Astr 450/550, Lecture Notes on MESA (2020 edition)

Joel Ong

Fall 2020

1 Introduction

MESA (short for "Modular Experiments in Stellar Astrophysics") is a comprehensive collection of stellar structure and evolution modelling codes, with the goal of permitting the modelling of self-gravitating (approximately) spherically symmetric objects, with masses ranging from ~Jupiter masses through several tens of solar masses. In order to do this correctly, one requires knowledge of various phenomena over a broad range of physical regimes, including but not limited to e.g. opacities, nuclear reaction rate constants (via cross-sections) and networks, thermodynamic equations of state, 1D approximations to 3D convective turbulence, etc. A comprehensive description of the underlying physics is beyond the scope of these notes (although much of it will be covered elsewhere in this course); suffice it to say that stellar modelling is in itself a fairly complicated affair. The appeal of MESA is that it collects the accumulated wisdom of the ages into a convenient place, for you to experiment with at your leisure.

In order to do this, MESA comes with a very large number of tunable parameters (and correspondingly large number of potential footguns). While it tries to choose sane defaults, I recommend you read through the list of default parameters (or ask the mailing list for help) as you familiarise yourself with its inner workings. The purpose of these notes is not to get you started with stellar modelling per se, but to get you sufficiently proficient with how to use MESA that you should not spend too much time figuring out how to get it to do what you want it to do (so that you can focus instead on figuring out what you want to do with it).

1.1 Technical requirements

These notes assume some elementary knowledge of Fortran, shell scripting, and generally getting a Unix system to do what you want it to do. If these are new to you, I provide links to some resources, to help you get up to speed:

- If you don't already have a preference for a shell, I strongly recommend `zsh` (as of macOS Catalina, the default on Apple systems, and also commonly available on most Linux distributions) or `bash` (the default on most Linux systems). `fish` is a popular alternative that is unfortunately not POSIX-compliant (i.e. scripts written for other shells will usually break when sourced with `fish`) but provides handy interactive functionality like intelligent autocomplete and history completions.
- A good shell-scripting cheat sheet can be found here.
- Assuming you are familiar with programming in at least one language (usually Python these days), you can find a good code-translation reference here.
- For text editing over an `ssh` connection, as you will have to do on the astro compute nodes, I personally recommend `vim`, but `emacs` and `nano` are also commonly used alternatives.
- The `mesasdk` provides a Fortran compiler, `gfortran`, as well as a standalone copy of e.g. the GNU autotools, so you will not need to worry about tooling. However, if you are used to the Python-centric Jupyter-based workflow, I highly recommend playing with LFortran, which can be used as a Jupyter kernel, to get some intuition for how Fortran works. In particular, LFortran supports plotting via matplotlib, which may help you with code introspection.
- Good references for Fortran syntax can be found here and here.

1.2 Human Resources

As is the case for many astronomical software projects, MESA is primarily authored by a very small number of volunteers (although it is technically open-source). You might want to know who some of these people are, in particular if you're interested in using MESA on a regular basis. If you send an email to the `MESA` mailing list asking for assistance or complaining about a bug, these are the people who will usually respond. In no particular order, the team includes:

- Bill Paxton (KITP, UCSB) — primary developer; core modules
- Frank X Timmes (ASU) — `eos` and mailing-list manager
- Josiah Schwab (UCSC) — miscellanous maintenance
- Richard H. D. Townsend (UWisc Madison) — `mesasdk` and `GYRE`
- Warrick H. Ball (Birmingham) — `astero` module

The `mesa-users` mailing list archive can be found at https://lists.mesastar.org/pipermail/mesa-users. I recommend signing up to receive mailing list updates at https://lists.mesastar.org/mailman/listinfo/mesa-users if you plan to use `MESA` regularly.

Finally, the `MESA` marketplace can be found at http://mesastar.org: if you use `MESA` for science, you should submit your setup here after your paper gets accepted; conversely, you can find `MESA` configurations for many published papers here, to use as a starting point for your own work. Don't forget to cite these, as well as the `MESA` and `GYRE` instrument papers, when submitting your own articles!

2 Installation

I have set up a working version of MESA on many of the astro department computers[1], but I include this section for the sake of completeness, since you may need eventually to run MESA on the HPC or on your own machines.

2.1 System Requirements

MESA is designed to run only on UNIX systems[2] — most Mac or Linux systems should be able to run it just fine. As of the time of writing, a full installation of the latest version of MESA (release 12778), along with the relevant tooling needed to actually run it, takes up a total of 10 GB. Moreover, MESA itself may take up to 2 GB of memory (depending on multiprocessing configuration and the kind of stellar model you're dealing with). If all else fails, you should be able to run it on the department compute resources, or the HPC.

2.2 Setting up the environment: mesasdk

MESA is written in modern Fortran, and therefore will require a Fortran compiler to operate. Unlike the horror stories you may have heard of FORTRAN 77, modern Fortran is surprisingly clean and easy to use, especially if you come from a numpy-heavy background (array broadcasting!).

Unfortunately, Fortran as a software ecosystem seems to have fallen by the wayside in many modern Unix distributions. While it's technically possible to build and run MESA using the standard gfortran that comes with your OS, this is strongly not recommended: e.g. Apple's version of gfortran is more than 15 years old at this point, and many of the libraries MESA uses are fairly obscure, and may not be commonly provided by most distributions[3]. Instead, the developer community has coalesced on a standardised collection of tools to be used with MESA, called the "MESA Software Development Kit" (or mesasdk for short). This is maintained by Richard Townsend at University of Wisconsin-Madison, and can be found at the following URL:

http://www.astro.wisc.edu/~townsend/static.php?ref=mesasdk

Once you have downloaded the version of the mesasdk appropriate to your Unix system of choice, you will extract its contents to a folder which I will refer to as $MESASDK_ROOT. Place this somewhere with fast read access (e.g. on an SSD; I don't recommend placing this on a network drive). For instance, I could do the following:

```
~ $ cd ~/Downloads
~/Downloads $ curl -O http://www.astro.wisc.edu/~townsend/resource/download/mesasdk/mesasdk-x86_64-linux-20.3.2
~/Downloads $ tar -xzf mesasdk-x86_64-linux-20.3.2.tar.gz mesasdk
~/Downloads $ mv mesasdk /opt
~/Downloads $ cd /opt/mesasdk
/o/mesasdk $ export MESASDK_ROOT=`pwd`
/o/mesasdk $ echo $MESASDK_ROOT
/opt/mesasdk
```

Once this is done, you will need to tell your shell to use the tools and libraries included with the mesasdk instead of the system ones. Do this by issuing the following command:

```
$ source $MESASDK_ROOT/bin/mesasdk_init.sh
```

The first time you run this command, the mesasdk will run a few installation scripts. Every time you want to use MESA, you should make sure that you have set this environment variable and sourced this file before attempting to run or execute the relevant programs.

Remark: If you use Arch Linux, mesasdk is available on the AUR; the package is maintained by Josiah Schwab, one of the active MESA developers.

[1]Not out of altruism, mind you; I use it myself of course.

[2]Tough luck, Windows users… the developers tried supporting it on Windows for a while, but decided in the end that it was too much effort.

[3]As of r12115, MESA relies on the crmath library in particular as an external dependency rather than bundling it, which broke my ability to build it on the standard Arch Linux toolchain.

2.3 MESA Proper

MESA itself is primarily written and maintained by Bill Paxton, and is hosted at

http://mesa.sf.net

Once you have downloaded the relevant release, you should again unzip it and place it somewhere with fast read access. Again, here's a worked example (assume I've downloaded the current release in advance):

```
~ $ cd ~/Downloads
~/Downloads $ unzip mesa-r12778.zip
~/Downloads $ mv mesasdk-r12778 /opt/mesa
~/Downloads $ cd /opt/mesa
/o/mesa $ export MESA_DIR=`pwd`
/o/mesa $ echo $MESA_DIR
/opt/mesa
```

Before you can use MESA, you will need to run the installation script:

```
/o/mesa $ ./install
```

Again, you will need to set the environment variable $MESA_DIR each time before you want to use MESA.

2.4 Worked Example

On many of the department machines, I have installed mesa and the mesasdk into /scratch11. Here's what this looks like on mo:

```
Last login: Tue May 19 08:35:47 2020 from 172.27.19.45
[jjo25@mo ~]$ uname -n
mo.astro.yale.edu
[jjo25@mo ~]$ ls -l /scratch11 | grep mesa
lrwxrwxrwx  1 jjo25 grad          11 Oct  2  2019 mesa -> mesa-r12115
drwxr-xr-x 30 jjo25 grad        4096 Apr 23 18:02 mesa-r12115
lrwxrwxrwx  1 jjo25 grad          16 Apr 23 17:57 mesasdk -> mesasdk-1:20.3.1
drwxr-xr-x 10 jjo25 grad        4096 Apr 13 12:08 mesasdk-1:20.3.1
drwxr-xr-x  9 jjo25 grad        4096 Aug 30  2019 mesasdk-20190830
[jjo25@mo ~]$
```

Each time I want to use MESA, I source the following file:

```
[jjo25@mo ~]$ cat ~/.bin/mesa.zsh
if [ -e /opt/mesa ]; then
    MESA_PREFIX="/opt"
elif [ -e /scratch11/mesa ]; then
    MESA_PREFIX="/scratch11"
else
    MESA_PREFIX="/scratch"
fi

export MESASDK_ROOT=$MESA_PREFIX/mesasdk
source $MESASDK_ROOT/bin/mesasdk_init.sh

export MESA_DIR=$MESA_PREFIX/mesa
export OMP_NUM_THREADS=4
export GYRE_DIR=$MESA_DIR/gyre/gyre
export aprgdir=$MESA_DIR/adipls/adipack.c
```

Some of these environment variables control how MESA behaves (e.g. $OMP_NUM_THREADS controls how many threads each MESA executable is allowed to use). For a full list and description of these, read the contents of $MESA_DIR/star/defaults/env_vars.list.

2.5 Common issues

2.5.1 HPC Setup

When setting up MESA on the HPC (grace in particular), you may run into problems during the installation process — specifically when running install, which may fail on the eos and kap modules. This is because the interactive sessions on the HPC are configured with memory limits that are too small to accommodate the full EOS and opacity tables, and by default the installation procedure attempts to test that all of the module return the correct results (by directly running some test suites). I recommend turning off testing for the relevant modules. For example, if the eos module is giving you issues, you can force the installer to skip testing by creating a file skip_test like so:

```
$ touch $MESA_DIR/eos/skip_test
```

Alternatively, you might run install on a machine with identical capabilities to the HPC machines before transferring your prebuilt copy of MESA to the HPC, although I don't recommend this.

2.5.2 Shared libraries

If you want to be able to use MESA capabilities (e.g. EOS and opacity tables) with external code, you will need to edit the file $MESA_DIR/utils/makefile_header to read

```
USE_SHARED = YES
```

You will need to do this **prior to compilation/installation**; if you have previously compiled/installed MESA without doing this, and later decide to enable this option, you will need to recompile everything. With this compilation flag set, MESA will produce a list of shared libraries in the directory $MESA_DIR/lib (e.g. libkap.so, librates.so etc.) which can be linked against for your own code that requires EOSes, opacities, rates, etc. An additional advantage is that the executable produced by the MESA compilation scripts will become much smaller (a few hundred KB vs. 50 MB). I highly recommend doing this if you're planning to run MESA executables from networked resources, if you're able to place $MESA_DIR on local storage, as it will greatly reduce the i/o-limited startup time.

3 Using MESA: the star module

Once all of this setup is done, you will want to use `MESA` to generate stellar models. Generically the workflow is based on the assumption that you will set up a working directory containing a "driver" executable that invokes the relevant `MESA` libraries and reads configuration files somewhere *other* than `$MESA_DIR`. There are two main sets of drivers which people mostly use as standalone tools:

- The `star` module, which is for modelling single stars and planets, and
- The `binary` module, which is for modelling binaries (we won't touch this in this course).

