MESA Summer School 2014 Documentation

Release 1.0

Kevin Moore

September 02, 2014

Contents

Contents:

CHAPTER 1

Lab 1 - Custom stopping/output criteria

1.1 Intro

In the MESA Tutorial, we learned how to add various stopping criteria to MESA. Typically these are of the form "stop when a certain varibale passes a certain threshold". Often this is enough, but sometimes you want more complicated stopping criteria - or maybe you'll want to save a bunch of profiles/models at 100 different times without chaining together 100 different inlists.

1.2 Setup

Make a local copy of the work directory, you can name it whatever you want. Go to the location you want to place the copied work directory and type

```
cp -r $MESA_DIR/star/work work_kevin_part1
```

where `work_kevin_part1` can be replaced with any name you want to use. You'll then want to check that everything worked fine by compiling and running things with the normal commands

```
./mk
./rn
```

Any time we change the code of the files in the local src directory, we must recompile the code!

1.3 Exercise 1 - Profile output criteria in inlists

Recall that MESA provides two output file types by default as it runs:

- History files (eg. `work/LOGS/history.data`) give the global properties of the star as a function of time. This is the file you'd look in if you wanted to make an HR diagram or a luminosity vs. time plot.
- Profiles (eg. `work/LOGS/profile37.data`) give a spatial slice of the star at a fixed time. This is the type of file you'd look in if you wanted to make a plot of temperature vs. pressure or abundances vs. mass coordinate.

If you want to save a profile through inlist controls (see eg. the `&controls` documentation in `mesa/star/defaults/controls.defaults`), you can save profiles at a given timestep interval, through, eg.

```
profile_interval = 50
```

and/or save one on termination with (in the `&star_job` section)

```
write_profile_when_terminate = .true.
filename_for_profile_when_terminate = 'my_profile_name.data'
```

If you have some special criteria you want to save profiles on, then you'd need to use a separate inlist for each criterion and save a profile on termination. While this works, it's kind of a pain if you want to run a grid of models or change some parameters of your run since you need to modify all the inlists. If you want to have these same output criteria for a grid of stars, each with slightly different input parameters, then you can put the stopping criteria in `run_star_extras.f`.

MESA always reads its parameters from a file named `inlist`. In the default `work` directory we copied, this inlist file just reads from another file named `inlist_project`. This allows you to have several different inlists in your directory, only needing to change the file that `inlist` itself reads from through the lines

```
extra_star_job_inlist1_name = 'inlist_project'
extra_controls_inlist1_name = 'inlist_project'
extra_pgstar_inlist1_name = 'inlist_pgstar'
```

We're going to run a 20 $M_\odot$ star through the main sequence here, and want to save profiles and models at three different abundance criteria: $X_{\rm center} = 0.69$, $X_{\rm center} = 0.40$, and $X_{\rm center} = 0.01$. You can just take the `inlist_project` file and copy it into three new inlist files that we'll slightly modify. What all needs to be changed?

- stellar mass - now 20 $M_\odot$
- profile output criteria - see above
- stopping criteria - see below

The stopping criteria in the inlists go in the `&controls` section and are as follows. You should name your inlists something descriptive to help you keep track of them - the names and stopping criteria I used are as follows:

First inlist (`inlist_to_h_ignition`):

```
xa_central_lower_limit_species(1) = 'h1'
xa_central_lower_limit(1) = 0.69
```

Second inlist (`inlist_to_h_burn`):

```
xa_central_lower_limit_species(1) = 'h1'
xa_central_lower_limit(1) = 0.40
```

Third inlist (`inlist_to_h_depletion`):

```
xa_central_lower_limit_species(1) = 'h1'
xa_central_lower_limit(1) = 0.01
```

To save time, you should save models at the end of the first and second inlists and load them into the sedond and third inlist, repsectively. To save a model, you can add to the `&star_job` section of your inlist (for example)

```
save_model_when_terminate = .true.
save_model_filename = 'h_ignition.model'
```

and to load a model, you can add (also to the `&star_job` section)

```
load_saved_model = .true.
saved_model_name = 'h_ignition.model'
```

Here, your only task is to verify that these inlists work as intended (i.e. you get three profiles out with the different names you gave them). Enjoy watching the PGSTAR plots!

