Binary_grid: Flexigrid

Written by Robert Izzard, with greatly appreciated help from Fabian Schneider, Sutirtha Sengupta and the other users of *binary_grid* and the *flexigrid*.

Contents

1 Introduction

The *binary_grid* software has been developed over the last ten years or so by Robert Izzard with help from many people around the world. It is designed to work with the *binary_c/nucsyn* stellar evolution code, based on *BSE* of Hurley et al. (2002), but should work with any suitably prepared stellar evolution code. In 2011 and 2012, the old *binary_grid* code has been replaced with the *flexigrid* – this is the guide to the this version of software.

Before using *binary_grid* please note that it is subject to the *binary_c* licence file which is included with binary_c, the only difference being that while *binary_c* is based on *BSE*, *binary_grid* is entirely Robert Izzard's creation. As stated in the licence, appropriate citations (e.g. Izzard et al., 2004, Izzard et al., 2006 and Izzard et al., 2009) are *mandatory* if you publish work which uses *binary_grid* or *binary_c*, and co-authorship is an option which should be considered. Please remember that the amount of work that has gone into this project, especially at weekends and during holiday time, is quite considerable: give credit where it is due and help Rob to get another job!

Frequently Asked Questions are in Section 14 and installation instructions are given in Section 15.

It is up to you to test and debug this code because, as with all codes, it is not perfect and probably does not do what you would like it to do without some tweaking. You can contact Robert Izzard via his website http://www.astro.uni-bonn.de/~izzard/ (which gives an email address) or use the *binary_c* mailing lists:

- binary_c-nucsyn-announce@googlegroups.com at
  https://groups.google.com/forum/?fromgroups=#!forum/binary_c-nucsyn-announce
- binary_c-nucsyn-devel@googlegroups.com at
  https://groups.google.com/forum/?fromgroups=#!forum/binary_c-nucsyn-devel
- Facebook page at
  https://www.facebook.com/groups/149489915089142/?fref=ts

2 Aims: what the grid does and why

Binary population synthesis is a glorified form of accountancy, without the huge pay but with of course much more fun! It involves modelling perhaps millions of binary stars in a population to pick out those few that contribute to the population of scientific interest. The rate of formation, or number of these stars, or some other property of them can then be examined in a *quantitative* and *statistical* manner. This manual explains how the *binary_grid* module – part of the *binary_c/nucsyn* code package – can be used to model populations of single and binary stars.

For details about the binary stellar evolution code *binary_c/nucsyn* please see the nucsyn_manual document. This guide is *only* concerned with the details of running grids of models, not about the details of the models themselves.

2.1 Grid dimensions

There are many parameters which can be set before running a stellar model. The most important, for single stars, are the stellar mass M and metallicity Z. It is common to set the metallicity Z to be constant and vary the initial stellar mass M. In everything that follows I will assume you are working at a constant metallicity Z.

The situation is more complicated in binaries because instead of just M there are two stellar masses M_1 and M_2, the separation a (or, equivalently, the period P) and perhaps the eccentricity e. To run a population of binary stars then requires a grid of models in four dimensions. Often the eccentricity is ignored because close binaries – which are the ones in which you may be interested – tend to circularise rapidly. This reduces the problem to three dimensions.

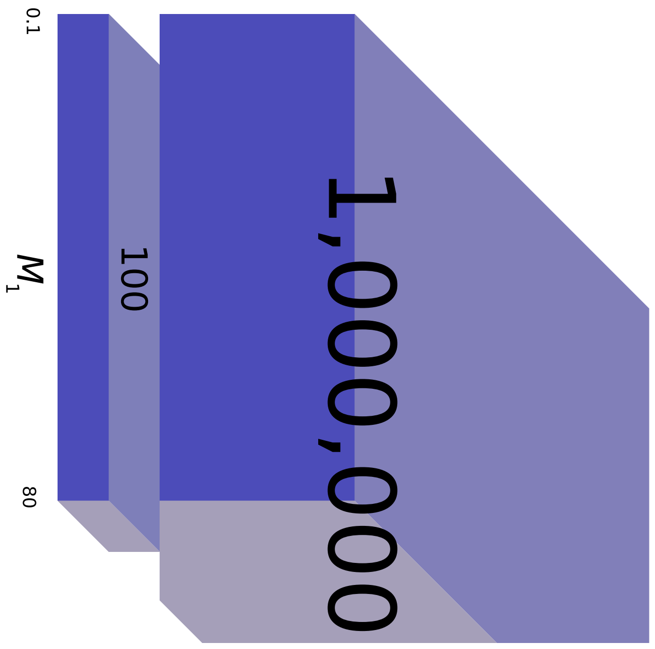

Figure 1: The relative size of the single- and binary-star grids. In single stars only one mass is required, M_1, while in binaries the two masses M_1 and M_2 and separation are needed. For a grid resolution of $n = 100$, the runtime for the single stars (left cuboid) is $100 \times \Delta t$, where Δt is the runtime for one stellar system. In binaries (right cuboid) this increases to at least $10^6 \times \Delta t$ because of the extra grid dimensions. In addition, Δt will be longer for binaries because of the short timesteps required during some phases of mass transfer, the fact that there are two stars to follow in the evolution/nucleosynthesis algorithms and extra overhead from logging and parsing output from both stars. Typically, $\Delta t \sim 0.1$ s on a 3 Ghz PC.

The number of stars on each side of the grid, i.e. the *resolution* n, roughly determines the total runtime of your simulation. In single stars this is just $\Delta t \times n$, where one model takes a time Δt to run. In the case of binaries on a three-dimensional grid, this increases to $\Delta t \times n^3$ (see Fig. 1). In a typical population $n \sim 100$ so the total runtime increases by a factor of $100^2 = 10^4$. This ignores the increase in Δt for binaries which is unavoidable because there are two stars under consideration as well as smaller timesteps during mass transfer. It is for this reason that the *binary_c/nucsyn* code (and its BSE ancestor) must be – and is – fast. It can run a population of 10^6 binaries in less than 24 CPU hours.

2.2 What to do with the grid

What do we do once the grid of stars is set up? In most cases this involves adding up some statistics related to each star on the grid. There are two approaches which are commonly used, constant star formation and a starburst. The one that suits you depends on what you are trying to calculate and to which observations you hope to compare.

2.2.1 Constant star formation rate

In this case the formation rate or (perhaps δt-weighted) probabilities for each star corresponding to those in which you are interested are simply added up. This is why I say it is like accountancy! Because the rate of star formation is assumed to be *constant* the evolution time of the stellar evolution is irrelevant. A hybrid scheme can also be used, you might only consider stars older than a given age (e.g. for halo stars).

2.2.2 Starburst

All stars form at time zero and you add up the (probably δt-weighted) number of stars in which you are interested which form or exist in each time bin. A good example is the delay-time distribution of type Ia supernovae.

The statistic you wish to add up is usually output from *binary_c/nucsyn.* Your “grid script” runs each star in your population, takes the output of *binary_c/nucsyn* and adds up the statistic. You can generate single numbers, histograms or arbitrarily complicated statistical constructs for comparison with observations. You can choose whether to do the calculations on the fly or save the progenitor information for later data processing.

The grid script can be used on multi-CPU machines and handles the administrative issues regarding parallel processing.