3.1 My first star

Copy the directory `$MESA_DIR/star/work` to somewhere with fast read *and* write access (I recommend a local scratch disk — note that your home directory on the department machines is hosted on a network drive, and will suffer from slow read/write access). We will refer to this as a "`work` directory". It contains the following:

```
[jjo25@houyi jjo25]$ cd /scratch/jjo25
[jjo25@houyi jjo25]$ cp -r $MESA_DIR/star/work .
[jjo25@houyi jjo25]$ ls -l work
total 48
-rwxr-xr-x. 1 jjo25 grad   19 May 22 10:39 clean
-rw-r--r--. 1 jjo25 grad  656 May 22 10:39 inlist
-rw-r--r--. 1 jjo25 grad  901 May 22 10:39 inlist_pgstar
-rw-r--r--. 1 jjo25 grad 1058 May 22 10:39 inlist_project
drwxr-xr-x. 2 jjo25 grad 4096 May 22 10:39 LOGS
drwxr-xr-x. 2 jjo25 grad 4096 May 22 10:39 make
-rwxr-xr-x. 1 jjo25 grad  130 May 22 10:39 mk
drwxr-xr-x. 2 jjo25 grad 4096 May 22 10:39 photos
-rwxr-xr-x. 1 jjo25 grad  254 May 22 10:39 re
-rwxr-xr-x. 1 jjo25 grad  165 May 22 10:39 rn
drwxr-xr-x. 2 jjo25 grad 4096 May 22 10:39 src
```

The notable contents of this directory are:

- A bunch of files called `inlist*`. These are **Fortran namelists**, a standard format for storing configuration variables for use with Fortran programs.
- A directory called `src`, which contains the Fortran source code that runs the `star` module. Right now all it contains is just a bunch of empty wrappers around the default behaviour.
- A directory `make`, containing recipes telling `gfortran` how to turn these source files into executables.
- A directory `LOGS`, where the output goes. You can change this as part of the configuration options.
- A directory `photos`, containing checkpoints.
- A few scripts:
  - `mk`: Compile the contents of the `src` directory
  - `clean`: Undo whatever it is that `mk` did
  - `rn`: start a fresh run of `MESA`'s star module from the supplied configuration files
  - `re`: resume from an earlier checkpoint

To get started with running `MESA`, you will first need to compile the `star` executable. Do this by running the `mk` script:

```
[jjo25@houyi jjo25]$ cd work/
[jjo25@houyi work]$ . ~/.bin/mesa.zsh
[jjo25@houyi work]$ ./mk
gfortran -Wno-uninitialized -fno-range-check -fmax-errors=7  -fprotect-parens -fno-sign-zero -fbacktrace -ggdb
gfortran -Wno-uninitialized -fno-range-check -fmax-errors=7  -fprotect-parens -fno-sign-zero -fbacktrace -ggdb
gfortran -Wno-uninitialized -fno-range-check -fmax-errors=7  -fprotect-parens -fno-sign-zero -fbacktrace -ggdb
gfortran -fopenmp -o ../star run_star_extras.o run_star.o  run.o  -L/opt/mesa/lib -lstar -lgyre -lionization -l
```

```
[jjo25@houyi work]$
```

If you have everything set up correctly, you will produce a binary executable called `star` in the work directory:

```
[jjo25@houyi work]$ ls -lh star
-rwxr-xr-x. 1 jjo25 grad 319K May 22 10:57 star
```

Rather than run this executable directly, you will instead call it indirectly via a “driver” program called `rn` (“run”). Invoke it as `./rn`, and watch the fireworks! (or not...)

With the default set of input namelists, what you should see is the evolution of a 15 $M_{\odot}$ stellar model from a pre-main-sequence collapsing ball of gas right up to the ignition of hydrogen fusion in the core. If you've done everything right up to this point, you will see a bunch of diagnostic plots pop up, showing the evolution of the star on a HR diagram, as well as an instantaneous snapshot of the star's pressure-temperature profile.

3.2 Configuration

By default, the `star` executable will read the file named `inlist`; in this bare work directory, the `inlist` file points at other subsidiary configuration files, each controlling different aspects of `MESA`‘s behaviour. `inlist_pgstar` controls a bunch of plotting options, which may be useful if you want to inspect the stellar evolution 'live' as it's happening. We won't be using it very much for the exercises in this course, but I strongly recommend finding a free afternoon to play with the different plotting options associated with the `pgstar` namelist, and then building some intuition for stellar evolution by trying out different initial conditions.

These initial conditions are handled with the `controls` and `star_job` namelists. Let's take a look at `inlist_project` for now:

```
! inlist to evolve a 15 solar mass star

! For the sake of future readers of this file (yourself included),
! ONLY include the controls you are actually using.  DO NOT include
! all of the other controls that simply have their default values.

&star_job

  ! begin with a pre-main sequence model
    create_pre_main_sequence_model = .true.

  ! save a model at the end of the run
    save_model_when_terminate = .false.
    save_model_filename = '15M_at_TAMS.mod'

  ! display on-screen plots
    pgstar_flag = .true.

/ !end of star_job namelist


&controls

  ! starting specifications
    initial_mass = 15 ! in Msun units

  ! options for energy conservation (see MESA V, Section 3)
     use_dedt_form_of_energy_eqn = .true.
     use_gold_tolerances = .true.
  ! stop when the star nears ZAMS (Lnuc/L > 0.99)
    Lnuc_div_L_zams_limit = 0.99d0
    stop_near_zams = .true.

  ! stop when the center mass fraction of h1 drops below this limit
    xa_central_lower_limit_species(1) = 'h1'
    xa_central_lower_limit(1) = 1d-3

/ ! end of controls namelist
```

This file illustrates some properties that all well-formed Fortran namelist files must have:

- Like in modern Fortran, comments are denoted by exclamation marks (`!`). There are no multi-line comments.
- Each file contains at least one namelist (marked out with ampersand `&` and forward-slash `/` characters), and may contain more than one namelist. In this case, this file contains one `star_job` namelist and one `controls` namelist. Note that it's possible for a star to contain more than one of a given namelist; e.g. it is valid to have two different sections both named `controls`.
- Within each namelist, configuration values are specified as `variable_name = value`. These must be parseable as modern Fortran, which I briefly summarise. Variable names are case-insensitive and may only take alphanumeric ASCII values. Arrays begin at 1 and are indexed via parentheses. Double precision is specified by marking the exponent with a `d` (i.e. `1d0` instead of `1e0`). Booleans are marked `.true.` or `.false.`.

Every one of `MESA`'s configuration variables has a default value, which can be found in `$MESA_DIR/star/defaults`, in a file corresponding to the namelist. Thus, the default values for the `controls` namelist can be found in `$MESA_DIR/star/defaults/controls.defaults`, and so forth. Generically speaking:

- The `pgstar` namelist controls `MESA`'s plotting capabilities,
- The `star_job` namelist controls setup/teardown logistics, and
- The `controls` namelist controls behaviour (e.g. physics and timestepping) during the simulation.

There is some perhaps nonintuitive overlap between these (e.g. stopping conditions are mostly found in the `controls` rather than `star_job` namelist), but this description is sufficiently accurate for most use cases. When in doubt, search the relevant `defaults` file (which is full of self-documenting comments — `grep` is your friend), and you'll usually find something useful eventually.

Remark: By default *all* stopping conditions will be checked for at every timestep, and `MESA` will terminate the evolution if any single one of them is triggered. This can be overridden by setting the `star_job` parameter `required_termination_code_string`.

Exercise

Set up a work directory for the following scenario:

- `star_job`:
  - Chemical abundances and opacities of Grevesse & Sauval (1998)
    - hint: search for `kappa_file_prefix`
  - Begin the evolution from a pre-main-sequence model with a central temperature of 10^5 Kelvin.
- `controls`:
  - Initial mass of 1 $M_\odot$
  - Initial helium abundance of 0.27
  - Mixing length parameter of 1.83
  - Enable diffusion and settling of heavy elements
  - Stop at the current solar age.

Once you've set this up, try running `MESA` as `./rn` and see what happens. If your machine permits, you may speed up the computation by enabling multithreading (set the environment variable `OMP_NUM_THREADS` to the number of threads you'll permit each instance of `star` to use); you will usually get the most performance gains with a power-of-two number of threads running (although also there are diminishing returns owing to Amdahl's Law).

3.3 Recovering output from `MESA`

As `MESA` runs, it will produce output in a log directory (`LOGS` by default) in addition to the stuff it prints to the terminal. There are two main forms of output:

- a "history" file, by default named `history.data`, containing global properties of the star (mass, radius, age, etc). I use the word "global" here in a very wide sense. Generally, anything that can be reduced to a single number at a given instant in time (including core temperature, surface metallicity, or integrals over the entire structure) can be meaningfully written out to the history file. Enabled if `do_history_file` is set to `.true.`, with a new line every n timesteps specified by `history_interval`.
- a series of "profile" files, by default named `profile%d.data` (e.g. `profile1.data`, `profile2.data`, etc). Each profile file contains a description of the structure of the star (i.e. radius, density, pressure, compositions, etc. as a function of the mass coordinate). Enabled if `write_profiles_flag` is set to `.true.`, with a new file every n timesteps specified by `profile_interval`.

Both of these output files are by default text-based, and can be inspected as e.g. `less -Sx32 history.data`. The quantities written out to these files are controlled by the configuration files named `history_columns.list` and `profile_columns.list` — if these are absent, then the default ones in `$MESA_DIR/star/defaults` are used. Each of these files contains a "header" section and a "body" section. For history files, the header contains information about the provenance of the evolutionary track (what version of MESA was used, when the computation was run, etc), while the profile files may also contain some global properties associated with the stellar model. If profiles are being written out, then a file named `profiles.index` will also be created, telling you which model (column `model_number` in `history.data`) corresponds to which profile.

Remark If an empty `profile_columns.list` file is supplied, then no attempt will be made to write profiles, even if `write_profile_flag` is set to `.true.`.

Exercise

For the evolutionary track you have constructed in the exercise above, plot a theoretical HR diagram (log luminosity vs. effective temperature, with temperature increasing from right to left). Mark out the solar point on the same diagram: $\log(L/L_\odot) = 0, T_{\text{eff},\odot} \sim 5777$ K. You may find some of the code snippets in the appendix to be helpful.

Exercise

From this evolutionary track, select the stellar model closest to the solar point. For this stellar model, make a plot of the superadiabatic gradient, $\nabla - \nabla_{\text{ad}}$, against the acoustic radial coordinate, $t(r) = \int_0^r \mathrm{d}r/c_s$.

Hint: None of these quantities are written out to the profile file by default, so you will need to supply a modified `profile_columns.list` file. The column names you will need are `acoustic_radius`, `actual_gradT` and `grada`.

3.4 Automation

Now that you know how to call `MESA`, it's time to do useful things with it. The most immediately useful thing to figure out next is how to automate calls to `MESA` (so that you can use it as a building block for more complicated things). The generic approach here is to

1. Make a copy of a template 'work' directory,
2. Programmatically modify the `inlists` of that `work` directory,
3. Automate calls to `rn` and/or `re` as appropriate.

For example, I might want to halt the evolution upon ignition of hydrogen fusion in the core, and restart the evolution afterwards, so that I can treat the pre-MS and MS phases of evolution with different model physics.

Exercise

Run the same physical scenario as described above, but with different choices of model physics before and after the onset of core hydrogen burning. In particular:

- Prior to the onset of core hydrogen burning, use:
  - A fixed atmospheric opacity relation
  - No element diffusion
- After the onset of core hydrogen burning, use:
  - A varying atmospheric opacity relation
  - Element diffusion
  - In `star_job`: set the stellar age to zero (so that the output `star_age` will be time since ZAMS).

Exercise

Reproduce the plot shown in Figure 1. To save time, you may truncate the evolution at an effective temperature of 4500 K. Ideally, you should separate this into two programs: one to call `MESA` appropriately, and one to actually generate the plot from the output. You may find some of the code snippets in the appendix to be helpful.