Note: There are example shell scripts on how to automate chaining multiple inlists together so that you only need to execute one command to run the whole batch. See, for example, mesa/star/test_suite/make_planets. You still have the inconvenience of having to edit multiple inlists to change parameters.

1.4 Exercise 2 - Data output criteria in `run_star_extras.f`

1.4.1 Profiles

Your next task is to express the profile output criteria in the inlists from the previous section in `run_star_extras.f`. This way you only need to modify one inlist which makes things much more convenient when changing things such as resolution.

Recall from the `run_star_extras.f` tutorial (see here) that you need to replace the include statement in `src/run_star_extras.f`

```
include 'standard_run_star_extras.inc'
```

with a copy/paste of the contents of that file (located in `mesa/star/job`). Check that this works by compiling and running again. There's no need to run to completion - you should just verify that the code still compiles/runs before making further changes. *This is good advice for starting any modification of MESA. As with voting, you should recompile early and often - it will help prevent you having to look through a bunch of changes that all of a sudden aren't compiling!*

Look through the documentation of the provided procedures you copy/pasted into `run_star_extras.f` and find which one you should put output/stopping criteria in (when in the time step do you want to check these criteria?).

The recommended option is (highlight below to reveal):

You can express your output criteria using the `star_info` data structure which contains all the info MESA knows about your star. Look in `mesa/star/public/star_def.f` for the definition of this data structure. Lots of the variables are included from other files so they're not all listed explicitly in `star_def.f` (in particular, most of the variables are actually defined in `mesa/star/public/star_data.inc`). You need to find the names of the composition variables (there are many options that will work) in order to write output criteria. Throughout MESA's code, you'll see this stucture refered to as `s`, so if you see something like `s% mass(k)` then that just means look inside the structure `s` for the array `mass` and give me the kth entry.

If you're having trouble finding the right variable to use, try (highlight below to reveal)

Once you can express your criteria in if/then statements, you need the subroutines for outputting models/profiles. These are listed in `mesa/star/public/star_lib.f`, so search there for the right ones. You can call all of these subroutines from `run_star_extras.f` because they are already included using the line at the top

```
use star_lib
```

If you can't find a suitable subroutine there, try looking at (highlight below to reveal)

Note that while the call signatures of these subroutines require you to pass several things to them (including other subroutines!), most of these have the same names in `run_star_extras.f` so you shouldn't need to track down additional arguments. The main difference between using this method versus setting `s% need_to_save_profiles_now = .true.` is that calling the subroutine allows us to specify the profile's name, while `s% need_to_save_profiles_now = .true.` just tells MESA to output a profile using its standard naming conventions (so you'll get a profile named `profileXX.data` with whatever profile number you're on).

Finally, you need to slightly modify the inlists from before. I'd suggest making a new one (eg. `inlist_full`) that will run the star through the entire main sequence and output the three profiles at the right points. What's the inlist stopping criteria now? Do you need any profile output statements?

Check that you again produce the same results as you did with the multiple inlists.

1.4.2 Models

You can also save model files from `run_star_extras.f` for use as restart points using a similar subroutine. Take a look through `mesa/star/public/star_lib.f` to find it.

1.4.3 Stopping criteria in `run_star_extras.f`

Specifying a stopping criterion is the same as what we did in the previous part, except instead of calling subroutines to output profiles or model files, we send MESA/star a command that tells it to stop evolution. Notice at the beginning of the function `extras_finish_step()`, it is set to return the value

```
extras_finish_step = keep_going
```

which tells MESA/star to keep calm and carry on evolving the star. If you want to terminate the evolution for some reason, then you can instead set the function to return the value

```
extras_finish_step = terminate
```

among other possible integer stop codes listed in the `extras_check_model()` function located in `run_star_extras.f`. You can also return termination codes from that function. Documentation is provided on how to output custom messages for different stopping criteria in that function as well.