3 Grid setup (in theory)

This section presents the *theory* behind the setup of a grid in M, for single stars, or M_1, M_2 and a for binary stars. In general a *logarithmic* spacing is chosen for the mass grid because the important relevant physical processes are “more different” for the mass range $1-10\,M_\odot$ than between $10-100\,M_\odot$. This is rather subjective but you can chose the bounds of your grid to reflect the stars in which your interest lies.

Each star in the grid represents a *phase volume*. This means that a star of mass M represents stars which, in reality, have masses $M-\delta M/2$ to $M+\delta M/2$, where δM is the grid spacing. In an ideal world δM would be very small, so our simulations match reality as closely as possible. However, this is not usually possible if the grid is to run in a reasonable amount of time (before funding runs out). The usual way around this is to 1) be clever and/or 2) use a brute-force approach where the resolution is increased (in this case δM decreased) until your “answer” converges.

One concept which must be grasped is that the bounds of the *grid* are not necessarily the bounds of the *initial distributions* you will be using (e.g. the initial mass function, IMF). This is a critical point which can lead to confusion, it is best explained with an example. Say you wish to calculate the ratio of type II to type Ib/c supernovae. You would do this by adding up the number of stars that explode as type IIs and divide by the number of stars that explode as type Ib/cs. You *could* do this with a grid from 0.1 to $100\,M_\odot$ and, given enough stars, you would come up with the correct answer. This is easy for single stars, because 10^4 single stars are easily run on a modern PC. However, for binaries this is difficult, because even with only 100 stars in the M_2 and a dimensions, there would be 10^8 stars in your simulation and the runtime would be huge. You can, however, be smarter and reduce the grid bounds and resolution. You know in advance that at solar metallicity ($Z=0.02$) type II and Ib/c supernovae occur in *single stars* with $M \gtrsim 8M_\odot$. If you choose M_1 to run between (say) 4 and $100\,M_\odot$ you know you will record the supernovae from mass transfer (as $M_2 \leqslant M_1$ so the minimum mass of merged stars is the required $8\,M_\odot$) as well as all the others at higher masses. So, you will get the *same answer* from a grid between 4 and $100\,M_\odot$ as with a grid between 0.1 and $100\,M_\odot$. Note that the *grid* has changed, but the result has not. This is because the initial distributions have fixed bounds independent of the grid bounds. See Section 8 for more details on initial distributions.

This process requires some knowledge in advance, which you may not have. In that case, run a low resolution grid, find suitable grid bounds and then repeat at higher resolution until you are happy. The downside to the process is that you may well miss small parts of the parameter space which contain interesting systems. In practice this is often a compromise worth making and, if in doubt, test your model runs at low resolution but then crank the resolution to the maximum you think is possible for the final model run.

Another advantage of choosing the grid bounds is that sub-grids can be run on different computers or CPU cores and the results combined. This is discussed further in Section **??**.

3.1 Phase Volume

Given a chosen set of these distributions a logarithmic grid is set up in one-dimensional M space for single stars or 3D M_1–M_2–a space for binary stars. The grid is split into n stars per dimension such that each star represents the centre of a logarithmic grid-cell of size δV where

$$\delta V = \begin{cases} \delta \ln M & \text{single stars} \\ \delta \ln M_1\, \delta \ln M_2\, \delta \ln a & \text{binary stars} \end{cases} \tag{1}$$

and

$$\delta \ln x = \frac{\ln x_{\max} - \ln x_{\min}}{n}\,, \tag{2}$$

where x represents M, M_1, M_2 or a and $x_{\max}$ and $x_{\min}$ are the grid limits. The total number of stars is denoted by N such that $N = n$ for single stars and $N = n^3$ for binary stars.

The probability of existence of star i is given by

$$\delta p_i = \Psi \delta V_i = \Psi \delta V \,, \tag{3}$$

because δV is a constant. The function Ψ is discussed in Section 8.

The probability, δp_i, is sometimes the quantity you wish to add up, although commonly you want $\delta p_i \times \delta t$ where δt is the time spent in the phase of interest. This is usually the case when comparing the number of stars which go through a particular phase of evolution because the *time* in the phase of evolution is as important as the *probability* of the star existing in the first place. However, for *event rates* (e.g. supernovae) the time involved is zero, they are instantaneous events on a stellar evolution timescale, so you have to sum up the δp_i. You have to be careful to calculate the appropriate statistics for comparison with the observations.

3.2 Grid loops

In binary systems, the secondary mass M_2 typically depends on the primary mass M_1, and the separation, a, or period P, may depend on both. This means that usually the grid must be constructed in a series of nested loops:

$$\sum\nolimits_{M_1=M_{1,\text{min}}}^{M_{1,\text{max}}} \left(\sum_{M_2=M_2}^{M_{2,\text{max}}} \left\{ \sum_{a=a_{\text{min}}}^{a_{\text{max}}} [\ldots] \right\} \right) . \tag{4}$$

In the most grids, M_2 depends on M_1 through the mass-ratio, $q = M_2/M_1$, distribution. The default mass-ratio distribution is flat in q (see Section 8), with $0.1\,\text{M}_\odot \leqslant M_2 \leqslant M_1$. The period or separation distribution may also depend on M_1 and M_2.

3.3 Resolution

The choice of grid resolution depends very much on the problem you wish to address and what you are trying to calculate. As a general rule, assuming you are modelling stars from $M_{\text{min}} = 0.1\,\text{M}_\odot$ to $M_{\text{max}} = 100\,\text{M}_\odot$ you probably want at least 100 stars in the M (or M_1) dimension.

However, before rushing off to run millions of stars, run some low-resolution test models and gradually increase the number of stars. The best way to determine the required resolution is simply the brute force method: keep increasing the resolution until your answers converge.

You can roughly estimate the *statistical* error by counting the number of stars that satisfy your criterion (say k stars) and then use Poisson statistics (i.e. the error on a value x is $x/\sqrt{k}$). **Be warned that this is *not* the error due to the use of a grid.** If your data calculation binning is *over*-resolved you may see problems with artefacts and aliasing. These may or may not affect your results. If in doubt, increase the resolution until the problem goes away (if it goes away!). In some cases this may not be possible because your computer is not fast enough. You could always buy a time machine... but then you wouldn't be doing astrophysics!

Overall resolution is often not your only concern: perhaps you want more stars in a certain region of the parameter space. Well, with the flexigrid and its spacing_functions module you can define your own grid spacing function which as flexible as you like (see Sections 7 and 9).

4 Data parsing (Accountancy without the salary)

Before you actually run your own population synthesis grid you have to think about what you want to discover from your stellar models. A few standard cases are outlined here. You may, of course, want a combination of all these methods.

This section describes the theory, the practical aspects of combining *binary_c/nucsyn* with your grid script are described in the following, e.g. Section 6.

4.1 Rates of things

The simplest thing to calculate is the rate of formation of some type of star, or alternatively the rate of stellar death (e.g. the supernova rate), merger rate, etc. These are *events* which have no duration so the important thing to log is the *time at which the event happened*, t_i, and *the probability of existence of the stellar system,* p_i.

You will probably want to count the number of systems that explode in a given time bin, i.e. between time t and $t + \Delta t$, where Δt is the time bin width (*not* the *binary_c/nucsyn* timestep!). The formation rate is the product of the star formation rate S and the summed probability of the stars that do whatever it is that is interesting in time t to $t + \Delta t$, $\sum p_i$. You have to somehow calculate S (see e.g. Hurley et al., 2002 for a simple prescription). You would be better comparing the relative rates of two types of event, in which case S (and its associated uncertainty) cancels out.