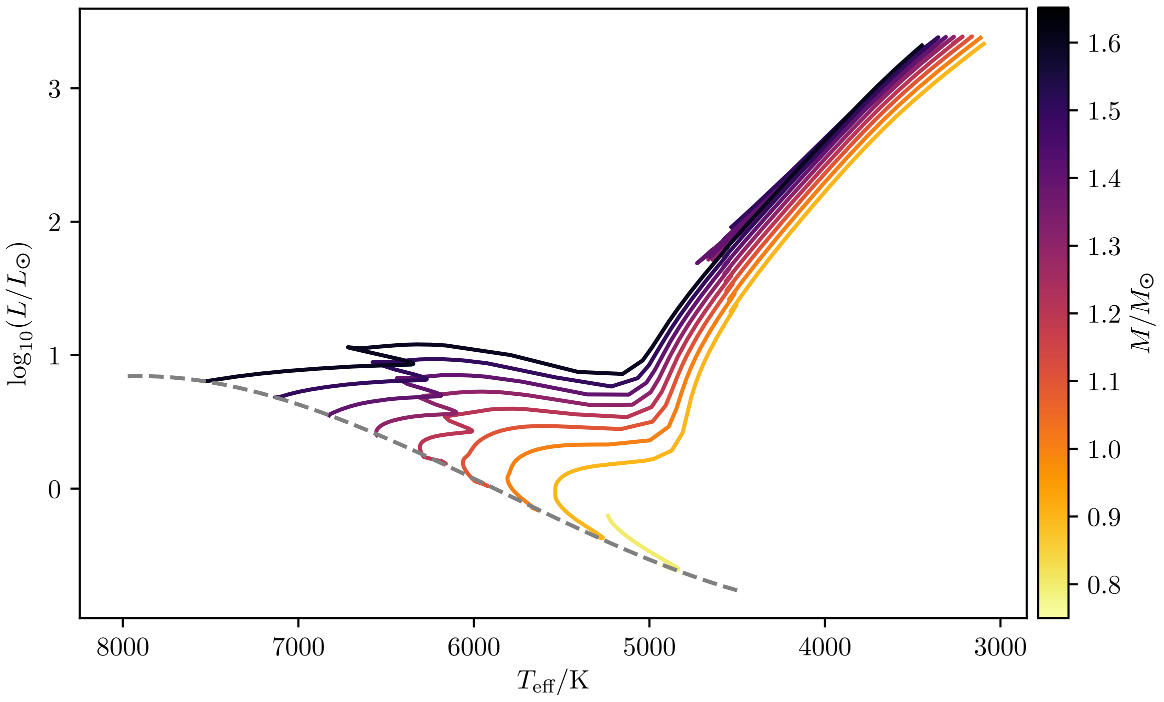


Figure 1: This plot shows the evolution (on the theoretical HR diagram) of stars with identical composition and mixing-length parameter to the one in the preceding exercise, for stellar masses between 0.8 and 1.6 solar masses (in steps of 0.1 solar masses). A fit to the loci of the zero-age main sequence is shown with the grey dashed line.

Exercise

Do the same for a series of 1 $M_\odot$ evolutionary tracks, *ceteris paribus* except for the initial helium abundance (from 0.25 to 0.32 in steps of 0.05).

Exercise

Do the same for a series of 1 $M_\odot$ solar-abundance evolutionary tracks, *ceteris paribus* except for the mixing length parameter (1.6 to 2.1 in steps of 0.1).

3.5 Common Issues

3.5.1 Hydrostatic solver convergence failure during helium flash

As of r11532, evolution through the tip of the RGB to the HB/RC will fail to converge (usually resulting in the solver trying smaller and smaller step sizes before giving up). This is because this release introduces stricter energy conservation for stellar evolution, at the potential expense of smaller step sizes. Since the tip of the RGB/helium flash yields sudden changes in the energy generation rate, the required stepsizes to comply with these energy conservation requirements can get unphysically small (of order several seconds). There are two possible resolutions to this:

- Setting `convergence_ignore_equL_residuals = .true.` — this option to ignore the luminosity term in the energy conservation equation is **undocumented**, but used by all of the relevant test suite examples.
- Setting `use_gold_tolerances = .false.`, reverting energy conservation checking to the behaviour of r10398. This will give you a warning every timestep, unfortunately. On the flip side, this is also *considerably* faster, since it allows the use of much larger timesteps.

3.5.2 Element diffusion for high-mass stars

For high-mass stars (in particular, with very thin or no convective envelopes), element diffusion at the surface is so efficient that essentially all helium and heavy elements get drained away from the surface, resulting in predicted spectroscopic metallicities that are unphysically low. Again, there are multiple possible resolutions to this, including

- Changing the diffusion efficiency in such a way that diffusion is effectively inoperative above 1.5 $M_\odot$. See e.g. Eq. (1) of Viani et al. (2018) for an example parameterisation. Unfortunately this can't be done with namelist options in `MESA`, although it may be implemented with custom physics hooks (as in the next section).
- Setting `diffusion_min_dq_at_surface` to a much larger value than the default (e.g. 10^{-3}).

3.5.3 I/O-limited startup

Under some circumstances, `MESA` may suddenly consume a large amount of memory very briefly, and also read/write heavily to the work directory (as opposed to `$MESA_DIR`). This usually happens when it's caching various tables, such as the EoS or opacity tables, when they're encountered for the first time running a given working directory. While this is a priori a good thing (the caches on disk can be read directly into memory instead of reconstructing the tables from the data files every time, which speeds up future runs), this behaviour is undesirable where there are many working directories running in parallel, or when the working directories are on slow storage devices, both of which are often the case when using `MESA` on HPC systems. This behaviour can be disabled by setting the following environment variable:

```
MESA_TEMP_CACHES_DISABLE=1
```

3.5.4 Exceeding storage limits

By default, `MESA` will limit you to writing out at most 100 profiles (which can be changed by the `controls` option `max_num_profile_models`; set it to a negative value to remove the limit). For many modelling applications, it is desirable not to limit the number of models written out, but on the flip side this means that the risk of running out of disk space is real. This ultimately boils down to user error. **Be very careful about this**.

3.6 Further reading

I recommend Josiah Schwab's introductory lectures, prepared for the `MESA` Summer Schools every year. You can find them at e.g. https://jschwab.github.io/mesa-2019/, for different years going back to 2014 or so.

4 Extending MESA: run_star_extras.f

Inside a plain work directory, you will find two Fortran source files in the src directory. One of them, run.f, simply invokes the star module; you will in most cases not need to touch this. The other, run_star_extras.f, contains hooks to user-customisable routines. This is what it looks like by default:

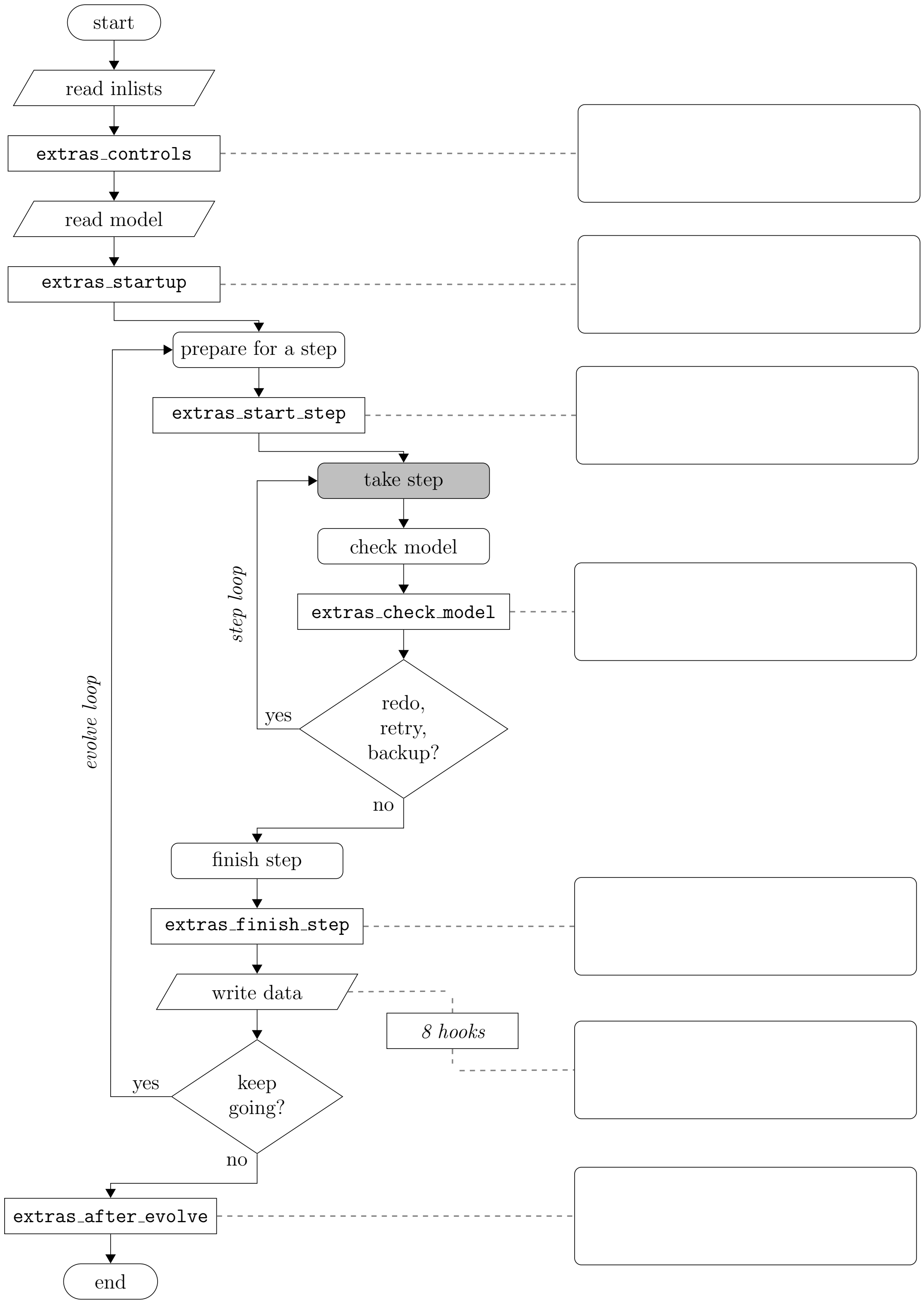
```fortran
! run_star_extras.f
      module run_star_extras

      use star_lib
      use star_def
      use const_def
      use math_lib

      implicit none

      ! these routines are called by the standard run_star check_model
      contains

      include 'standard_run_star_extras.inc'

      end module run_star_extras
```

It is helpful to try to understand what is going on here. When you compile MESA, you are creating an executable called star in your work directory, that uses a module (run_star) that resides in $MESA_DIR, and which was compiled at installation. This module in turn usees a module called run_star_extras that does *not* reside in $MESA_DIR, and was not compiled at installation. Instead, it is supplied from definitions given in the src subdirectory of the work directory — hence the module definition — and is compiled at the same time as the star executable. run_star expects run_star_extras to satisfy a known API — in this case, to provide a standard named set of procedures, all of which have reference implementations in the file standard_run_star_extras.inc. These can be overriden by the user at compile time to provide custom behaviour, with more granularity and flexibility than afforded by the usual namelist files. This choice of architecture allows user customisability without requiring you to directly modify the contents of $MESA_DIR, so that different work directories run from the same installation of MESA can have different custom behaviours.

Unfortunately, Fortran will raise compilation errors for multiply-defined procedures (i.e. you can't define a function twice and hope the compiler picks the later definition). In order to do anything useful, therefore, the include statement must be replaced with the entire contents of $MESA_DIR/include/standard_run_star_extras.inc. If you use vim, here is a sequence of steps to follow to do this:

- In normal mode, place the cursor on the line with the include statement
- Issue the action dd (delete this line)
- Issue the command :r $MESA_DIR/include/standard_run_star_extras.inc (read the contents of the specified file to my cursor location). vim supports both tab-autocomplete and environment variables in tab-autocomplete, so you don't actually need to type all of this.
- Bonus: if you've done this once, you can use history substring searching to repeat this action in future

If you've done this right, you will see a long list of procedure definitions (functions and subroutines) appearing. Each of these procedures permits modification of various steps MESA takes each timestep (or attempt at timestep), and at the start and end of the evolutionary calculation. I reproduce the following diagram from Josiah Schwab's lecture notes:



In fact, each of the procedures in the above figure can also be overriden by modifying `extras_controls`, which sets pointers to them. I excerpt it below:

```fortran
      subroutine extras_controls(id, ierr)
         integer, intent(in) :: id
         integer, intent(out) :: ierr
         type (star_info), pointer :: s
         ierr = 0
         call star_ptr(id, s, ierr)
         if (ierr /= 0) return

         ! this is the place to set any procedure pointers you want to change
         ! e.g., other_wind, other_mixing, other_energy  (see star_data.inc)


         ! the extras functions in this file will not be called
         ! unless you set their function pointers as done below.
         ! otherwise we use a null_ version which does nothing (except warn).

         s% extras_startup => extras_startup
         s% extras_start_step => extras_start_step
         s% extras_check_model => extras_check_model
         s% extras_finish_step => extras_finish_step
         s% extras_after_evolve => extras_after_evolve
         s% how_many_extra_history_columns => how_many_extra_history_columns
         s% data_for_extra_history_columns => data_for_extra_history_columns
         s% how_many_extra_profile_columns => how_many_extra_profile_columns
         s% data_for_extra_profile_columns => data_for_extra_profile_columns

         s% how_many_extra_history_header_items => how_many_extra_history_header_items
         s% data_for_extra_history_header_items => data_for_extra_history_header_items
         s% how_many_extra_profile_header_items => how_many_extra_profile_header_items
         s% data_for_extra_profile_header_items => data_for_extra_profile_header_items

      end subroutine extras_controls
```

The generic structure of all of these procedures is as follows:

1. The first few lines are **variable declarations** (aka type statements). We have an integer `id` (as more than one `star` can be in play, e.g. in a `binary` simulation), an integer error code, and a pointer to a `star_info` object named `s`.
2. We initialise values: the error code is set to 0, and the pointer `s` is mapped to the actual `star_info` object. If this fails, `ierr` is set to a nonzero value, and the procedure will exit. If this happens your `star` executable will also exit with an error.
3. Various attributes of the `star_info` object `s` are set (yes, you can do object-oriented programming in Fortran!). The only things being done here are setting various pointers to procedures defined elsewhere in this file. To customise behaviour, either modify these procedures, or change the pointer to point at a different, user-supplied procedure.