1.5 Brief overview of PGSTAR

Frank's lab will cover making custom PGSTAR plots in more detail, but you'll want some graphical output to stare at while your stars are evolving. There are two things you need to put in your inlist to make sure you have graphical output. In the `&star_job` section, you need to enable PGSTAR with the line

```
pgstar_flag = .true.
```

and in the `&pgstar` section, you need to specify what plots you want to see. The grid windows (several plots at once) are the best for starting out since they give you a dense set of info. Try turning on one of these windows with

```
Grid1_win_flag = .true.
```

You should also tell PGSTAR not to close the plots as soon as the run is over so you can still see what your star looked like at the end. You can do this with the line (also in `&pgstar`)

```
pause = .true.
```

MESA reads the `&pgstar` section of the inlist at each time step, so you can add and modify plots virtually in real time. Try it!

CHAPTER 2

Lab 2 - Adding new variables & using `star_info` structure

2.1 Intro

In this section, we'll practice extracting data from the `star_info` structure and use it to calculate some auxillary variables. In this activity, you should start by making a fresh copy of the `mesa/star/work` directory in your local work space. Name it whatever you want, for example

```
cp -r $MESA_DIR/star/work work_kevin_part2
```

2.2 Exercise 1 - Finding what MESA calculates

The global dynamical time scale of a star is given by

$t_{\rm dyn} \sim (G \langle \rho \rangle)^{-1/2}$

where $\langle \rho \rangle$ is some measure of the mean stellar density. Within prefactors, this is the free-fall time at the stellar surface as well as the Keplerian orbital frequency there. MESA already calculates a dynamical timescale at every cell in the star. How is it defined?

Search through MESA/star and find where this is calculated. What is the exact formula?

But what's the name of the variable to search for? That's part of what you're looking for. This may seem like a silly activity, but when you're writing your own code for MESA this type of question comes up over and over so it's good to get some practice hunting things down.

If you're having trouble, try narrowing your search to (highlight to reveal):

The definition you should find is (highlight to reveal):

which definition of the dynamical time does this prefactor correspond to?

2.3 Exercise 2 - Calculating the dynamical time scale(s)

Now that we know what MESA is calculating, what if we want a slightly different calculation? Perhaps we have a planet with a massive core and just want to use the central density for $\langle \rho \rangle$, or instead take a mass-weighted average of the densities of interior cells.

The first step is to tell MESA that you're going to be adding some extra user-defined columns to the profiles. This is done through the function `how_many_extra_profile_columns()` simply by making the return value equal to the number of extra columns you're adding. We'll start with just one,

```
how_many_extra_profile_columns = 1
```

The next step will be to fill in a placeholder for our dynamical timescale calculation, just to check that the code is set up correctly before we start trying to do calculations with the `star_info` data. This will go in the subroutine right below, `data_for_extra_profile_columns()`.

We can modify the `inlist_project` file if we want, but the default one will still spit out profiles every 50 steps, so we can leave it as it is. You should get 90 columns by default, so any you add will go at the end.

2.3.1 Center density approach

Let's start out really simple and make a new dynamical timescale based on just the central density of the star. This is a pretty useless value to put in a profile since it's not even spatially-dependent, but we'll start with the easiet thing to code up. Inside `data_for_extra_profile_columns()`, there's an example calculation in the documentation, we'll start out with a similar scaffold and just set everything equal to zero to check if things still compile.

```
!We're going to try and calculate some dynamical time scales:
names(1) = 't_dyn (central)'
do k = 1, nz
   vals(k,1) = 0d0
end do
```

Check that this works (you should get a 91st column in your profiles that's filled with all zeroes), and then figure out how to to the calculation with central density, $t_{\rm dyn} = (G \langle \rho_c \rangle)^{-1/2}$. One possible solution is (highlight to reveal):

Running this should make all the cells in your new profile column have the same value. For `profile10.data`, I got $t_{\rm dyn} \approx 6.3 \times 10^4$ s.

2.3.2 Mass-weighted average density approach

Another, more realistic, way to estimate the dynamical time would be to use a mass-weighted mean density of the interior cells. This requires for each cell k, looping over the cells interior to k (what indicies are those?) and finding the mass-weighted mean density.

If you get stuck, my code looked like this (highlight):

Remember to declare any new variables you need and to set

```
how_many_extra_profile_columns = 2
```

You should now have two extra columns, the mass-weighted calculation should be linearly increasing from the core outward and equal to the value of the central density method at the central point since our formulae should be the same there.