This is all correct for a starburst at time $t = 0$. If you want to use a more realistic star formation history with varying metallicity you have to convolve the results of many starbursts. You can do this manually (and it is easy if, say, the metallicity and other physics is not a function of t) or use a pre-existing code such as Rob's Galactic Chemical Evolution (GCE) code. The latter has the advantage that it already exists and is well tested. You will, however, have to ask Rob about it because it is not (yet) a part of the standard *binary_c/nucsyn* distribution.

4.2 Numbers of things

Alternatively, perhaps you wish to calculate the number of stars of a certain type e.g. the number of carbon stars or the number of K-type stars. In this case you have to count both the probability of the existence of the star p_i and the time the star spends in the evolutionary phase of interest. If we assume $S = 1$ (i.e. constant star formation) you should then count $\sum p_i \, \delta t \, \bar{\delta}$ where $\bar{\delta} = 1$ if the star is interesting to you, but zero otherwise.

As in the previous example, if S is not simply a constant things get tricky. But usually a constant star formation rate is “good enough” for e.g. Galactic stellar population studies of anything but the youngest stars. In other cases a starburst is more appropriate and you will have to factor in a check on the time t into $\bar{\delta}$ to match the present-day age of the population.

5 Grid setup (in practice: the flexigrid*)*

The grid is set up by use of a Perl module called *binary_grid*. This is available with *binary_c/nucsyn* in the *src/perl/modules* directory (see also the *binary_c/nucsyn* manual). *binary_grid* uses a number of secondary modules as well as some standard (CPAN) Perl modules.

The newest version of *binary_grid* uses the *flexigrid.* For details on this part of the grid, see Section 7.

5.1 Your grid script

You load the *binary_grid* Perl module from your *grid script* (see Sec. 6 for full details). This is a Perl script which contains the definition of both your population synthesis grid and your input physics. Do not make this yourself: instead, use the example scripts in the *src/perl/scripts* directory of the *binary_c/nucsyn* tree and modify them appropriately (Sec. 6).

5.2 The binary_grid *Perl Module*

The *binary_grid* Perl module contains (almost) all the code to set up, manage and run the grid, as well as the interface to the *binary_c/nucsyn*. This should be (almost) completely transparent to you, the user, and the process is as automated as possible. You still have to do something, of course, and – most importantly – decide on input physics.

5.2.1 Variables

You can set variables in two data hashes *binary_grid::grid_options* and *binary_grid::bse_options*. Usually you do this in the *defaults* subroutine of your grid script.

- The *grid_options* control the behaviour of the grid and allow you to set up the *flexigrid*. These options control the grid loops, logic, logging, threads etc.
- The *bse_options* control the binary-star physics, such as the metallicity, common-envelope parameters, etc. The names of the *bse_options* keys (mostly) match those of the arguments to *binary_c*. This is deliberate: if you add a command-line option to *binary_c* you can add it directly to *bse_options* to have it work immediately.
- In your grid script you need a globally defined hash to store the population synthesis results, usually called *%results*. You must tell *binary_grid* what it is called, e.g. in your *defaults* subroutine (see Sec. 5.3) insert
$binary_grid::grid_options{'results_hash'}=\%results;

5.2.2 Important Subroutines in *binary_grid*

Only the most important subroutines which you may call from your grid script are listed here.

grid_defaults sets up the default values in the *binary_grid::grid_options* and *binary_grid::bse_options* hashes. You have to call this before you do anything else with *binary_grid*.

setup_binary_grid After setting up your *binary_grid::grid_options* and *binary_grid::bse_options* hashes, call this function to finalise setup.

parse_grid_args This loops over the command-line arguments of your script and looks for them in the form *x=y*. If *x* matches a *grid_option* or *bse_options*, it is set appropriately to *y*. (You can pass an array of arguments to *parse_grid_args* which is used in place of the command-line, otherwise it defaults to *@ARGV*.)

flexigrid(n) Call this subroutine to launch your grid after you have done all the above setup. *n* is the number of threads.

5.3 Important Subroutines in your grid script

defaults (required) You have to set up the variables of your grid before doing anything else, usually in a *defaults* function. This must call *grid_defaults()* and then override variables in the *binary_grid::grid_options* and *binary_grid::bse_options* hashes. It must also set the code reference to the *results_hash grid_option*, e.g.
$binary_grid::grid_options{'results_hash'}=\%results;

parse_bse (required) You have to write a function to parse the data which comes from *binary_grid*, as described in Section 6. You can tell *binary_grid* about your function by setting *$binary_grid::grid_options{'parse_bse_function_pointer'}=\&my_function_name;* although binary_grid should use *parse_bse* (actually *\&main::parse_bse*) by default. The first argument passed to *parse_bse* is always a pointer to the thread's data hash, usually referred to as *$h*. This stores all the data in the hash and is added to the global *%results* hash when the thread finishes so you can output the result of your population synthesis (see Section 10.1 for more details on how data is joined to make the results).

tbse_line This gets a line of data from *binary_c/nucsyn*. This should be called by parse_bse (see examples in Section).

join_thread (optional) You can override the default thread-joining function with your own, e.g. with
$binary_grid::grid_options{'threads_join_function_pointer'}=\&join_thread;

output (required) After the population synthesis is complete, you have to output data to screen or (better) to disk. This function should do this.

6 *Your* grid script *and coupling your population synthesis to* binary_c/nucsyn

This describes what you should do in your grid script in order to make the *flexigrid* work. Learn about the *flexigrid* in Section 7.

Before you start: you should copy the `grid-flexigrid.pl` script in the `src/perl/scripts` directory to a script of your own naming (usually grid-*xxx*.pl where you choose a suitable label to replace *xxx*). Work on your copy, not the original!

6.1 *Logging in* binary_c

When *binary_c* is run for one star with the `tbse` script the output is dumped to the screen (known in *UNIX* as `stdout`). A standard build of *binary_c* outputs very little information because it does not know what you wish to know (this is a *known unknown*). Every line that is output is expensive in terms of CPU time, and CPU time is critical when running millions of stars. This means you have to think carefully about what you will output from *binary_c* and when. Too much information slows things down, too little is useless – you have to decide.

You may want to output one line of information per *binary_c* timestep. This should be done in the `iteration_logging` subroutine which is called every timestep. Your logging code should look something like:

```
printf(“MYLOG %g %g %g %i\n”,
                stardata->model.model_time,
                stardata->model.probability,
                stardata->model.dt,
                stardata->star[1].mass,
                stardata->star[1].stellar_type);
```

The string `MYLOG` is critical: this is what will be recognised and used by *binary_grid*. You should have a think (and consult Section 4) to determine whether you want to output either the probability (`stardata-> model.probability`) or both the probability and timestep (`stardata->model.dt`). Often it is the number of stars in a particular phase that you want to count, in which case you want to add up the product: `stardata-> model.probability*stardata->model.dt`, also known as *dtp*.

6.2 *Coupling your* binary_c/nucsyn *and your grid:* `tbse_line` *and* `parse_bse`

Now you know how to make lines of logging information come out of the *binary_c* program. It remains to couple this to the grid. This is done through the `tbse_line` subroutine in the `parse_bse` function (see the example `grid.pl`). Each time `tbse_line` is called it gets a line of output from *binary_c* and loads returns it (usually to `$_` the Perl default variable). It is then up to you to decide what to with the data from `tbse_line`.