This can be a little overwhelming at first, so I will break things down by common use cases, starting from the back.

4.1 Custom output

The easiest use case for `run_star_extras` is to get information into your history/profile files that isn't already accessible as a predefined column in the provided column lists. I will only discuss the history files, as the procedure is very similar for all of them. Here are the relevant functions:

```
      integer function how_many_extra_history_columns(id)
         integer, intent(in) :: id
         integer :: ierr
         type (star_info), pointer :: s
         ierr = 0
         call star_ptr(id, s, ierr)
         if (ierr /= 0) return
         how_many_extra_history_columns = 0
      end function how_many_extra_history_columns


      subroutine data_for_extra_history_columns(id, n, names, vals, ierr)
         integer, intent(in) :: id, n
         character (len=maxlen_history_column_name) :: names(n)
         real(dp) :: vals(n)
         integer, intent(out) :: ierr
         type (star_info), pointer :: s
         ierr = 0
         call star_ptr(id, s, ierr)
         if (ierr /= 0) return

         ! note: do NOT add the extras names to history_columns.list
         ! the history_columns.list is only for the built-in history column options.
         ! it must not include the new column names you are adding here.


      end subroutine data_for_extra_history_columns
```

To get your custom data into the history file, you will need to:

1. Change the return value of `how_many_extra_history_columns` (e.g. 1).
2. In the subroutine `data_for_extra_history_columns`, set the name and value of this column after the boilerplate preamble.

This can get arbitrarily complex (e.g. you may invoke external procedures). Properties of the star can be accessed via attributes of the `star_data` object, and the names are usually intuitive (e.g. `s% r` for an array of radial coordinates; `s% T`, `s% rho`, `s% Pgas` for the usual thermodynamic quantities). Check the file `$MESA_DIR/star_data/public/star_data.inc` for definitions and units (usually CGS. but with some exceptions where solar units are more conventionally intuitive).

4.1.1 Worked Example

`MESA` doesn't have a history column for metallicity, defined in the sense of

$$[Z/X] = \log_{10}(Z/X) - \log_{10}(Z_\odot/X_\odot)\,.$$

If you think you know the solar value, you might want to instruct `MESA` to write out metallicities in the following way:

```
! in subroutine data_for_extra_history_columns
    ! before star_ptr is called
     real(dp) :: x, y, z
    ! in body
     names(1) = "FeH"
     x = s% surface_h1
     y = s% surface_he3 + s% surface_he4
     z = 1 - x - y
     vals(1) = safe_log10_cr(z/x / 0.023) ! vs. GS98 value
```

Of course, you don't actually need to do this; you could easily obtain the same data post-hoc from the history columns `log_surf_cell_z` and `log_surface_h1`. However, you might want to do this if you intend to use this to affect the stellar evolution, e.g. as a custom stopping condition.

Note: I have used the function `safe_log10_cr`, which is provided by the `crmath` library included with the `mesasdk`, to perform the logarithm in such a way that the final bit in the floating-point representation is correctly rounded. Generically `MESA` always tries to do this where it can.

Exercise

One possible way of defining the base of the red giant branch is by computing an empirical discriminant

$$G = \frac{\mathrm{d}\log L}{\mathrm{d}\log T},$$

with respect to the theoretical HR diagram (i.e. along the evolutionary history of the star). In particular, the luminosity rises sharply as temperature decreases along the Hayashi track, so this discriminant takes strongly negative values for red giants. Empirically, $G \leq -5$ is a good discriminant between subgiants and red giants.

Calculate this quantity for each timestep, and write it out as a supplementary history column.

Hint: $\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\dot{y}}{\dot{x}}$.

Exercise

A naive implementation of the above will be met with a nasty surprise when the computation is stopped and restarted! This is because any custom module-scope variables that you may have defined (in order to store the immediate history of the evolutionary track) are not saved to snapshots, and are undefined upon restarts.

For simple cases like this, `MESA` allows you to store additional data in the `star` pointer, which can be referenced as `s% xtra(n)` (`real`), `s% ixtra(n)` (`integer`), or `s% lxtra(n)` (`logical`). Each of these are arrays. For readability, I recommend defining integer constants in your module preamble (i.e. `integer, parameter :: my_quantity = 1`) and then indexing these arrays semantically (`s% xtra(my_quantity) = do_thing()`).

For more sophisticated cases, e.g. when the quantities in question cannot be easily represented by scalars, it may be desirable to store these variables outside of the star pointer. In such cases it might be better to directly customise the i/o routines associated with reading and writing snapshots. `MESA` provides hooks to do this, although they're less well-documented:

```
module run_star_extras

! other boilerplate here; e.g. implicit none
! custom array variable definition

real(dp), dimension(:), allocatable :: custom_variable

contains

! In order to store/retrieve this variable in snapshots,
! we need to customise the photo read/write subroutines.
! this implementation is from the conductive_flame test suite example.

subroutine extras_photo_read(id, iounit, ierr)
   integer, intent(in) :: id, iounit
```

```
      integer, intent(out) :: ierr
      ierr = 0
      read(iounit) custom_variable
   end subroutine extras_photo_read


   subroutine extras_photo_write(id, iounit)
      integer, intent(in) :: id, iounit
      write(iounit) custom_variable
   end subroutine extras_photo_write


   subroutine extras_controls(id, ierr)
      integer, intent(in) :: id
      integer, intent(out) :: ierr
      type (star_info), pointer :: s
      ierr = 0
      call star_ptr(id, s, ierr)
      if (ierr /= 0) return

      ! this is the place to set any procedure pointers you want to change
      ! e.g., other_wind, other_mixing, other_energy   (see star_data.inc)

      s% other_photo_read => extras_photo_read
      s% other_photo_write => extras_photo_write
```

Armed with this knowledge, modify your solution to the previous exercise so that it continues to work across restarts.

Remark: In previous versions of `MESA`, this was done with a subroutine called `extras_move_info`, with helper subroutines `move_int`, `move_dbl` etc. This was largely limited to storing scalars. The API for this still exists, but is no longer recommended.

4.2 Custom controls

From the above flowchart, we see that each timestep can be attempted multiple times (mediated by `extras_check_model`) before proceeding to the next one, which can be modified by `extras_finish_step`. They differ in the following ways:

- In `extras_check_model`, you may direct `MESA` to either keep going (i.e. the step is acceptable), redo (repeat the timestep under consideration), retry (repeat the timestep under consideration with a smaller stepsize), backup (redo the previous timestep), or terminate (exit the loop).
- In `extras_finish_step`, you may only direct `MESA` to either keep going or terminate.

Generically, use `extras_check_model` to determine if the timestep is acceptable, and use `extras_finish_model` to do stuff after an acceptable timestep.

You can do basically anything to the `star_data` structure inside each of the functions. Common examples are:

- Changing the return code of the function depending on the state of the `star_data` structure
- Changing controls associated with the `star_data` structure (e.g. changing physics based on evolutionary state instead of using the input namelists)
- Directly modifying physical properties of the `star_data` structure (e.g. depositing mass into the outer layers)

However, I recommend that you limit yourself to only manipulating the logistical rather than physical properties of the `star_data` structure, since any modifications to the stellar model proper will probably not be physically consistent, e.g. knocking it out of hydrostatic equilibrium.

4.2.1 Worked Example

Note: This example is adapted from Josiah Schwab's lecture notes.

One common use case for `run_star_extras` is to implement stopping at a predetermined condition. A blunt way of doing this would be to make a check for if that condition is met in `extras_finish_step`, and to issue a termination return code if it is the case. However, occasionally it may be desirable to have some fine-tuning in the process. For example, we might be tracking the evolution of a star up the red giant branch, and wish to terminate the evolution *when*, and not *after*, the star has achieved a particular radius.

One possible way of doing it is as follows: we allow the user to specify a target radius. `MESA` allows extra data to be supplied in the `controls` namelist via namelist array items `x_ctrl(n)` (for real values), `x_logical_ctrl(n)` (for boolean values), and `x_integer_ctrl(n)` (for integer values). These are all initialised to `0` (or `.false.`) by default. In this example, we will implement a custom stopping condition for the radius, such that we will halt the evolution "*at*" a predetermined value of the stellar radius, supplied by the user in solar units via `x_ctrl(1)`. I use scare quotes because we can't do this precisely, so we'll have to supply some tolerances. In this example, I won't let the user determine these tolerances, but (depending on how complicated you want to go) you might elect to do so for your own purposes.

Per the above flowchart, the only procedure that allows me to retry a timestep with smaller stepsize is `extras_check_model`. What we will need to do are:

1. Check that we have tripped the switch (i.e. the radius at this timestep exceeds the specified radius)
2. Check that the model's radius is within acceptable tolerances
3. If not, retry the step with a smaller stepsize.

Given that the radius was smaller than the threshhold at the start of the step and larger than the threshhold at end of the step (with unmodified stepsize), by the intermediate value theorem there must exist some modified stepsize such that the radius at the end of that step is within the required tolerances.

Here is one possible implementation:

```fortran
      ! returns either keep_going, retry, backup, or terminate.
      integer function extras_check_model(id)
         integer, intent(in) :: id
         integer :: ierr
         type (star_info), pointer :: s

         ! other variables

         real(dp) :: rtol, atol, error, value, target_value

         ierr = 0
         call star_ptr(id, s, ierr)
         if (ierr /= 0) return
         extras_check_model = keep_going


         ! if you want to check multiple conditions, it can be useful
         ! to set a different termination code depending on which
         ! condition was triggered.  MESA provides 9 customizeable
         ! termination codes, named t_xtra1 .. t_xtra9.  You can
         ! customize the messages that will be printed upon exit by
         ! setting the corresponding termination_code_str value.
         ! termination_code_str(t_xtra1) = 'my termination condition'
```

```
        if (s% x_ctrl(1) > 0 .and. s% photosphere_r > s% x_ctrl(1)) then
           rtol = 1d-6
           atol = 1d-6
           value = s% photosphere_r
           target_value = s% x_ctrl(1)
           error = abs(value - target_value)/ &
                   (atol + rtol*max(abs(value),abs(target_value)))
           if (error > 1) then
               extras_check_model = retry
           else
               extras_check_model = terminate
           end if
        end if

        ! by default, indicate where (in the code) MESA terminated
        if (extras_check_model == terminate) s% termination_code = t_extras_check_model
     end function extras_check_model
```

Note: the `controls` namelist has many similar options, such as minimum central abundances, etc. All of them are controlled by the options `when_to_stop_rtol` and `when_to_stop_atol`, which are used internally by the `star` module exactly as above. In principle, we could have let our implementation respect those variables as well. The default values for these are no different from a loose threshhold (i.e. `MESA` will make no attempt to "hit the target").

Note: `MESA` keeps an internal counter for the number of times it has to retry timesteps (for each timestep; the counter resets when a timestep is accepted), and will backtrack (i.e. redo the previous timestep with a smaller stepsize) once this counter exceeds a predetermined threshhold (`max_retries`, set in the `controls` namelist). If you want to get around this, or alternatively, if you want to change the timesteps differently from `MESA`'s built-in retry heuristics, you can manually modify the timestep size and tell `MESA` to redo, rather than retry, the timestep. In the above example, this would look something like the following:

```
           ! ...
           if (error > 1) then
               s% dt = s% dt * 0.75
               extras_check_model = redo
           else
           ! ...
```

Again, `MESA` has a user-configurable quantity called `timestep_dt_factor` which is used natively by `retry` for precisely this purpose, so in most applications you will have no reason to reinvent the wheel in this manner. Broadly speaking, the semantics are as follows:

- If you only want to repeat the timestep with a smaller stepsize, use `retry`
- Otherwise, if you want to re-attempt the timestep after making other changes (e.g. changing input physics), use `redo` for everything else

---

Exercise

Implement a custom hook in `extras_start_step` so that you only write out profiles for T_{eff} less than some threshhold set in `x_ctrl(1)`.