2.3.3 Comparison to sound crossing time

Our last goal is to compare these calculations with the sound-crossing time through the interior of the star, another measure of how fast parts of the star can respond to perturbations. This calculation should be similar to the previous section, except you'll want to approximate the integral

$t_{\rm sound}(r) = 2 \int_0^r \frac{dr}{c_s}$

in MESA. There are several ways to calculate this given the variables in the `star_info` pointer, give one a try. Again, my solution is below (highlight):

How is this different from the other values we calculated? Why do you think this is?

CHAPTER 3

Lab 3 - Specifying custom physics with `other_*`

3.1 Intro

In this part, we'll look at how to modify how MESA evolves stars (eg. the equations it solves, the micro/macrophysics it employs, etc.). There are many 'hooks' you have access to in order to change how MESA's internals function, and these are the `other_*` subroutines (see `mesa/star/other` for all of them). Each subroutine broadly lets you overwrite a subroutine that calculates some piece of physics for MESA/star.

The hook subroutine we'll use here is `other_mlt()`, which MESA uses as a helper method when solving for the structure of a star. It's called on each cell of the star during each Newton iteration and its job is to determine the mixing type of the cell and figure out its temperature gradient, given the local conditions. Again, we'll be starting with a fresh copy of the `work` directory, which you can make via

```
cp -r $MESA_DIR/star/work work_kevin_part3
```

3.2 Context for this code

This section is for your own info - not strictly necessary for the activities

Most of the code you'll modify through these hooks is far removed from the actual evolution step, located in `mesa/star/private/evolve.f`. The results you calculate are often passed through several intermediate wrapper routines before they're attached to actual variables in the `star_info` structure. The MLT module is no different, and we'll briefly trace the subroutine calls in this section. **Note: In your own research, you'll want to follow similar calls to whatever hook you're using. It's very important to know where your code gets called in relation to the overall evolve step to ensure that your modifications work as you intend. The following is presented as an example outline of how to trace the calls in MESA/star.**

To get some more context of what the MLT module does during evolution, look for where `mlt_eval()` is called in the private code. You should only see it called once - it's immediately put inside a wrapper called `do1_mlt_eval()`, defined in `mesa/star/private/mlt_info.f`.

So what does `do1_mlt_eval()` do? Aha! That's the subroutine that checks for a user-defined subroutine if we set `use_other_mlt = .true.` in the inlist. `do1_mlt_eval()` is in turn called from `do1_mlt()`, which finally attaches the results of the MLT calculation to the corresponding stellar variables. `do1_mlt()` then is called from `set_mlt_vars()`, which loops over a range of cells of interest and sets the MLT variables for each cell. We now know where our changes will show up in `mlt_info.f`, and what subroutines MESA will use to interface with this.

Where does `set_mlt_vars()` appear in the overall structure of the evolution? It is only called from the `set_hydro_vars()` subroutine located in `mesa/star/private/hydro_vars.f`. `set_hydro_vars()` is called in two main places:

1. During the Newton solve

Here, the calls initial come from subroutines in `mesa/star/private/hydro_mtx.f`, which contains subroutines that provide the implementation of various subroutines that get passed to the actual Newton solver. `set_hydro_vars()` is called in `set_vars_for_solver()`, which is wrapped by `set_newton_vars()`. This is then called from `eval_equations()` in `mesa/star/private/hydro_newton_procs.f` which is passed to the Newton solver though the `setequ()` subroutine in `mesa/star/private/star_newton.f`. The actual Newton solve happens in the `do_newton()` subroutine which is wrapped by `newton()`. The Newton solve is called from `hydro_newton_step()` inside `mesa/star/private/solve_hydro.f` through the wrapper `newt()`. `hydro_newton_step()` is called from `do_hydro_newton()` which is called from the function `do_hydro_converge()`. `do_hydro_converge()` is then called from `do_struct_burn_mix()` in `mesa/star/private/struct_burn_mix.f`, which is finally called during the top-level `do_evolve_step()` subroutine located in `evolve.f`.

