of the calculation.

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. **??**). 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.

9.2.5 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.

9.2.6 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.

9.3 References

- Dziembowski, 1971, for the first publication of the equations in (**??**).
- Unno et al., *Nonradial Oscillations of Stars*, 1979, for detailed description of the equations (**??**) 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.

Chapter 10

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 (**??**) simplifies to the equations of motion for an oscillation mode

$$\frac{d\xi^r}{dr} = \left[\frac{g}{v_s^2} - \frac{2}{r}\right]\xi^r + \left[\frac{\ell(\ell+1)}{r^2\sigma^2} - \frac{1}{v_s^2}\right]\chi \tag{10.1a}$$

$$\frac{d\chi}{dr} = \left[\sigma^2 - N^2\right]\xi^r - A\chi \tag{10.1b}$$

which we rewrite in the form

$$x\frac{dy_1}{dx} = [V_g - 3 + (2-\ell)]y_1 + \left[\frac{\ell(\ell+1)}{c_1\bar{\omega}^2} - V_g\right]y_2 \tag{10.2a}$$

$$x\frac{dy_2}{dx} = [c_1\bar{\omega}^2]y_1 + [1 + A^* - U + (2-\ell)]y_3 \tag{10.2b}$$

or in matrix form

$$x\frac{d}{dx}\vec{Y} = x\frac{d}{dx}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \mathbf{A}\vec{Y}. \tag{10.3}$$

These variables have the same meaning as in (**??**).

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

$$\vec{Y}^{(c)} = (1, y_2), \vec{Y}^{(s)} = (1, y_3). \tag{10.4}$$

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. **??**.

10.1 References