There may be many lines of data pumped into `tbse_line`, so you have to choose those labelled with your header string – in the above example this is `MYLOG`. This is best done with a Perl substitution-regular expression: the regular expression matches `MYLOG`, the substitution removes it (because it is not data). In the following example note that the check for ’`fin`’ *must* be there. *binary_c* returns a ’`fin`’ when the star has finished its evolution. Without this it would be stuck in an infinite loop, so avoid removing the `$brk` stuff.

```
$h=$_[0]; # data hash
while($brk==0)
{
     $_=tbse_line(); # get line of data from binary_c
```

```
        if($_ eq 'fin')
        {
             $brk=1; # the end of output
        }
        elsif(s/^MYLOG //) # regular expression to match and remove MYLOG
        {
             my @x=split(' ',$_); # convert space-separated data string to array
             #...  do stuff with @x ...
             # e.g.  $x[0] is the time, $x[1] is the probability etc.
        }
  }
```

6.3 Adding things up

So you have the probabilities (and maybe the timesteps) for each timestep of *binary_c* and for each star. What do you do now? Well, you have to create meaningful statistics by summing up the results. You should put results into the *$h* hash pointer as follows:

```
        elsif(s/^MYLOG //) # regular expression to match and remove MYLOG
        {
             my @x=split(' ',$_); # convert data string to array
             my $dtp=$x[1]*$x[2]; # calculate dt * p
             if($x[3]>8.0) # check mass (element 3 of @x)
             {
                  # high mass star
                  $$h{'high-mass'}+=$dtp;
             }
             else
             {
                  # low mass star
                  $$h{'low-mass'}+=$dtp;
             }

       }
```

6.4 Joining the results

The *$h* hash pointer is unique to a single thread: it does not contain your final result, just part of it. You have to add up the results by *joining* the various threads. In practice this is done automatically by binary_grid in combination with some Perl wizardry. You can, however, override this yourself by writing your own join function, which should be set in the following manner:
$binary_grid::grid_options{'threads_join_function_pointer'}=\&join_thread;

6.5 Outputting the results

At the end of the grid.pl script you should output the results, usually by writing an *output* subroutine and calling it. It is up to your to do this: *binary_grid* cannot possibly know your results or how you have structured your results hash.

6.6 More advanced accountancy

Consider the previous example. Perhaps instead of simply having "low" and "high" mass stars you actually want to calculate a histogram of number of stars as a function of mass at a certain time.

The problem of how to force *binary_c* to output at given times is a tricky one which is not dealt with here. However, you should consider following the `VROT_LOGGING` code in *binary_c* (see `deltat.c`) which does just this. Let us assume you have fixed *binary_c/nucsyn* so it outputs the mass of each star every 1 Myr. You want to bin the masses at each timestep, so you will have a series of histograms. Note that `$dtp` is not required because you are not counting the number of stars in a given *phase* but rather an instantaneous property of the stars at a given time `$t`.

Something like this will do it:

```
elsif(s/^MYLOG //) # regular expression to match and remove MYLOG
{
     my @x=split(' ',$_); # convert data string to array
     my $t=$x[0]; # time (an integer number of Myr since the starburst)
     my $p=$x[1]; # probability
     my $m=$x[3]; # stellar mass

     # bin stellar mass in 1Msun bins
     $m = rebin($m,1.0);

     $$h{'mass histogram'}{$t}{$m}+=$p; # construct histogram
}
```

Things to note include:

- We use the *rebin* function of *binary_grid* to do the binning very efficiently to the nearest 1.0 $M_{\odot}$.
- You can construct a similar expression for an arbitrary bin width. Why do we do this? Well, if we take the instantaneous mass of each star in a population at a given time there will be a huge number of different masses because mass loss may be effective. This would give a histogram with $N \times N_t = n_{M1} \times n_{M2} \times n_a \times N_t$ bins where N_t is the number of timesteps of *binary_c* output. This is too much data to deal with and negates the whole purpose of making a histogram! Bin your data: it makes life a lot easier.
- The time and mass are integers in this case, so we could have used arrays (which have integer index) instead of hashes for the output, but if you want finer resolution they will not be and hashes are *required*. It is good to get into the habit.
- The 'mass histogram' hash label is not really necessary in this simple example because nothing else is stored, but in a real grid you may be constructing many different histograms, each will required a different label. It is good to get into the habit of using nested hashes.
- The results are saved in hash of hashes of hashes. This is a typical example of a Perl nested variable, help for how to deal with (and output) hashes of hashes is easily found with a simple web search (or consult the excellent *Programming Perl*).
- Beware resolution! Always test different resolutions and beware binning and aliasing effects.

6.7 Save your data – or process on the fly?

Before running a grid you have to make the big decision: do you save your data or process it all on the fly? There are advantages and disadvantages to both, as described below.

6.7.1 Saving data

In this case you run a grid and tag the initial parameters of each system of interest (i.e. the M_1, M_2, a and $\delta V = \delta \ln M_1 \, \delta \ln M_2 \, \delta a$). *You save these initial parameters into a large datafile.* The big advantage of this technique is that you can rerun the saved grid *from the datafile,* possibly without running *binary_c/nucsyn* again, and with a different initial distribution function (e.g. initial mass function) or very slightly different physics (on the assumption that changing the physics will not change the systems which were saved).

You could save, as well as the initial parameters of the systems, the time spent in the phase of interest, Δt. Each time the grid is rerun you would then calculate the probability p_i for each system again, hence you know $p_i \Delta t$ which is your required "number of stars". You can either

1. Change p_i and recalculate your statistics. This does *not* require the running of *binary_c/nucsyn* and is the fast option.

2. Change p_i and some physics, which requires rerunning of *binary_c/nucsyn*. This is slower, but still faster than running the full grid.

Saving the progenitor data set is a very useful way of saving time after an initial high-resolution grid run.

6.7.2 On the fly data processing

If you do not want to recalculate your answer with different initial functions, you might be better off calculating the results you require "on the fly". This means you do your data processing *in your grid script*. The main advantages to this are speed and data storage. For example, if you have 10^6 stars in your grid, that is 10^6 lines of data – this might be a lot of data.

In some cases you *must* do this. If you are looking at data which is time dependent then you will have $10^6 \times N_t$ lines of data, where N_t is the number of timesteps during which your system is of interest. N_t may be many hundreds for a given system, hence the total number of lines of data is of the order of 10^8 and the data storage requirements become quite ridiculous. For example, a typical line of data has about 200 characters, so $200 \times 10^8 = 2 \times 10^{10}$ bytes of data, or 20 GB. The time it takes to process all this data may be longer than the time to *rerun binary_c/nucsyn* for each system, especially if your data storage is non-local (e.g. on an NFS (network) partition).

You can save some space by outputting data every, say, Myr, but this may not be the time resolution you require to catch systems of interest.

7 Flexigrid

As of summer 2011, the “flexigrid” was developed. This is an attempt to replace much of the old grid framework with a new, dynamic set of instructions which are much more flexible and, at the same time, more efficient. The key concepts are the following:

- When a flexigrid is run, it sets up n concurrently running *threads*. Each of these is fed with stars until the population is complete. Because of this, flexigrid is designed specifically to work on the latest multi-core CPUs on shared-memory machines (which are the typical PCs of the early 21st century).
- Flexigrid sets up its $M_1 - M_2 - a$ (or P) etc. grid *dynamically*. It does this by writing the grid code itself and then compiling it with Perl’s *eval* command. This means it is truly flexible, you can add or remove *any variable you like* to the grid with an *arbitrary spacing* and do not have to write any new code in the *binary_grid* module!
- Probabilities are calculated more efficiently and flexibly by the *distribution_functions* module.