2. After the Newton solve, during the main evolve step

In this part, `set_hydro_vars()` is called by the `update_vars()`, which is called by `set_some_vars()`, which is called by two top-level subroutines in `mesa/star/private/hydro_vars.f`, `set_vars()` and `set_final_vars()` (oh my). Finally, `set_vars()` and `set_final_vars()` are called in the top-level `do_evolve_step()` subroutine located in `evolve.f` (`set_vars()` is called through the wrapper `do_set_vars()`, since its functionality changes slightly before/after element diffusion). Specifically, `do_set_vars()` is called during the implicit $\dot{M}$ loop, and `set_final_vars()` is called near the end of `do_evolve_step()`.

So in summary - after 4 levels of calls, our MLT calculations in `run_star_extras.f` bubble up to the subroutine `set_hydro_vars()`, which is called during every step of the Newton iteration (another 9 levels of calls below the actual Newton solve in `do_evolve_step()`) as well as at the end of each call of each call of `do_evolve_step()` (via `set_final_vars()`).

3.3 Exercise 1 - Enabling other_mlt()

Following the `run_star_extras.f` tutorial on the MESA homepage, there are a few steps in telling MESA to use `other_mlt()`.

1. The first step is to copy/paste the contents of `mesa/star/job/standard_run_star_extras.inc` into your local `work_kevin_part3/src/run_star_extras.f` file.

1. The first step is to copy/paste the `null_other_mlt()` subroutine into `run_star_extras.f`, and rename it to something like `my_other_mlt()`.

2. Next, in the `extras_controls` subroutine of `run_star_extras.f`, you need to say you want MLT calls to go though this new subroutine by setting

```
s% other_mlt => my_other_mlt
```

3. Finally, you also need to enable this new subroutine with the `inlist` command

```
use_other_mlt = .true.
```

in the `&controls` section. While it may seem redundant, this inlist flag is there so you can turn your custom implementation on/off easily without having to recompile things.

Try everything out by compiling and running things as you normally would. Nothing should change in the output yet when you toggle the `use_other_mlt` flag since `my_other_mlt()` is still calling the normal MLT subroutine. It's good practice to put a write statement in there to make absolutely sure it's getting called - also so you can see how often it's being called since we have some expectation from tracing the subroutine calls above. *Note: you should be able to get an idea of how MESA is splitting the MLT calls among the different OpenMP threads from this output if your environment variable ''OMP_NUM_THREADS'' is > 1. Try it out if you can!*

Once you've verified that things are working as intended, you can comment out the output statements if you want since they really slow things down.

3.4 Exercise 2 - Modifying the standard convection prescription outside mlt_eval()

MESA includes a variety of MLT calculations (see eg. Cox and Giuli's Principles of Stellar Structure for the gory details) that relate the local chemical/thermodynamic conditions to the temperature gradient and compositional mixing rate.

Some aspects of this calculation are easier to change in the MESA implementation than others. We'll go through some of simpler cases first before tackling a more involved one in the next exercise. The first thing we'll try is to slightly increase the value for $\nabla \equiv \partial \log T / \partial \log P$ that `mlt_eval()` returns.

Look at the call signature of `mlt_eval()` as well as its implementation in `mesa/mlt/public/mlt_lib.f` along with variables in `mesa/mlt/public/mlt_def.f`. Figure out where ∇ gets returned and modify it after the `mlt_eval()` so that it gets increased/decreased by a fixed percentage. Start by making it 0.01% larger and increase from there. How much can you increase ∇ before MESA has trouble converging. What do you think may be causing problems?

You can also modify the chemical diffusion coefficent returned by `mlt_eval()` to adjust the speed at which convecttion smooths out chemical gradients. There is an inlist control for this in the `&controls` section,

```
! mixing coefficients are multiplied by this factor
mix_factor = 1
```

which scales all of the mixing coefficients (eg. convection, thermohaline, semiconvection, rotational mixing, etc.). While the specific rotational mixing components can be multiplied by specific factors (see `mesa/star/defaults/controls.defaults`), you have to go into the code if you want to modify the mixing rate of, say, convection alone.