Exercise

Implement a custom stopping condition (in either `extras_check_model` or `extras_finish_step`) by comparing the HR-diagram discriminant G defined above against a user-supplied value in `x_ctrl(2)`.

4.3 Custom physics

In addition to the stubs exposed in `run_star_extras.f`, `MESA` also contains hooks by which one might override the built-in physics with essentially arbitrary complexity. The generic steps to follow are laid out in the README file in `$MESA_DIR/star/other`, which I recommend you also read. In summary:

0. Set up the `run_star_extras.f` file as described above.
1. Copy the template routine from the appropriate file in `$MESA_DIR/star/other` to `run_star_extras.f`, and modify appropriately.
2. Edit the subroutine `extras_controls` to set the appropriate pointer to your custom physics subroutine (e.g. `other_diffusion`)
3. In the `controls` namelist, enable your custom physics by setting, e.g., `use_other_diffusion = .true.`

Each of these hooks is fairly idiosyncratic, and it doesn't make much sense for me to go through each one in detail. In any case, the API for each of them is illustrated pretty well by the "no-op" reference implementation. Rather, to illustrate the general principles, I will demonstrate the implementation of something that is actually useful.

4.3.1 Worked Example: Soft Diffusion Turnoff

Element diffusion is known to produce unphysical results for high-mass stars with very thin or nonexistent convective envelopes. One mitigation strategy is to turn it off at higher masses, say above some predetermined mass M_0 above which we don't want diffusion to happen, but this naive strategy introduces discontinuities for mass ranges containing this cutoff point. To get around this, one possible alternative is as follows: we compute the diffusion coefficients as usual. However, we then apply a multiplier $C(M) \leq 1$ to the diffusion coefficient before it is used by the solver. This multiplier is constructed so that $C \to 1$ for $M \ll M_0$ and $C \to 0$ for $M \gg M_0$. For example, Viani et al. (2018) suggest

$$C(M) = \begin{cases} \exp\left[-\frac{(M/M_\odot - 1.25)^2}{2(0.085)^2}\right] & M/M_\odot > 1.25 \\ 1 & M/M_\odot \leq 1.25 \end{cases} \tag{1}$$

For stellar masses much higher than 1.25 $M_\odot$, the diffusion coefficient goes to zero, and we effectively have turned off element diffusion, but in a way that doesn't introduce modelling discontinuities.

Having set up my `run_star_extras` file as above, I proceed with the next few steps:

Step 1: Rummage around in `$MESA_DIR/star/other`.

Looking through the files here, I see that there is a file named `other_diffusion_factor.f90`. The "no-op" implementation sets an array called `extra_diffusion_factor` to be 1 everywhere in the star. Reading through the relevant files (which you can find with `grep -r "extra_diffusion_factor" $MESA_DIR/star`), it appears that this is treated as a multiplier applied to the diffusion coefficient, which is exactly what we require.

I define the subroutine `custom_diffusion_factor` as follows (based on the reference implementation):

```
      function diff_factor(M) result(C)
         implicit none
         real(dp), intent(in) :: M
         real(dp) :: C
         if (M < 1.25) then
            C = 1d0
         else
            C = exp( - (M - 1.25)**2 / 2d0 / (0.085)**2)
         end if
      end function

      subroutine custom_diffusion_factor(id, ierr)
         integer, intent(in) :: id
```

```
        integer, intent(out) :: ierr
        type (star_info), pointer :: s
        integer :: k, num_classes
        ierr = 0
        call star_ptr(id, s, ierr)
        if (ierr /= 0) return

        if(s% diffusion_use_full_net) then
           num_classes = s% species
        else
           num_classes = s% diffusion_num_classes
        end if

        ! Custom code comes here

        s% extra_diffusion_factor(1:num_classes, 1:s% nz) = diff_factor(s% star_mass)
     end subroutine custom_diffusion_factor
```

I place this within the module definition in `run_star_extras.f` in my work folder.

Step 2: Edit `extras_controls`

I modify the subroutine `extras_controls` to contain the following pointer definition:

```
s% other_diffusion_factor => custom_diffusion_factor
```

After all of these changes have been made, you will have to rebuild your work folder (`./clean && ./mk`).

Step 3: Enable custom physics

In the controls namelist, I set

```
do_element_diffusion = .true.
use_other_diffusion_factor = .true.
```

This also means that, if you want to do a quick comparison of the results with and without your custom physics, you can just set this flag to `.false.` rather than recompiling your work directory again. Convenient!

Exercise: Core Heating in a Hot Jupiter

a. Construct a Hot Jupiter model, starting from a sphere already in hydrostatic equilibrium (MESA doesn't do protoplanetary disks, unfortunately). Terminate the evolution at an age of 1 Gyr (roughly the current solar system age). You may use the test suite examples as a guide. I would personally set the following parameters:

- `star_job`:
  - `relax_initial_radiation = .true.` (i.e. the model is being irradiated by its star)
  - `relax_to_this_irrad_flux = 1.36d7` (this is ten times the solar constant — we are placing the HJ at about .3 AU from a Sun-like star)
  - `irrad_col_depth = 1` (the irradiation is assumed to affect the outer $4\pi R^2 \times$ `col_depth` grams of the atmosphere, in cgs units).
  - `kappa_lowT_prefix = 'lowT_Freedman11'` (low-temperature opacities for not-stars)
- `controls`:
  - `initial_mass = 0.001` (about 1 Jupiter mass)
  - Optional: `atm_option = 'irradiated_grey'` (planetary atmospheric boundary condition of Guillot & Havel, 2011, To use this, you will need to specify the surface temperature, opacity, and pressure).

b. Implement additional heating in the core, by overriding `s% other_energy`. To do this, first find the index of the grid point associated with the core:

```
integer :: k

! ...

k = s% nz
```

Next, introduce extra heat into the stellar model specifically into this grid point:

```
s% extra_heat(k) = my_extra_heating_term(s, ... )
```

Possible formulations of this include terms associated with MHD ohmic heating (Batygin & Stevenson, 2010), radioactive decay and thermal inertia (Chen & Rogers, 2016), or tidal interactions (Millholland, 2019). If you don't care and just want to do this exercise, you may also just introduce a constant extra heating, Whichever formulation you use, multiply it by a value specified in `x_ctrl(1)`.

c. Now, obviously it's not physical to introduce heating into a single mesh point; if you're not careful, this can cause Bad Things to occur (for instance, it may cause your model to go out of hydrostatic equilibrium, or at least give the hydro solver a hard time converging to a consistent state). Rather than depositing this extra heat in the central mesh point, introduce smoothing in terms of a half-Gaussian kernel (becomes a Gaussian thanks to spherical symmetry) of fixed fractional radius, which you may optionally define with `x_ctrl(2)`.

d. This extra heating produces "radius inflation". The reason why there are papers investigating radius inflation at all is because the radii of photometric gas giants, inferred from transit depths, are uniformly larger than predicted from naive modelling. Compare the increase in radius that you obtain from your extra heating to when no extra heating is induced, by plotting families of curves of R against t for different values of your overall multiplier `x_ctrl(1)` and smoothing width `x_ctrl(2)`. Remember that you will need to set `use_other_energy = .true.` in the `controls` namelist to see an effect.

4.4 Further reading

Again, I recommend Josiah Schwab's MESA Summer School lectures (see above). Note that the `MESA` API has changed significantly in a non-backwards-compatible way recently; quite a bit of the older material (particularly pertaining to `run_star_extras.f`) is no longer applicable.

5 Normal modes of oscillation: using GYRE

Now that we know how to evolve stars, let us take a moment to think about what we can do with snapshots in time. Since MESA is a one-dimensional stellar evolution code, it can only deal with things that emerge from 1. spherical symmetry and 2. hydrostatic equilibrium. However, many stellar phenomena have to do with deviations from both of these simplifying assumption. “Small” perturbations from such equilbrium structure can all be expressed as linear combinations of normal modes, which oscillate freely at their associated natural frequencies. These normal modes and natural frequencies emerge as the eigenfunctions and eigenvalues of a boundary value problem constructed with respect to a given stellar structure. While MESA can be used to derive this structure in the first place (in a hopefully physically consistent manner), it cannot by itself be used to solve the associated boundary value problem.[4] For the purposes of this course, we will use the GYRE code for two reasons: it has been extensively benchmarked for accuracy against older codes (like `adipls`), and it is also *much* easier to use, since it is configured via namelist files instead of custom input formats.

5.1 Installing GYRE

MESA already comes with a version of GYRE, which can be used interoperably for pulsation calculations within MESA itself (as seen in the `astero` module). However, this is compiled as a library. You will otherwise need to head into `$MESA_DIR/gyre/gyre` and call `make`, which will compile the standalone form of the `gyre` executable. I recommend that you also set the environment variable `GYRE_DIR=$MESA_DIR/gyre/gyre`. Finally, I should note that the version of GYRE that comes included with MESA is fairly old (as of MESA r12778). If you want to use a newer version of GYRE, compile that elsewhere (using the `mesasdk`) and change your `GYRE_DIR` environment variable to point there instead — generally speaking, the `astero` module (next section) isn’t well-tested with newer versions of GYRE, and may suffer incompatibilities.

5.2 Using GYRE

In order to get your stellar structures into a format that GYRE likes, I recommend you set the following options in your MESA `controls` namelist:

```
    write_pulse_data_with_profile = .true.
    pulse_data_format = 'GYRE'
```

GYRE is a general-purpose stellar pulsation solver. Actually, it is a general-purpose boundary-value-problem solver, but the customisation hooks are much less easy to get at than those of MESA, so as far as this course is concerned, it’s basically only useful for stellar pulsations.

The solution strategy for GYRE has two main steps:

1. There exists a discriminant function $D(\omega)$ associated with the boundary value problem, such that the zeroes of this function are eigenvalues of the boundary value problem. This is a function of the frequencies, and is evaluated numerically. For example, for the linear BVP under consideration, which is of the form $\frac{\mathrm{d}}{\mathrm{d}x}\mathbf{y} = \mathbf{A}(\omega)\mathbf{y}$, where $\mathbf{A}$ is a matrix-valued function of ω, the derivative operator can be discretised into a finite-difference matrix $\mathbf{D}$, yielding a (not necessarily linear) matrix eigenvalue problem of the form $\hat{\mathbf{D}}\hat{\mathbf{y}} = \hat{\mathbf{A}}(\omega)\hat{\mathbf{y}}$. Eigenvalues are roots of the discriminant function $D(\omega) = \det\left(\hat{\mathbf{D}} - \hat{\mathbf{A}}(\omega)\right)$, which approximates that of the continuous problem. Therefore, the first step GYRE takes is simply finding where all the zeros of this function are (within a specified interval). If your objective is only to find the oscillation frequencies, you can stop here.
2. Once the eigenvalues are found, they are inserted into the boundary value problem, which is also solved numerically, through e.g. matrix relaxation methods for finding eigenvectors. GYRE implements several solution schemes, accurate to different orders.

Just like with MESA, the input parameters to GYRE are specified via Fortran namelist files. The namelists you need to provide to GYRE are as follows:

[4]New versions of MESA actually allow you to simulate radial (i.e. spherically symmetric) stellar pulsations directly, effectively by directly simulating the hydrostatic response of the star under the action of small perturbations with very small timesteps. This still doesn’t tell you anything about the pulsation eigensystem, however.

- `model`: Location and input format of the stellar structure model.
- `constants`: Physical constants (e.g. this is where you override the gravitational constant if you want to test cosmology)
- `mode`: Properties of the modes you're solving for (i.e. quantum numbers l and m)
- `osc`: Properties of the eigenvalue problem (whose description and dynamical variables you want `GYRE` to use)
- `rot`: How you want `GYRE` to deal with stellar rotation. The first-order effect of rotation is to yield symmetric perturbations linear in the azimuthal quantum number m, but `GYRE` is also capable of performing computations for not-so-slow rotators incorporating some nonlinear effects. **(note: not present in v5.2; options were formerly in `osc`)**
- `num`: Numerical methods (e.g. difference scheme) for the boundary value problem
- `scan`: Numerical methods (e.g. grid spacing, units) for finding the roots of the discriminant function
- `grid`: Sometimes `MESA` does not provide a fine enough coordinate mesh to resolve the oscillation eigenfunctions. `GYRE` will perform an adaptive resampling, and this is where you specify how this is to be done.
- `ad_output`: output format for adiabatic oscillations.
- `nad_output`: output format for nonadiabatic oscillations. This namelist must still be present (it may be empty) even if only adiabatic calculations are being done.