7.1 Setup in your grid script

You need to do a few things:

1. Set up the grid in the `grid_options` hash by calling *grid_defaults*.
   Note that you *must* link a hash in your script to `$binary_grid::grid_options{results_hash}` e.g.
   `$binary_grid::grid_options{results_hash}=$results;` (if you have a global anonymous hash e.g. `$results={};`)
   or
   `$binary_grid::grid_options{results_hash}=\%results;` (if you have a global hash e.g. `%results;`)
2. Change parameters as you see fit.
3. Call `flexigrid(n);` where n is the number of threads
4. Output.

7.1.1 Grid variables: 1D example

Setting up which variables you use is best defined by example. Let’s start with M_1 which you will always require. First, define the grid resolution as \$n$\times$\$n$\times$\$n (here $10 \times 10 \times 10$)

```
$n=10;
```

Next, set up the variable number corresponding to the grid variable. Each grid variable must have a unique number, starting with 0 and working upward.

```
my $nvar=0;
```

Now use an *anonymous hash* in the grid option `{'flexigrid'}{'grid variable '.$nvar}` to give the options:

```
# Mass 1
my $mmin=0.1;
my $mmax=80.0;
$binary_grid::grid_options{'flexigrid'}{'grid variable '.$nvar++}=
{
    'name'=>'lnm1',
    'longname'=>'Primary_Mass',
    'range' => [$mmin,$mmax],
    'resolution'=>$n,
    'spacingfunc'=>"const(log($mmin),log($mmax),$n)",
    'precode'=>"my\$m1=exp(\$lnm1);my\$eccentricity=0.0;",
    'probdist'=>"ktg93(\$m1)*\$m1",
    'dphasevol'=>"\$dlnm1 "
};
```

(Note the ++ after `$nvar` which raises the variable number for the next variable.)

What does this mean?

`name` *string* This is the variable name. It should be parsable in Perl, e.g. in the above case it is expanded to `$lnm1` in the gridcode.

`longname` *string* This is a long (human-readable) name used in logging. It should contain no spaces (use underscores).

`range` *min max* This defines the range of the grid, from *min* to *max.* In the above case from `log(0.1)` to `log(80.0)`. (We are using a grid in *log* M_1.) Note that this is sent as an *anonymous array* of the form `[min,max]` (remember the square brackets!).

`resolution` *n* This is the grid resolution, given above by `$n`.

`spacingfunc` *func* This defines the spacing function which should be in the *spacing_functions* Perl module. In the above example the spacing is constant between `log(0.1)` and `log(80.0)` with `$n` steps.

`preloopcode` *code* This is code executed before the loop is set up.

`precode` *code* This is code executed before the next part of the grid is set up. In our case we have to specify `$m1` because we have only calculated `$lnm1`. We also set up the eccentricity because there is no grid for this and it is required.

`postcode` *code* This is code executed after the next part of the grid is set up. (In our case there is no *postcode*).

`probdist` *func* This is a probability distribution function as given in the *distribution_functions* Perl module. In the above we use the `ktg93` (Kroupa et al., 1993 IMF) function.

`dphasevol` *expression* This defines the contribution to the phase volume from this variable. In general this is just `$dlnm1` (i.e. should be `$dname`) but you can set it to whatever you like.

7.2 3D grid example

The next step is to set up the further grid variables. Usually these are M_2 and a, as defined in the code below.

```
# Binary stars:  Mass 2 and Separation
if($binary_grid::grid_options{'binary'})
{
   my $m2min=0.1;
   $binary_grid::grid_options{'flexigrid'}{'grid variable '.$nvar++}=
   {
      'name'=>'m2',
      'longname'=>'Secondary_mass',
      'range'=>[$m2min,'$m1'],
      'resolution'=>$n,
      'spacingfunc'=>"const($m2min,\$m1,$n)",
      'probdist'=>"const($m2min,\$m1)",
      'dphasevol','$dm2'
   };

   $binary_grid::grid_options{'flexigrid'}{'grid variable '.$nvar++}=
   {
      'name'=>'lnsep',
      'longname'=>'ln(Orbital_Separation)',
      'range'=>['log(3.0)','log(1e4)'],
      'resolution',$n,
      'spacingfunc',"const(log(3.0),log(1e4),$n)",
      'precode'=>'my $sep=exp($lnsep);my $per=calc_period_from_sep($m1,$m2,$sep);',
      'probdist'=>'const(log(3.0),log(1e4))',
      'dphasevol'=>'$dlnsep'
}
}
```

The above setup has the same general form as for M_1 but with different spacing functions (the *const* spacing function is used, for M_2 in the range 0.1 to M_1 and for ln a in the range 3 to 10^4).

7.3 Viewing The (Automatically Generated) Gridcode

When you run `flexigrid` in your grid script, a (flexi)grid with two threads will be constructed and then run via the Perl *eval* operator.

You can find the code that is constructed by the *binary_grid* module (and executed by your call to *flexigrid*) in the file `/tmp/gridcode`. View it with *less* or some other tool that can parse ANSI colours, or use the file `/tmp/gridcode.clean` instead. Note that the location of */tmp* can be overridden in *$binary_grid::grid_options{tmp}*.

7.4 Error handling

Things do go wrong. Usually when this happens, the threads will be shut down and your grid script will exit, telling you as much as it can about the error.

There is, however, a small chance that a *zombie binary_c* will still be running. You will notice it when it starts using 100% of your CPU power! Check on your system using an appropriate system-monitor (e.g. *top* on Unix) and kill the process (see *man kill*) if required.

7.5 Flexigrid global options

The flexigrid is set up inside a hash called, oddly, *%flexigrid*. Options that affect it are:

`grid type` *string* Usually this is “grid”, which is a normal grid. However, an (experimental) Monte Carlo grid is also available, in which case set it to “monte carlo”.

7.6 Flexigrid variable options

Note that any *code* is executed in the context of the flexigrid *eval* call, so you should prepend calls to subroutines with *main::* or *module_name::*

`condition` *code* The loop is executed only if *condition* is true.

`dphasevol` *expression* This defines the contribution to the phase volume from this variable.

`gridtype` *string* Most grid variables are of type “grid” which implies cell-centred variables. However, you can also specify “edge” to keep the variable on the cell edge instead.

`longname` *string* This is a long (human-readable) name used in logging. It should contain no spaces (use underscores).

`name` *string* This is the variable name. (It is best that it contains no special characters, just a-z, A-Z, 0-9 or _, but is converted internally into a hash key, so in theory it could be any string.)

noprobdist *n* If this is 1 then no probability is calculated. Useful for changing, e.g., the duplicity (see Recipe 13.4).

`preloopcode` *code* This is code executed before the loop is set up.

`precode` *code* This is code executed before the next part of the grid is set up.

`postcode` *code* This is code executed after the next part of the grid is set up. (In our case there is no *postcode*).

`postloopcode` *code* This is code executed after the loop and before the beginning of the next loop.

`probdist` *func* This is a probability distribution function as given in the *distribution_functions* Perl module.

`range` *min max* This defines the range of the grid variable, from *min* to *max.*

`resolution` *(n|code)* This is the grid variable resolution, which is *eval*ed in the gridcode context, so it can be a function or a constant. Mostly this is used for logging to estimate when the grid is finished.

`spacingfunc` *func* This defines the spacing function which should be in the *spacing_functions* Perl module.