Modify the variables returned by `mlt_eval()` so that you can scale the diffusion coefficient by a constant factor. You may want to change the stopping criterion to be somewhere past the ZAMS so you can see a difference in the evolution. *In practive, perhaps you'd want to make this scaling a function of position, or thermodynamic conditions - hopefully you can see how this could be done (you don't need to actually do something like this here though).*

3.5 Exercise 3 - Adding custom inlist controls

For the modifications we just made, it would be convenient if there was some way to toggle them or control their strength from the inlist. We can use additional controls, defined through the following variables (see `mesa/star/defaults/controls.defaults`)

```
! extra params as a convenience for developing new features
! note: the parameter 'num_x_ctrls' is defined in 'star_def.f'

x_ctrl(1:num_x_ctrls) = 0d0
x_integer_ctrl(1:num_x_ctrls) = 0
x_logical_ctrl(1:num_x_ctrls) = .false.
```

What is `num_x_ctrls` set to by default? Can you change this in the inlist? How do you refer to these parameters through the `star_info` pointer, `s`?

Once you've answered those questions, try adding some sensible inlist controls to the modifications you just made. As an example, one could be toggling the ∇ increaese on/off, or adjusting its strength. Check that such things work before moving onto the next part.

3.6 Exercise 4 - Modifying the standard convection prescription inside mlt_eval()

Here, we'll slightly modify the MLT calculation to make the mixing length scale with temperature scale height instead of pressure scale height. This part is much more involved, so take your time.

From looking at the call signature of `mlt_eval()`, it doesn't get the scale height passed to it (just a logical flag `alt_scale_height` and the `mixing_length_alpha` parameter that multiplies the scale height calculated inside). That means we have to actually go into the internals to change the definition of the scale height used in the MLT calculation.

A brief outline of how to go about this:

1. Since we want to change how `mlt_eval()` functions without touching the private code, we should copy the entire subroutine into our `run_star_extras.f` file. Search through `mesa/mlt` to find where it is defined, you should find it in (highlight to reveal):

Unfortunately, it looks like `mlt_eval()` is just a wrapper for the private function `do_mlt_eval()`, which we can again search for, finding it in (highlight to reveal):

The actual code we want to modify is `do_mlt_eval()`, and the various supporting methods it calls. You should therfore copy/paste all the subroutines and functions in this file into your `run_star_extras.f`. Once they're in there, then they should replace the call to MESA's public subroutine `mlt_eval()` with a call to your copy/pasted local implementation, `do_mlt_eval()`. You should comment out the

```
use mlt_lib, only: mlt_eval
use mlt_def
```

lines in `my_other_mlt()` to make sure that it looks in this file for the modified subroutines instead of using the MESA defaults.

2. In order to compile this new code, you'll have include the MLT module at the top of `run_star_extras.f`. Along with the other `use` statements, add the others that the copy/pasted subroutines are expecting,

```
use mlt_def
use mlt_lib
use crlibm_lib
```

You will also need to add the module variable (can put it right between the `implicit none` and `contains` statements)

```
integer, parameter :: nvbs = num_mlt_partials
```

Finally, you also need to remove all the `#ifdef` blocks to get things to compile with the default makefile.

3. Once you've copy/pasted the private implementation and made the changes above, check that everything still works as it should by compiling and running the code. Make sure all the functions required by `do_mlt_eval()` are now in `run_star_extras.f`.

4. Now we can finally start making modifications to the convection routine! In the interest of time, the subroutine that actually calculated the MLT results for convection is `standard_scheme()`, while the scale height is define in `Get_results()`. Search for all the places where the scale height is used and instead of using the pressure scale height, use the temperature scale height.

How do you calculate the temperature scale height here? The star_info pointer doesn't get passed here! One way is to extract the `star_info` pointer from the `id` variable in `my_other_mlt()` with the `star_ptr()` subroutine (located in `mesa/star/public/star_def.f`) that fills in a `star_info` pointer that you have to declare (`s` here) via the `id` variable.

```
call star_ptr(id, s, ierr)
```

You can then extract the data necessary to calculate the temperature scale height and pass it down to `Get_results()`. You have to be careful here and pass it as an argument rather than making it a module variable since there is a different value for every cell and the `my_other_mlt()` subroutine will be called in parallel (as you saw in Exercise 1). You may be able to get away with making it a module variable using the OpenMP directive,

```
!$OMP THREADPRIVATE(temperature_scale_height)
```

but I'm not sure if that would prevent all the possible problems, and that's way beyond the scope of this exercise (did anyone actually get this far anyway??).