That's a lot of stuff to specify! As with using `MESA`, I recommend that you keep a "template" version of your input file handy, and just swap out the values of the appropriate variables as required. Like `MESA`, `GYRE` tries to have sane defaults, so hopefully you won't need to intervene too much.

Once you've assembled your input namelist file, you will then invoke `GYRE` as follows:

```
$ $GYRE_DIR/bin/gyre ./gyre.in
```

Sit back and watch the fireworks!

5.3 Worked example

Here is a `GYRE` input file that I have used for some red giant work I've been doing lately:

```
&model
    model_type = 'EVOL'
    file = '../$bname'
    file_format = 'MESA'
    ! file_format = 'FGONG'
    ! data_format = '(1P5E16.9,x)'
/

&constants
    G_GRAVITY = 6.67408d-8
/

&mode
    l = 0
    tag = 'l0'
/

&mode
    l = 1
    tag = 'l1'
/

&mode
    l = 2
    tag = 'l2'
```

```
/

&osc
    outer_bound = 'JCD'
    variables_set = 'JCD'
    inertia_norm = 'BOTH'
    !reduce_order = .FALSE.
/

&rot
/

&num
    diff_scheme = 'COLLOC_GL4'
/

&scan
    grid_type = 'LINEAR'
    freq_min_units = 'UHZ'
    freq_max_units = 'UHZ'
    freq_min = $lower
    freq_max = $upper
    n_freq = 100
/

&grid
    alpha_osc = 10            ! At least alpha points per oscillatory wavelength
    alpha_exp = 4            ! At least alpha points per exponential 'wavelength'
    n_inner = 0                ! At least n points in the evanescent region
    tag_list = 'l0'
/

&grid
    alpha_osc = 10
    alpha_exp = 10
    n_inner = 20
    tag_list = 'l1,l2'
/

&ad_output
    summary_file = '$fname.dat'
    summary_file_format = 'TXT'
    summary_item_list = 'l,n_pg,n_p,n_g,freq,E_norm'
    freq_units = 'UHZ'
/

&nad_output
    summary_file = '$fname-nad.dat'
    summary_file_format = 'TXT'
    summary_item_list = 'l,n_pg,n_p,n_g,freq,E_norm'
    freq_units = 'UHZ'
/
```

Note that you can specify multiple copies of each namelist, distinguished with "tags". For example, I have three `mode` namelists, one for each of $l = 0, 1, 2$; I also have two `grid` namelists, one for radial modes and one for nonradial modes. Each of the subsequent namelists are matched up with tagged `mode` namelists via the `tag_list` option, which specify a comma-separated list of tags. If you specify a tag that is not matched by any of the tag lists, the final namelist of that type will be used for calculations (so, for example, I could have omitted the `tag_list` specification in the second `grid` namelist, or only have specified it for either dipole or quadrupole modes, and it still would have yielded the same result). This can potentially allow for some quite versatile behaviour!

Exercises: asteroseismology

On Canvas, you will find a `GYRE` file called `Patched.GYRE`. With respect to this structure file:

1. Compute the radial, dipole, and quadrupole adiabatic mode frequencies using the COLLOC GL2 difference scheme.

Adiabatic oscillations are pretty well-behaved, so the choice of difference scheme shouldn't really matter, but in this case I've subtly doctored the atmosphere of the stellar model (to match it with 3D MHD simulations). This is a solar-like model, so search for frequencies within the range 2500 μHz to 4500 μHz.

2. Make an echelle diagram.

From your lectures on pulsation phenomenology, you will recall that the oscillation frequencies obey an approximate relation

$$\nu_{nl} \sim \Delta\nu \left(n + \frac{l}{2} + \epsilon_l(\nu)\right), \tag{2}$$

where $\Delta\nu \sim 1/2T_0$, with T_0 being the radial sound-travel time (i.e. $\Delta\nu$ is the inverse sound-crossing time). To a first approximation, ϵ doesn't vary much with frequency — equivalently, the structure of the star is more or less homogenous/polytropic. Departures from a constant value of ϵ are how we measure structural properties from acoustic oscillations.

One way of quickly diagnosing these frequency-dependent variations is via an **echelle diagram**. Using the relation above, perform a least-squares linear fit for ν against n to the radial modes you have computed (i.e. $\Delta\nu$ and ϵ are the free parameters that you are fitting for). With the value of $\Delta\nu$ that you have obtained in this manner, construct a plot of $\nu \mod \Delta\nu$ against ν. This is called an "echelle diagram", so called because different modes of identical degree are stacked up on top of each other ('echelle' means ladder in French).

Hint: You may find code in the appendices to be useful.

3. Construct a propagation diagram for the quadrupole modes.

In the WKB limit, adiabatic oscillations have a radial wavevector

$$k_r^2 = \frac{\omega^2}{c_s^2}\left(1 - \frac{N^2}{\omega^2}\right)\left(1 - \frac{S_l^2}{\omega^2}\right), \text{where} \tag{3}$$

- N is the Brunt-Väisälä[5] frequency, which is constructed from the local gravitational field strength and other thermodynamic quantities[6] as
$$N^2 = -\frac{g}{\rho}\left.\frac{\partial\rho}{\partial s}\right|_P \frac{\mathrm{d}s}{\mathrm{d}r} = \frac{g\delta}{c_P}\frac{\mathrm{d}s}{\mathrm{d}r}, \text{ and} \tag{4}$$
- S_l is the Lamb freqency, which is the acoustic equivalent to the angular-momentum term in the effective radial potential of a central-force problem:
$$S_l^2 = \frac{l(l+1)}{r^2} \cdot c_s^2. \tag{5}$$

[5]look upon my glöriöüs ümläüts and despair

[6]see http://hyad.es/bv

Waves propagate when $k_r^2 > 0$, and are evanescent when $k_r^2 < 0$. This means that eigenvalues can only exist for ω such that there exist some intervals of the radial coordinate where $(\omega^2 > S_l^2) \wedge (\omega^2 > N^2)$, and/or $(\omega^2 < S_l^2) \wedge (\omega^2 < N^2)$, which will be where the corresponding eigenfunctions are propagating instead of evanescent.

For dipole modes ($l = 1$), make a propagation diagram (i.e. a plot of both of the above quantities against either the physical radius or the acoustic radial coordinate — I recommend the latter) from the provided `GYRE` structure file.

Hint: You may find code in the appendices to be useful.

4. Turning points

For each of the dipole modes you computed in (1), draw a horizontal line segment on the propagation diagram such that the interior extent of the line terminates at where $\omega = S_l$, and the exterior terminates at where $\omega = N$. These are the inner and outer turning points (in the classical WKB sense) of the modes. In practice, the mode eigenfunctions decay exponentially in the evanescent regions.

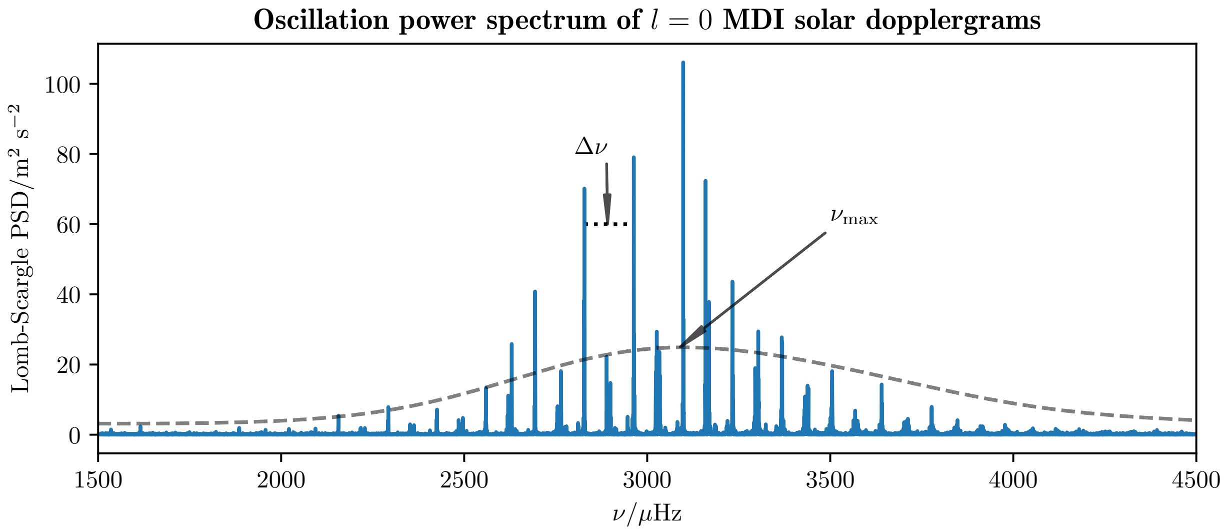


Figure 2: Solar oscillation power spectrum, showing the frequency of maximum power ν_{max}.

Now, solar-like oscillations possess a characteristic excitation frequency, sometimes called the frequency of maximum power, or ν_{max} (see Figure 2). ν_{max} satisfies a scaling relation that is given in the main lecture notes:

$$\nu_{\text{max}} \sim (M/M_\odot)^1 (R/R_\odot)^{-2} (T_{\text{eff}}/T_{\text{eff},\odot})^{-1/2} \cdot 3090\ \mu\text{Hz}. \tag{6}$$

Give the above lines opacities based on how close they are to ν_{max}.

5. More propagation diagrams

I provide on Canvas other structure files called `SG.GYRE` and `RG.GYRE`, corresponding to a subgiant and a first-ascent red giant star. For each of these, make a similar propagation diagram, with the horizontal axis scaled logarithmically. You should obtain something that looks like Fig. 1b of Ong & Basu (2020). Mark out the position of ν_{max} on each of these diagrams. What can you say about the propagation of waves in these stars? How does it differ from the solar-like model?

6. Avoided crossings

Compute frequencies of the dipole modes for the provided subgiant model, and show them on an echelle diagram (using the value of $\Delta\nu$ implied by the radial modes alone). How do they differ from those of the solar-like model?

Exercises: Planetary seismology

For this exercise, you may either use the file `HJ.GYRE` that I've placed on Canvas, or select a hot Jupiter model produced from the exercise you did in the last section. Note that atmospheres created with the `irradiated_grey` atmosphere cannot be appended to pulsation models.

1. Propagation diagram

For the provided model, construct a propagation diagram as above. How does it differ from that of the two stellar models?

2. Ionization zones

Plot Γ_1 against T for each of the above models, on the same axes. For the solar model, you should see that $\Gamma_1 = 5/3$ (the ideal-gas value) for most of the star, except for a narrow region. What is this region? (hint: look at the title of the question)

3. Superadiabatic regions

Plot the dimensionless entropy gradient

$$-\left(\frac{H_p}{c_P}\right)\frac{\mathrm{d}s}{\mathrm{d}r} = \nabla - \nabla_{\mathrm{ad}} + \frac{\phi}{\delta}\nabla_\mu = -\frac{N^2 H_p}{g \cdot \delta} \tag{7}$$

against x for each of the above models, on the same axes. For the stellar models, you should see a narrowly confined enhancement just under the photosphere. This is called the "superadiabatic layer" because the temperature gradient is much higher than the adiabatic lapse rate. What does the outside of the HJ model look like, and why?

5.4 Common issues

5.4.1 `MESA` parameters

I generally recommend that you set the following options in your `MESA` `controls` namelist:

```
      add_atmosphere_to_pulse_data = .true.
      add_center_point_to_pulse_data = .true.
      keep_surface_point_for_pulse_data = .true.
      add_double_points_to_pulse_data = .true.
```

These are quite self-explanatory. `GYRE` can handle all of these cases; they're mostly there to accommodate other pulsation codes (or if you want to use nonstandard boundary conditions with `GYRE`).