7.7 Flexigrid loop structure

Each loop is logically structured as follows:

- setup `%fvar` (min, max, initial grid spacing δ from the *spacing_functions,* type=*grid* or *edge*)
- calculate the resolution (with an *eval* call)
- execute preloopcode
- check condition code
- { (start the loop)
-     execute precode
-     calculate this loop's contribution to the phase space and probability (using the *distribution_functions*)
-     update the counter
-     move loop variable on by $\delta/2$ (if it is is a grid variable)
-         `<next loop>` (or push commands onto the thread queue)
-     execute postcode
-     move loop variable on by $\delta/2$ (if it is is a grid variable) or δ (if it is an edge variable)
-     update grid spacing δ (using the *spacing_functions)*
- } (close the loop)
- execute postloopcode

Only the innermost loop contains the calls to push commands onto the thread queue.

7.8 Flexigrid and Condor

Warning: this is work in progress and subject to change!

Flexigrid now supports the Condor distributed computing engine. To use it, a few minor changes need to be made to your grid script. Please see `src/perl/scripts-flexigrid/grid-condor.pl` for an example.

- Add a line (near the top of your Perl script) to use the condor module:
  `use binary_grid::condor;`
- Replace the call to `flexigrid(...);` with `condor_grid();`
- At the beginning of your output() subroutine, add a line
  `return if($binary_grid::grid_options{condor_command} eq 'run_flexigrid');`
  (This suppresses output except for the final running job which has all the data)
- Put the following in your `defaults()` subroutine:
  `$binary_grid::grid_options{condor_dir}='some world readable directory';`
  This sets the working directory for this Condor job: it should be readable *from every Condor machine!* Usually this is an NFS mounted disk. I have not tried to make Condor work without a universally-writable directory.
- Every job you run should use a different output directory.

Condor issues and FAQ

(Some of these problems are AIfA specific, but may apply to you)

- What are the requirements for the output directory?
  The output directory must be in a world-writable location, e.g. an NFS mount (see the problem with `/export` below). Each time you run a grid script with a call to condor_grid, a new directory must be used. *It is up to you to ensure this is the case!*

- How are the jobs numbered?
  The jobs are numbered *x.y* (where *x* is the job number, *y* is the total number of jobs). These numbers just have to be unique, you are free to change them.

- What goes into the output directory?
  Each job's output (`stdout` and `stderr`) goes into the directories *stdout* and *stderr*.
  Each job also has a log which goes into the *log* directory.

- How do I put extra parameters in the Condor submit scripts?
  You can use a hash called `%binary_grid::grid_options{condor_options}` e.g.
  `$binary_grid::grid_options{condor_options}{Requirements}='Machine!="aibn73.astro.uni-bo`

- I have problems running my script with errors `/export/...` not found
  The problem here is the `/export` is local to your PC: you should run your script from `/vol/...` (a world-readable NFS mount point) instead. There is a function in the example condor scripts called `fix_aifa_environment()` ... try using it.

- Condor jobs are using all my CPU!
  They should be niced (nice -n 19) and preferably ioniced (ionice -c3) and given SCHED_IDLE priority (`chrt -i -p 0 <pid>`) as well. The *rob_misc* Perl module is used to do this automatically (with a call to `renice_me();`) There are issues with the *Linux* kernel and *cgroups* which are pending.

8 Initial distributions (IMF etc.)

The initial distributions of stellar masses, mass ratios, separations and periods are handled by the *distribution_functions* Perl module. The general idea is that given a set of stellar parameters M_1, M_2 and a (or P) and phase volume $\delta \ln M_1$, $\delta \ln M_2$ and $\delta \ln a$ (or $\delta \ln P$) the probability of existence of star i is calculated from

$$\begin{aligned} p_i &= \Psi(M_1)\Phi(M_2)\chi(a)\,\delta \ln M_1\,\delta \ln M_2\,\delta \ln a \\ &= \Psi\Phi\chi\delta \ln V \end{aligned} \tag{5}$$

where $\delta \ln V$ is the 'phasevol' (as stored in the progenitor hash, see Section A). The functions Ψ, Φ and χ are not necessarily independent (e.g. $\Phi(M_2)$ may actually be $\Phi(q = M_2/M_1)$ in which case it depends on M_1) but the assumption we make is that p_i is separable. Note that $\sum p_i$ must add to one, i.e. $\int p\, dV = 1$. In general, each of the distributions must also satisfy $\int \Psi(M_1)dM_1 = \int \Psi(M_2)dM_2 = \int \chi(a)da = 1$ where the integrals are from $-\infty$ to $+\infty$.

There are many forms for each initial distributions and they depend on which authors you believe and/or which stars you are looking at (e.g. low- or high-mass stars). There is no one true answer – yet!

8.1 Predefined functions in the distribution_functions module

const(a,b,x) Returns a constant distribution function between x and y i.e. returns $1/(b-a)$. x is optional, if given then a result is returned only if $a \geqslant x \geqslant b$, otherwise zero. (This is useful for specifying a flat-q distribution for the secondary mass or a flat-ln a separation distribution.)

powerlaw(a,b,k,x) A power-law distribution in x between a and b with slope k, zero if out of range.

three_part_power_law(x,x_0,x_1,x_2,x_{max},p_1,p_2,p_3,`$consts`) A three-part power-law distrubition between x_0 and x_{max}, with slopes p_i between x_{i-1} and x_i ($i = 1, 2, 3$), zero otherwise.

ktg93(M) A wrapper function to use the Kroupa et al. (1993) initial mass function for mass M. All the constants are set up for you, you just have to specify the mass M.

Kroupa2001(M) A wrapper function to use the Kroupa (2001) mass function, similar in design to *ktg93* above.

gaussian(x,μ,σ,x_{min},x_{max}) A Gaussian distribution, mean μ, variance σ, between x_{min} and x_{max} evaluated at x.

9 Grid spacings

The *spacing_functions* module is used to set up the grid spacings. At the moment there are only a few functions, but you can add your own of course.

const(x_{min},x_{max},n) Constant spacing between x_{min} and x_{max} with n steps, i.e. spacing $dx = (x_{\text{max}} - x_{\text{min}})/n$.

number(x) Simply returns x, i.e. spacing $dx = x$.

const_dt(...) Sets up a grid in primary mass that has constant spacing in stellar lifetime, i.e. time. It does this by constructing a table of stellar lifetimes as a function of mass and inverting this table. See Sec. 13.3 for an example of its use.

10 The Thread Model

CPUs used to be simple devices that did one job at a time. No more: every CPU these days has multiple *cores* which can run simultaneously while sharing system resources (e.g. RAM, disk). Binary_grid takes advantage of this by splitting the workload of running stars over a parameter space into multiple *threads* which run concurrently, thus speeding up execution of the grid.

When you call *flexigrid* it takes a single argument: the number of child threads. Let's say you launch *flexigrid(4);* then the parent thread launches four children which are managed in a thread queue (using Perl's Thread::Queue module). Each star you want to run is placed in the queue and the threads pull them off one by one until the parameter space is complete.

10.1 Thread resources

Each thread stores its results in a *hash table* (marked by % in Perl, see http://en.wikipedia.org/wiki/Hash_function#Hash_tables and http://www.perltutorial.org/perl-hash.aspx). This is like an array, which is probably familiar to you from other programming languages e.g. *C*, *C++* and *FORTRAN*, but instead of using numbers (e.g. 0, 1, 2. . .) as indices, it uses *keys* instead which can be any scalar, i.e. strings or numbers. Hashes can be nested, e.g. as hashes of hashes or hashes of arrays etc. This concept is very powerful because it allows you to store the results of your population synthesis in a *single hash table.* Because hashes can be nested, a single table can store *everything* you can throw at it.