One last important tip is when you calculate things from `star_info` in these subroutines, you can't always count on the arrays being allocated and filled with data (especially while the star is evolving on the pre-MS). If you want to do a calculation in `my_other_mlt()` using the `star_info` pointer `s` you extracted, then you need to check that the arrays are actually populated. One way to do this (not sure if it's the best, but it has worked for me) is to wrap all your calculations in an if-statement such as

```
if(k.gt.0) then
   !Calculate things using arrays like s% P(k) without running into segfaults
endif
```

After all that effort, can you think of some stellar evolution contexts where this change would make a significant difference?

3.7 Defining your own mixing prescription

You can also write your own mixing routine, based on what you see calculated in other subroutines such as `standard_scheme()`, `set_thermo_haline()`, or `semiconvection()`. Notice how all of these are called in similar ways from `Get_results()`. They all set a few variables:

1. `mixing_type` - an integer specifying what kind of mixing is occuring in this cell. See `mesa/mlt/public/mlt.def` for their definition.
2. `gradT` - the actual temperature gradient determined for this cell, ∇.
3. `D` - the compositional diffusion coefficient which controls the speed of chemical mixing.
4. `conv_vel` - a characteristic velocity of this type of mixing, usually set to 3D/(mixing length) in an isotropic process (cf. Fick's Law).
5. `d_gradT_dvb` - partial derivatives of `gradT` with respect to all the other MLT variables (see the call structure of `Get_results()` and compare to what's passed to it in `do_mlt_eval()`).

According to Bill (see this thread), the only partial derivative you explicitly need to calculate is that of `gradT` - the others are already done for you in `Get_results()`.

We won't have time to implement a new mixing prescription here, but you will in Pascale's lab - have fun!

CHAPTER 4

Part 4 - other_* reference page

4.1 Intro

There are quite a few other_* routines (also referred to as hooks) that MESA provides. Rather than modifying the private code, these are the way to access and modify MESA's internal functions. If you'd like to modify something that you don't see an obvious subroutine for, then Bill will be happy to add a new one for you. Here's a list of some of the more commonly used hooks:

Please let us know (eg. by email to the list) if something becomes out of date, if you find an error, or if there's another use case you think we should explicitly mention for one of the hooks.

4.2 Hooks

- other_mlt() - Allows you to change the implementation of mixing length theory (MLT) that MESA uses to determine what type of mixing occurs in a cell. The actual temperature gradient is also calculated here, along with the diffusion coeffienct for mixing, among other related quantities. For details on the expected return variables, see mesa/mlt/public/mlt_def.f and mesa/mlt/public/mlt_lib.f. Example uses for this hook include defining your own local mixing prescription, modifying the thermal and compositional transport rates for standard convection (or any other mixing type), etc. *Note: Mixing is not actually performed in this subroutine. If you want to modify how MESA mixes things, you need to look at other hooks such as ‘‘other_split_mix()‘‘ or ‘‘other_diffusion()‘‘*
- other_energy() - Allows you to specify an anomalous heating/cooling rate in the star. This is pretty simple, since it's not replacing a call to another MESA routine like some of the other hooks.
- other_wind() - Allows you to specify a mass-loss prescription (eg. a formula for $\dot{M}$ as a function of other stellar variables). Can also be used for more general mass-loss scenarios like Roche-lobe overflow.
- other_eos() - Allows you to use your own implementation of an equation of state. An example would be using FreeEOS within MESA.
- other_kap() - Allows you to implement your own routine for calculating opacities. Examples include trying to calculate them on the fly instead of looking them up from tables (still too slow according to those who have tried), as well as forcing things like pure electron conduction or electron scattering in certain regions of a star/planet.
- other_torque() - Allows you to implement other sources of torque in rotating stars. This could be used, for example, to implement a magnetic breaking prescription.
- other_mesh_functions() - Allows you to define other functions that MESA will examine when determining when to increase/decrease resolution.

- `other_diffusion()` - Allows you to implement your own diffusion routine in between stellar time steps. This bypasses the entire default diffusion solver, so you either need to copy/paste and make appropriate changes or write your own from scratch.
- `other_atm()` - Allows you to specify a custom atmosphere boundary condition. Could be useful in trying to contstruct models to match known planets.
- `other_split_mix()` - Allows you to modify the composition profiles after the mixing step.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*