5.4.2 Input/output errors

On some systems (including the HPC, but also on some of the department machines), `GYRE` may **nondeterministically** fail to read or write files with i/o errors. Error messages may look like this:

```
HDF5-DIAG: Error detected in HDF5 (1.10.2) thread 0:
#000: ../../src/H5F.c line 445 in H5Fcreate(): unable to create file
major: File accessibilty
minor: Unable to open file
#001: ../../src/H5Fint.c line 1519 in H5F_open(): unable to lock the file
major: File accessibilty
minor: Unable to open file
#002: ../../src/H5FD.c line 1650 in H5FD_lock(): driver lock request failed
major: Virtual File Layer
minor: Can't update object
#003: ../../src/H5FDsec2.c line 941 in H5FD_sec2_lock(): unable to lock file,
errno = 524, error message = 'Unknown error 524'
```

```
major: File accessibilty
minor: Bad file ID accessed
```

This is a known issue with the file locking feature as implemented in versions of the HDF5 library newer than 1.10.x (which `GYRE` uses for i/o operations) interacting with Linux filesystem drivers, particularly for networked drives, which may not properly implement file locking — see https://support.nesi.org.nz/hc/en-gb/articles/360000902955-NetCDF-HDF5-file-locking. So far, Rich hasn't come up with a fix for this. I recommend working around it by setting the following environment variable:

```
export HDF5_USE_FILE_LOCKING=FALSE
```

5.5 Further reading

You might see references to a `GYRE` wiki hosted on BitBucket; it has recently become defunct, as BitBucket has finally discontinued support for Mercurial repositories. Rich has recently migrated `GYRE` to `git` and, while he was at it, GitHub; `GYRE` is now hosted at http://github.com/rhdtownsend/gyre, and the GYRE documentation can be found at https://gyre.readthedocs.io/en/latest/.

5.6 Planned changes

Rich is currently undertaking a major refactoring of `GYRE` in anticipation of a 6.0 release; many things are getting rearranged at the moment. The above input files are liable to break; in particular, Rich plans to introduce new namelists (e.g. tidal forcing) and rename several variables (e.g. `alpha` for the grid parameters will be renamed to another letter, to avoid confusion with different switches in the standard equations). Hopefully things will stabilise after the 6.0 release.

Of particular note is that many of the current qualitative flags will be replaced with quantitative ones. For example, previously the Cowling approximation was effected by setting `osc % cowling_approx = .true.`; in GYRE 6.0, it will be effected by setting `osc % alpha_gr = 0.`, where $\alpha_{\rm gr}$ is a tuning parameter in the oscillation equations that is 1 for the full equations and 0 in the Cowling approximation.

More generally, `GYRE` packs a general-purpose BVP solver under the hood, whose application to asteroseismology is done through a thin wrapper around the oscillation equations in particular. It's possible to use it for other problems, such as finding unstable modes of collapsing disks, or even just pedagogical demonstration like the oscillations of a string. If you plan to use `GYRE` for serious science, I strongly recommend you acquaint yourself with the GYRE documentation and/or Rich Townsend better than these notes can hope to do.

5.7 Alternatives

`MESA` also includes a different pulsation code, `adipls`, written by Jørgen Christensen-Dalsgaard (known at large as JCD). The configuration files are in fixed format and so are much less user-friendly, as are the output files. For these reasons, I won't go into the details of how to operate `adipls` in these notes. However, `adipls` is also significantly faster, as long as your problem is embarassingly parallel. By contrast, `GYRE` is a little slower, but is parallelised via OpenMP (like `MESA` itself). If you're using `adipls`, you'll want to tell `MESA` to write your pulse data in the `FGONG` format.

6 Optimisation: the astero module

So far, we have discussed only how to generate stellar models given decisions about their properties (physical and astronomical) made a priori. Now it is time to consider how we might go about matching these generative models to observational data.

The usual approach that is taken in these cases is to consider the stellar evolution code as a black box accepting arguments $\{x_i\}$, producing a bunch of observables $\{y_j(\mathbf{x})\}$. These correspond to observed values Y_j with associated uncertainties σ_j. Assuming the observational errors to be i.i.d. and Gaussian, we might construct a cost function

$$\chi^2 = \sum_j^J \left(\frac{Y_j - y_j(\mathbf{x})}{\sigma_j}\right)^2, \tag{8}$$

in the usual manner. In this case, the y_j can be both spectroscopic quantities, like the metallicity, effective temperature, or luminosity; the input parameters include the astrophysically interesting ones, such as the stellar mass and age, or other model-specific parameters, such as the mixing length/overshoot parameter/mass loss rate etc. The objective of this whole exercise is to derive values of these parameters that extremise this cost function. If you read Bayesian statistics textbooks, this is essentially the maximum-likelihood approach to parameter estimation, with the quantity χ^2 being the log-likelihood function (up to additive constant determined by the uncertainties). (Un)fortunately, this is not a statistics textbook, and the precise interpretation of these quantities lies beyond the scope of these notes.

Conveniently enough, MESA comes with a built-in optimisation package. It's called the astero module because its author, Warrick Ball, mostly works on asteroseismic data; however, it can be used for generically any kind of stellar modelling problem of the above form. In bare terms, the astero module implements simplex optimisation (i.e. the Nelder-Meade “amoeba” algorithm), as well as several other “derivative-free” methods, to find local minima of the cost function in parameter spaces of MESA input variables. Depending on your needs, this may be good enough for simple problems.

For science purpose, I recommend a different way of deriving final estimates, for several reasons — e.g. derivative-free optimisation cannot be used to estimate uncertainties robustly by itself; you will need to additionally calculate the local Jacobian/Hessian at the optimal point. Moreover, simplex optimisation (as with any kind of pointwise optimisation scheme) is prone to trapping in local optima, which makes it ill-suited for estimating parameters drawn from multimodal likelihood functions. This is problematic for stellar astrophysics in particular because many of the relevant observables exhibit discontinuous emergent behaviour over the course of stellar evolution (for example, the subgiant hook and the red giant bump in the HR diagram), and so yield strongly multimodal likelihood functions. It's possible to get the astero module to interface with external sampling schemes, but this might take a little work.

6.1 Setting up the optimisation problem

The astero module comes with its own driver program, which can be found in e.g. `$MESA_DIR/star/astero/work`. In it you will find an example set up by Warrick, with three auxiliary input namelist files referenced to `inlist` by `read_extra_star_job_inlist1`. In addition, there is an extra file that is *not* referenced by `inlist`, named `inlist_astero_search_controls`. Let's have a look at the contents of this file:

```
&astero_search_controls
   ! example based on HD49385, but modified for testing various new stuff

   ! the overall chi^2 is a combination of spectroscopic and seismic

      ! chi2 = chi2_seismo*chi2_seismo_fraction
      !      + chi2_spectro*(1 - chi2_seismo_fraction)
      chi2_seismo_fraction = 0.667d0

   ! chi2_spectro is a sum of terms for different non-seismic
   ! observables (e.g. Teff, logg, FeH)
   ! if normalize_chi2_spectro, the sum is divided by the number of
```

```
  ! terms
     normalize_chi2_spectro = .true.
     ! only need to give values for target and sigma if include in chi^2

     include_Teff_in_chi2_spectro = .true.
     Teff_target = 6095d0
     Teff_sigma = 65d0

     include_logg_in_chi2_spectro = .false.
     logg_target = 3.97d0
     logg_sigma = 0.02d0

     include_logL_in_chi2_spectro = .true.
     logL_target = 0.67d0
     logL_sigma = 0.05d0

     include_FeH_in_chi2_spectro = .true. ! [Fe/H]
     FeH_target = 0.09
     FeH_sigma = 0.05

     ! etc.
```

As you can see, we have a new namelist, named `astero_search_controls`, which is read by the `astero` module rather than the `star` module. Default values for these quantities can be found in the file `$MESA_DIR/star/astero/defaults/astero_search_controls.defaults` — note that `astero` basically lives in its little corner of the `star` module, rather than being a full `MESA` module in its own right.

Generically, there are two sets of things you need to worry about:

- Input parameters x_i (in the context of Equation 8), such as the mass, initial helium abundance, etc. For each of these, there is a family of namelist variables in the pattern e.g. `vary_mass`, `first_mass`, `min_mass`, `max_mass`, and `delta_mass` (if you are doing a grid scan).
- Output parameters/constraints y_i, such as the effective temperature, log luminosity, radius, etc. For each of them there is a family of namelist variables in the pattern `include_Teff_in_chi2_spectro`, `Teff_target`, `Teff_sigma` (for observational errors). The seismic constraints work a little differently, and I will discuss them in more detail later.

Note that if you set `vary_x` to `.false.`, the value you specify in `first_x` will be used for all computations. Also, you will need to set `search_type` to something other than `use_first_values` to get `astero` to do anything interesting. Otherwise, the configuration file is pretty self-documenting, so I won't go through everything in detail.

6.1.1 Running and recovering output

Once you've set up your optimisation, you will have to compile and run `MESA` in the usual manner (`./mk` and `./rn`). For each combination of input parameters `astero` will generate an evolutionary track, and find the minimal value of the cost function for all models in the track; this is treated as the cost function associated with the input parameters themselves. Formally, we have

$$\chi^2_\text{track}(\mathbf{x}) = \min_{t\in T} \chi^2_\text{model}(\mathbf{x}; t), \tag{9}$$

where t is the stellar age, and T is the set of stellar ages for all models along the evolutionary track associated with the parameters $\mathbf{x}$.

`astero` will then save the results to a history file associated with the optimisation scheme; e.g. by default if you use the simplex algorithm it will save a history file to `simplex_results.data`, which can be customised in `astero_search_controls`. Each line will contain the values of all the input parameters being varied, and the (global) output parameters included in the cost function, as well as the cost functions themselves, for the best-fitting model along each evolutionary track.

6.1.2 Specifying seismic constraints

The quantities T_{eff}, [Fe/H] etc. are typically measured by spectroscopy, and when including seismic constraints we make a distinction between these and seismological quantities, by splitting up the cost function into two parts as

$$\chi^2 = (1 - f_{\text{seis}})\chi^2_{\text{spectro}} + f_{\text{seis}}\chi^2_{\text{seismo}} \tag{10}$$

up to overall multiplicative constant. The tuning parameter f_{seis} is controlled by the parameter `chi2_seismo_fraction`. Likewise, recall that there are two main kinds of seismic constraints:

- Global properties, $\Delta\nu$ and ν_{max}
- Local properties, which are constructed out of the individual mode frequencies.

Again, each of these terms has an associated `chi2_seismo_x_fraction` parameter, to control their relative contribution to the overall cost term. The global properties each have a family of parameters similar to the spectroscopic ones above, while the individual mode frequencies are specified through a list of variables, like so:

```
nl0 = 9
l0_obs(1) = 799.70d0
l0_obs_sigma(1) = 0.27d0
l0_obs(2) = 855.30d0
l0_obs_sigma(2) = 0.73d0
l0_obs(3) = 909.92d0
l0_obs_sigma(3) = 0.26d0
l0_obs(4) = 965.16d0
l0_obs_sigma(4) = 0.36d0
l0_obs(5) = 1021.81d0
l0_obs_sigma(5) = 0.28d0
l0_obs(6) = 1078.97d0
l0_obs_sigma(6) = 0.33d0
l0_obs(7) = 1135.32d0
l0_obs_sigma(7) = 0.34d0
l0_obs(8) = 1192.12d0
l0_obs_sigma(8) = 0.45d0
l0_obs(9) = 1250.12d0
l0_obs_sigma(9) = 0.89d0
l0_n_obs(:) = -1
```

Note that you may supply radial order identifications with `l0_n_obs`, although in the above example they will be ignored. A similar set of variables is provided up to $l = 3$.

Now, the global properties can be (at least approximately) computed directly from the stellar model without necessitating the use of any pulsation code. Conversely, actually running the pulsation code is much more computationally expensive than e.g. evaluating an integral over the stellar structure. Moreover, the frequencies of oscillation evolve very rapidly; the size of typical observational uncertainties potentially correspond to relative changes over the course of 10^{-4} Gyr, which is ridiculously small. Therefore, the `astero` module contains controls that will control when, exactly, `MESA` decides to compute mode frequencies. `astero` considers a stellar model to be in one of three different possible states:

- **Cold**: no mode frequencies are computed.
- **Warm**: Only radial mode freqencies are computed. In order for this to happen, the current stellar model must pass several checks, including:
  - `min_age_limit` and `Lnuc_div_L_limit`: exclude pre-MS models
  - `chi2_spectroscopic_limit`: exclude models that don't match spectroscopy
  - `chi2_delta_nu_limit`: exclude models with different mean densities. **IMPORTANT**: This may still be used even if `chi2_seismo_delta_nu_fraction = 0`.
- **Hot**: A value of χ^2_{seismo} is computed using only radial mode frequencies for all models satisfying the "Warm" criteria. If this is lower than `chi2_radial_limit`, then the model is considered "Hot", and nonradial mode

frequencies are then also computed. Only values of χ^2_{seismo} for "Hot" models, which include nonradial modes, are considered in the computation of χ^2_{track}.

Since it doesn't make sense to adopt a max timestep size of 10^{-4} Gyr for the entire evolutionary track, this kind of subdivision allows you to specify different timestep sizes for the "Cold", "Warm", and "Hot" phases.