Each thread has its own hash, usually accessed (e.g. in *parse_bse*) through the variable $h (which is actually a *reference* to a hash, similar to a C pointer). You put your data in *$h* and it is propagated where it is required.

When the threads finish they are *joined* together – in this process, the thread's *$h* hash is added to a global hash (usually called *%results* or something similar): the mechanism to do this is automatically included in *binary_grid*, you do not usually have to do anything yourself. At the end of the grid, the *%results* hash contains the sum of the population synthesis over all the stars you have run.

10.2 CPU load and thread number

It is not obviously clear how many threads you should launch even if you know the number of CPUs in your system. If your *binary_c* process does a lot of work and outputs very little, then *binary_grid* will not have much processing to do and you may as well launch as many threads as you have CPUs. However, if binary_c outputs a lot of data (e.g. at every timestep) then binary_grid will have to work hard to keep up and you may need as much (or more!) CPU for binary_grid as binary_c.

The best option is to experiment with a limited (low-resolution) parameter space and gradually increase the number of stars while changing the number of threads. This is, in any case, good practice for debugging your newly programmed code.

10.3 Memory usage

Each thread stores its data in its own hash, so the more threads you have the more memory you will use. If you are not saving much data, this will not bother you, but obviously if you save a lot of high-resolution data you may start to use a lot of RAM.

You should bin your hash keys to save RAM: consider this example. If you save the luminosity at each timestep, you will end up with hashes with keys (e.g.) 1.0, 1.0001, 1.0002, 1.0003 etc. Very quickly – in seconds! – you will have more data than you or your RAM can handle. Instead, bin your data and use logarithms when you can. Binary_grid has a bin_data function for this, e.g. `$logL=bin_data(log10($L),0.1);` bins the logarithm of `$L` (e.g. the luminosity) to the nearest 0.1 dex.

Remember that if each thread uses N megabytes of RAM and you have n threads, the maximum RAM use is $\sim (n+1)N$ because, when the data is joined, the master thread contains a copy of the data as well. However, this memory is released because the thread-joining process is carried out only once at the end of the population synthesis and soon after you output your data and quit.

10.4 Thread variables

Each thread sets its own copy of the `%binary_grid::threadinfo` hash, containing information relevant only to the thread. Most of this is used internally but you have access to it (do not change it unless you know what you are doing!).

binary_c_pid binary_c/nucsyn process id associated with this thread

cmd Internal thread command number (used to label various stop conditions)

h Reference to the data hash filled by this thread (*$h* in *parse_bse*)

lastargs The last argument string which was fed to binary_c/nucsyn

runtime The total runtime of the thread (including Perl)

runcount The number of stars run by this thread

state Thread state variable: 0=finished, 1=running

t0 Start time of the thread (from `[gettimeofday]`)

thread_number The unique thread number (starts at zero)

thread_queue The thread queue object (see the CPAN module *Thread::Queue*)

tvb_fp File (pointer) to which output is sent if `$binary_grid::grid_options{tvb}` is true

tvb_repeat Thread logging control variable (prevents repeated lines)

tvb_last The previous thread log line

tvb_lasttime The time of the previous output to the thread log

thread_prev_alive Thread timer (from `time()`): this is the time the thread last registered as being alive

thread_prev_complaint Thread timer (from `time()`): this is the time the thread last registered a complaint about taking too long

11 Debugging options

This section provides a summary of debugging options and suggestions for ways to fix your code (or *binary_grid*!).

- `$binary_grid::grid_options{'vb'}` controls the amount of grid output to the screen, usually lines showing the number of stars evolved, the total number in the grid, the current time, M_1, M_2, a, P and Z, the probability, the total grid probability, % grid completion, estimated time until arrival (finish) i.e. *ETA*, *tpr* (time per run), *ETF* (estimated time at which it will finish) and memory usage. If set to 1 it shows output every second or every `$binary_grid::grid_options{'nmod'}` models. If `vb` is set to 2 then more output, showing the arguments sent to *binary_c/nucsyn*, will be dumped onto the screen for each model. This is only useful for small grids, for large grids it will overwhelm you. It is possible to set `vb=3` but this is experimental.

- `$binary_grid::grid_options{'tvb'}` controls individual thread logging.

- `$binary_grid::grid_options{'log_args'}` will output the arguments sent to *binary_c/nucsyn* in files in `/tmp/binary_c.thread_$n.lastargs` where `$n` is the thread number.

- `$binary_grid::grid_options{'args'}` stores a single string with the raw arguments that were passed into *binary_c/nucsyn*. This is useful if you want to do some debugging in the `parse_bse` function (which has access to this variable). Within a thread, this data is also accessible in `$threadinfo{lastargs}`.

- `$binary_grid::grid_options{'progenitor'}` stores the progenitor information in a single string, in the following order: duplicity, M_1, M_2, P, a, e, Z, $\mathcal{P}$ and $\delta \ln V$ (the '`logphasevol`'$=\delta \ln M_1 \delta \ln M_2 \delta a$) If the star is a single star (`$grid_options{binary}==0`) then M_2, P, a and e are omitted and $\delta \ln V = \delta \ln M$. This data is accessible from `parse_bse`.

- The nested hash
  `%binary_grid::grid_options{'progenitor_hash'}{...}`
  contains the same information as the progenitor string (above) in a more accessible form: the hash *keys* are binarity, M_1, M_2, P, a, e, Z, $\mathcal{P}$ and $\delta \ln V$, and the values contain the appropriate data. Again, this is accessible from `parse_bse`.

- Use the `timeout` feature if your code is freezing. This is *not* the same as fixing your problem, but you may be able to stop the code from using all your CPU when it freezes! (Note that this uses Unix signals to test for timeouts: it may not work e.g. with the Condor queueing system).

12 Future Plans

- A graphical frontend would be good! Please write one for me.

13 Cookbook

This section contains some examples of how to use the *flexigrid*.

13.1 Star Formation History

You may not want to use a constant star formation rate (SFR). If you can live with all your models having the same input physics (i.e. metallicity, etc.) you can easily fold in a star formation history (SFH).

1. Make sure you know the *time* at each output of *binary_c*, let's call this `$t`.
2. You also need $\mathrm{dt} \times \mathrm{p}$, i.e. `$dtp`, at each timestep.
3. Weight `$dtp` by a SFR function:
   `$dtp *= SFR($binary_grid::bse_options{'max_evolution_time'}-$t);`

You have to write the SFR function yourself, e.g.

```
sub SFR
{
     # SFR as a function of Galactic age in Myr
     my $t=$_[0]; # Galactic age
     return 1.0; # const SFR
     return $t<1.0e3 ?  1.0 :  0.0; # const for the first Gyr
     return exp(-$t/10e3); # exponential dropoff over 10Gyr
}
```

How does this work? What you're passing into the SFR function is $t_{\mathrm{max}} - t$, which is effectively the time at which the star was born, assuming everything begin t_{max} ago.

This does not describe the overall normalization, e.g. if you want N stars in total. You have to add up the statistic appropriate to N (i.e. the number of stars seen *now*) and apply this to your results.