6.2 Common issues: logistics

6.2.1 `run_star_extras`

As with `star`, `astero` allows you to override certain parts of the optimisation (rather than evolution) with user-specified behaviour. The most common use case for this are to define your own input and output parameters, although the assumptions behind the configuration interface might be somewhat unwieldy in the general case. For instance, you might want to vary the rotation rate of the star, and constrain it against the oblateness of the star, perhaps determined from interferometry.

Presently, `astero` only supports varying the parameters $(M, Y, [\text{Fe/H}]_0, \alpha_{\text{MLT}}, f_{\text{ov}})$ in the optimisation procedure. If you decide that other parameters need to be optimised, the canonical way of doing this is via the namelist options `my_param1` (up to 3), with the usual family of arguments (`vary_my_param1` etc) as well as an additional argument `my_param1_name` for semantic naming in the sample history file.

Note that just telling `astero` to vary the parameter won't do anything; you will need to change the stellar properties somehow corresponding to the supplied value of these custom parameters. Like with `star`, this is done in `run_star_extras`. **Unlike with `star`**, `astero` uses a nonstandard set of routines in `run_star_extras`. In particular, extra parameters are to be handled in the subroutine `set_my_param`, which is called at the start of the stellar evolution (i.e. equivalent to `extras_startup` with the normal `run_star_extras`). Warrick recommends manipulating `s% job` rather than `s` directly (i.e. setting `star_job` rather than `controls` parameters).

Likewise, `astero` also allows you to include custom observables `my_var1` (up to 3; again, with the family of associated arguments) in the cost function. These are to be set in `extras_check_model`, since these should be updated before the cost function is evaluated at the end of the timestep.

6.2.2 Other optimisation schemes

It's actually possible to use an external optimisation procedure with `astero`, without mucking around too much with `run_star_extras`, but it will require some work. Rather than use any of the provided optimisation schemes, `astero` allows you to define a sampling strategy based on inputs from a whitespace-delimited file — you will set `search_type = 'from_file'`, and then set the argument `filename_for_parameters` to your desired value. Schematically, you may then write a black-box function that simply calls `astero` with the supplied parameters, gets it to compute the cost function for you, and then reads it back into whatever language you are using. Here's a prototype of how you would do it in Python:

```python
def cost(*x):
    np.savetxt("filename_for_params", np.array([x]), header="dummy")
    system("./rn")
    return parse_output(outname)

best_fit = do_sophisticated_optimisation(cost, x0, *args, **kwargs)
```

You might want to save the cost function and output parameters of each sample somewhere convenient, since they're reset each time you run `astero`.

6.3 Common issues: asteroseismology

6.3.1 Frequency splitting identifications

Note that `astero` will not let you supply mode identifications for the quantum numbers m for your observed modes. If you compute frequencies for rotating models, and need to match modes with respect to the rotational splitting, you will need to do this separately. It's unlikely that you'll run into this issue within the confines of this course; conversely, if you're in a situation where the quantum number m matters, you're probably better off writing your own modelling code anyway.

6.3.2 Surface Corrections

`astero` implements the two-term correction of Ball & Gizon (2014), which is known to be the most robust of all currently proposed parametric corrections (it also implements a few more, but they don't work as well on real stars). However, it has poorer support for nonparametric corrections. In particular, its implementation of frequency separation ratios is not applicable for evolved stars, and also suffers from strong correlated errors by combining r_{01} and r_{10}. Warrick recommends you implement new surface corrections by defining your own `other_astero_freq_corr` subroutine in `run_star_extras`.

Exercise: Solar Calibration

Using the Grevesse & Sauval (1998) abundance mixture and opacities, with element diffusion enabled (as in the exercises in Section 3.2), construct a $1M_\odot$ stellar model such that at a post-ZAMS age of 4.569 Gyr, it has a radius of $1R_\odot$, a luminosity of $1L_\odot$, and a surface metal abundance ratio of $Z/X = 0.023$, per GS98. You should be able to do this by varying only the initial helium abundance Y_0, the initial metal abundance Z_0, and the mixing-length parameter α_MLT. This is an exactly constrained problem (3 constraints for 3 inputs), and in principle you should be able to hit $\chi^2 = 0$. Most of these values are already in the default `astero_search_controls` namelist file. What is the optimal value of Z_0/X_0 that you obtain from this procedure, and why is it different from the GS98 value?

Hint: You may want to set `eval_chi2_at_target_age_only = .true.` for this exercise.

Exercise: Spectroscopic modelling of ι Draconis

With no element diffusion and overshoot, find the best-fitting model parameters (including age) that match the following global constraints:

- $\Delta\nu = 4.02 \pm 0.02\ \mu\text{Hz}$
- $\nu_\text{max} = 38.4 \pm 0.5\ \mu\text{Hz}$
- $T_\text{eff} = 4504 \pm 62$ K
- $[\text{Fe/H}] = 0.03 \pm 0.08$ dex
- $L = 52.78 \pm 2.10 L_\odot$

Warning (logistics): This may take you quite a long time, since the star is quite evolved.

Warning (science): The actual posterior distributions for this are highly multimodal (why?). Local estimators of the certainties of e.g. the ages and masses are therefore liable to be significant underestimates.

Exercise: Radius Inflation

Consider the irradiated Hot Jupiter scenario that we implemented additional heating for in the exercise of Section 4.3.

1. Fixing the irradiation received at the atmosphere to the Jovian value (i.e. flux received at Jupiter's orbit from the Sun's luminosity), using the chemical mixture of Grevesse & Sauval (1998), the solar-calibrated mixing-length parameter and initial helium abundance that you found earlier, and element diffusion, find the irradiation column depth required to produce a 1 M_J model of 1 R_J at the present solar-system age of 4.569 Gyr. Note that since most planetary models don't have superadiabatic stratification at the outer boundary of the convection zone (see exercises in Section 5.3), the planetary radius at a given age is quite insensitive to the mixing-length parameter, unlike the stellar case.

2. Using this Jupiter-calibrated irradiation column depth, hold all other parameters constant and set the irradiation flux to 8×10^9 erg s^{-1} cm^{-2}. Find the amount of extra heating (from whatever source you chose, parameterised by `x_ctrl(1)`) required to recover a radius of 1.7 R_J. This is the most extreme data point of Demory & Seager (2011). Repeat this for several masses between 0.7 and 3 M_J, and make a plot of `x_ctrl(1)` against the planetary mass.

A Some useful snippets

All of these snippets can be found (in more convenient machine-readable form) either on Canvas or on my personal research repository. I include these here for completeness on the off-chance that you retain this PDF for reference but somehow lose access to the supplementary material.

A.1 Python: `MESA` to pandas DataFrames

I use these often enough that I've collected them into my personal research script repository at http://gitlab.com/darthoctopus/mesa-tricks/ (specifically the `io` submodule), but I also provide them here in the off-chance that you'll find them useful.

```python
# excerpted from mesa_tricks/io/__init__.py

import numpy as np
import pandas as pd

def read_track(x, **kwargs):
    '''
    Read track file produced by MESA. If a directory is supplied,
    combine that with the profiles.index file in the directory.

    Input: folder containing file named history.data (and optionally
    profiles.index, integrals.csv)
    '''
    if not isdir(x):
        return pd.read_csv(x, **{'delim_whitespace': True, 'skiprows': 5, **kwargs})
    else:
        df1 = read_track(f'{x}/history.data')
        df2 = read_index(f'{x}/')

        df = pd.merge(df1, df2, on='model_number')
        if isfile(f"{x}/integrals.csv"):
            dfi = pd.read_csv(f"{x}/integrals.csv")
            df = pd.merge(df, dfi, on='model_number')
        return df

def read_profile(x, **kwargs):
    '''
    Read profile file produced by MESA.

    Input: filename (e.g. "profile\\d+.data")
    '''
    return dict(pd.read_csv(x, **{'delim_whitespace': True,
            'skiprows': 1, 'nrows': 1, **kwargs}).iloc[0]), read_track(x)

def read_index(track):
    '''
    Read index file produced by MESA

    Input: folder containing file named profiles.index
    '''
    return pd.read_csv(track+'/profiles.index', delim_whitespace=True,
                        skiprows=1, names=['model_number', 'priority', 'profile'])
```

A.2 bash: modifying configuration parameters

These functions were originally written by Earl Bellinger (formerly Yale/MPS grad student, now postdoc at Aarhus/Sydney), and I've found them immensely useful.

```bash
change_param() {
    # Modifies a parameter in the current inlist.
    # args: ($1) name of parameter
    #       ($2) new value
    #       ($3) filename of inlist where change should occur
    # Additionally changes the 'inlist_0all' inlist.
    # example command: change_param initial_mass 1.3
    # example command: change_param log_directory 'LOGS_MS'
    # example command: change_param do_element_diffusion .true.
    param=$1
    newval=$2
    filename=$3
    search="^\s*\!*\s*$param\s*=.+$"
    replace="      $param = $newval"
    sed -r -i.bak -e "s/$search/$replace/g" $filename

    if [ ! "$filename" == 'inlist_0all' ]; then
        change_param $1 $2 "inlist_0all"
    fi
}

set_inlist() {
    # Changes to a different inlist by modifying where "inlist" file points
    # args: ($1) filename of new inlist
    # example command: change_inlists inlist_2ms
    newinlist=$1
    echo "Changing to $newinlist"
    change_param "extra_star_job_inlist2_name" "'$newinlist'" "inlist"
    change_param "extra_controls_inlist2_name" "'$newinlist'" "inlist"
}
```

A.3 Python: `GYRE` to pandas DataFrames

The format of the structure files that `MESA` spits out for consumption with `GYRE` is fixed and basically not configurable. The full list of columns is specified in `$GYRE_DIR/doc/mesa-format.pdf`. It contains everything you need to compute the frequencies of oscillation for adiabatic p- and g-modes, as well as nonadiabatic oscillations, including those excited by the κ and ε mechanisms that you're more likely to get in classical pulsators. The one real thing it's missing is information about the chemical composition (which *is* present in some other formats, like `FGONG`).

```python
# excerpted from mesa_tricks/io/__init__.py

import numpy as np
import pandas as pd
GYRE_GLOB_NAMES = [
        "nrows", "M", "R", "L", "version"
    ]
GYRE_NAMES = [
        "k", "r", "m", "L", "P", "T", "ρ", "nabla", "N2", "Γ1", "nabla_ad", "δ",
        "κ", "κκ_T", "κκ_ρ", "εnuc", "εε_T", "εε_ρ", "Ω"
        # Double letters here indicate boldface
    ]
def read_gyre(f, **kwargs):
    '''
    Read GYRE model file produced by MESA

    Input: filename (e.g. "profile\\d+.data.GYRE")
    '''
    # GYRE accepts CGS units for its MESA input format; see documentation
    # Mesh points go from centre outwards (opposite from usual MESA profile files)
    info = pd.read_csv(f, delim_whitespace=True, nrows=1, names=GYRE_GLOB_NAMES, **kwargs).iloc[0].to_dict()
    profiles = pd.read_csv(f, delim_whitespace=True, skiprows=1, names=GYRE_NAMES, **kwargs)
    return info, profiles

def read_freqs(f, **kwargs):
    '''
    Read frequencies returned from GYRE: returns dict.
    '''
    df = pd.read_csv(f, skiprows=5, delim_whitespace=True, **kwargs)
    out = {}
    out['v'] = df['Re(freq)'].values
    out['l'] = df['l'].values
    out['n_p'] = df['n_p'].values
    out['n_g'] = df['n_g'].values
    out['E'] = df['E_norm'].values
    return out
```

References

Ball, W. H., & Gizon, L. 2014, A&A, 568, A123, doi: 10.1051/0004-6361/201424325

Batygin, K., & Stevenson, D. J. 2010, ApJL, 714, L238, doi: 10.1088/2041-8205/714/2/L238

Chen, H., & Rogers, L. A. 2016, ApJ, 831, 180, doi: 10.3847/0004-637X/831/2/180

Demory, B.-O., & Seager, S. 2011, ApJS, 197, 12, doi: 10.1088/0067-0049/197/1/12

Grevesse, N., & Sauval, A. J. 1998, SSRv, 85, 161, doi: 10.1023/A:1005161325181

Guillot, T., & Havel, M. 2011, A&A, 527, A20, doi: 10.1051/0004-6361/201015051

Millholland, S. 2019, ApJ, 886, 72, doi: 10.3847/1538-4357/ab4c3f

Ong, J. M. J., & Basu, S. 2020, ApJ, 898, 127, doi: 10.3847/1538-4357/ab9ffb

Viani, L. S., Basu, S., Ong, J. M. J., Bonaca, A., & Chaplin, W. J. 2018, ApJ, 858, 28, doi: 10.3847/1538-4357/aab7eb