13.2 Aliasing

You output your histogram but it's all spiky and noisy! What's wrong? The simple answer is that you don't have enough stars on your grid for the bin width of your histogram. (See e.g. http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem) Solutions are:

1. More stars
2. Wider bins

Now, I hear you cry that your old Monte Carlo code gives a lovely smooth histogram. Yes, for the same number of stars, it might. But it might also give you worse spikes. With a MC code you can never guarantee resolution, and in the limit of a large number of stars the result is the same. At least with a grid-based solution you can *see* the spikes so you *know* the limit of your resolution. With an MC code you're just playing dice (see the FAQ 14.1).

Usually I just run with more stars. See also Recipe 13.1 above.

13.3 Time-resolved mass grid

Some problems require a more carefully spaced grid than the simple $\ln M$ grid. A good example is the calculation of stellar yields. The yield sets are generally required to give yields out to a time of many Gyr and with a time resolution of, say, 10 Myr. At early times this is not a problem, but at late times the stellar lifetime scales with the mass as $t \sim M^3$ (roughly) so in order to have good time resolution we require $\delta M \sim \delta t/(3M^2) \sim t^{-2/3}\delta t$ which is rather small at late times when t is large.

We could just set a normal log-mass grid and have a very large resolution. A smarter alternative is to set up the grid to *enforce* $\delta M = f \times \delta t/(3M^2)$ as our grid spacing (with a factor $f < 1$ which ensures over-resolution to avoid gridding artifacts). Implementations of this, and a fixed $\delta \ln t$ grid, are available in *binary_grid* via the *spacing_functions* module.

In the HRD project with Peter Anders, I use the following:

```
# Mass 1
$binary_grid::grid_options{'flexigrid'}{'grid variable '.$nvar++}=
{
      'name'=> 'lnm1',
      'longname'=>'Primary mass',
      'range'=>["log(0.1)","log(80)"],
      # const_dt spacing function options
      'preloopcode'=>"
            my \$const_dt_opts=\{
            max_evolution_time=>20000,
            stellar_lifetime_table_nm=>100,
            nthreads=>1,
            thread_sleep=>1,
            mmin=>0.1,
            mmax=>80.0,
            time_adaptive_mass_grid_log10_time=>1,
            time_adaptive_mass_grid_log10_step=>0.05,
            time_adaptive_mass_grid_step=>100,
            extra_flash_resolution=>0,
            time_adaptive_mass_grid_nlow_mass_stars=>10,
            debugging_output_directory=>undef,#'/tmp/adaptive_mass_grid',
            max_delta_m=>2.0,
            savegrid=>undef,
            vb=>0,
      \};
      spacing_functions::const_dt(\$const_dt_opts,'reset');",
      # use const_dt function
      'spacingfunc'=>"const_dt(\$const_dt_opts,'next')",
      # and its resolution
      'resolution'=>"spacing_functions::const_dt(\$const_dt_opts,'resolution');",
      'precode'=>'$m1=exp($lnm1); $eccentricity=0.0;',
      'probdist'=>"Kroupa2001(\$m1)*\$m1",
      'dphasevol'=>'$dlnm1',
};
```

The key line is `time_adaptive_mass_grid_log10_step=>0.05` which specifies that I want ($\log_{10}$) time resolved to every 0.05 dex. Note how I set up the `$const_dt_opts anonymous` hash of options and then send it via the spacing function with the option `'reset'`. This sets up a list of masses which are just returned for each call to `const_dt()` with the `'next'` parameter. Calling `const_dt()` with the `'resolution'` parameter just returns the number of masses in the list, which is trivially the resolution.

13.4 Single and binary stars combined

You can set up a grid containing both single and binary stars quite easily. The following code defines a grid variable "*duplicity*" which is either 0 or 1, and sets the grid_option "binary" appropriately. Deeper nested grid variables can then depend on the duplicity and choose, through the *condition* variable, whether to execute grids over M_2 and beyond.

```
# duplicity
$binary_grid::grid_options{'flexigrid'}{'grid variable '.$nvar++}=
{
     'name'=>'duplicity',
     'longname'=>'Duplicity',
     'range'=>[0,1],
     'resolution'=>1,
     'spacingfunc'=>'number(1.0)',
     'precode'=>'$binary_grid::grid_options{binary}=$duplicity;',
     'gridtype'=>'edge',
     'noprobdist'=>1,
};
```

13.5 Snapshots: I want to stop a grid and restart later

The ability to stop and restart the grid is called *snapshotting*. Note that while this currently works, it assumes that you:

1. Use the identical grid script to restart
2. Do not change any of your installation (*binary_c*, *binary_grid*, etc.) between stopping and restarting,

thus snapshotting is quite limited in its functionality but it might save your data when your computer is about to die.

13.5.1 Suspending the grid

Binary_grid looks in files, defined in `$binary_grid::grid_options{'suspend_files'}=[...]`, and if one exists, it executes its snapshot code. To make one exist, use the Unix command *touch* e.g.

```
touch /tmp/force_binary_c_suspend
```

This joins all the current threads and saves their data in one file so you can restart it later. It may take some time to join all the threads, so please be patient. By default, binary_grid looks at the files /tmp/force_binary_c_suspend and `./force_binary_c_suspend` (although you can add your own with `push(@$binary_grid::grid_options{'suspend_files'}, "new filename");`)

The saved file name is defined in `$binary_grid::grid_options{'snapshot_file'}` (default `'/tmp/binary_c-snapshot'`).

All the above /tmp/ are actually `$binary_grid::grid_options{tmp}` so you can define your own location for all the files (which is probably more secure, because then you can prevent anyone from touching the file and stopping your grid).

13.5.2 Restarting the grid

If `$binary_grid::grid_options{starting_snapshot_file}` is defined, it is used to load in a previously saved snapshot and the grid is restarted. You can usually just load this on the command line, e.g. by running your grid script as

```
./src/perl/scripts/my_grid_script.pl starting_snapshot_file=my_snapshot_file
```

13.6 World Domination

I'm working on it.

14 FAQ

Frequently asked questions.

14.1 Why not Monte Carlo?

The alternative to running a grid is a Monte Carlo method, where you throw systems in according to some initial distribution and a random number generator. In the high number limit this should give the *same result* as a grid. However, in the low-number limit, the results are probably going to be different, and this is when you test your code. The big advantage of a grid is that you have a good handle on *errors* due to the finite resolution of your sampling of the initial distribution(s) without the smearing out of a Monte Carlo approach. Furthermore, MC may accidentally miss part of the initial parameter space (it is, after all, random) which you know is covered by a grid approach (at least to within a known error e.g. $\delta \ln M_1$). You are also guaranteed, on a grid, to sample even the rare systems which would be sparsely populated (if at all) in an MC simulation. Such systems will have a small probability per star, p_i, but on the other hand these might be the systems of particular interest to you!

14.2 Why MC SN kicks?

Traditionally supernova (and white dwarf) kicks have had their velocity chosen in a Monte-Carlo way, rather than on a grid. This was left as-is because there are four dimensions for each kick, which – given a coarse grid of $10 \times 10 \times 10 \times 10$ – means the parameter space expands by a factor of 1, 000 for stars with kicks. The runtime increase is not worth the effort as in order to finish anything in your lifetime you would have to either buy a supercomputer or run such a coarse grid in the original parameters that the whole exercise becomes pointless. The brute force resolution test is what you need here: just keep increasing the resolution until the numbers converge. It helps if you are clever about it: if you're interested in supernovae only then M_1 is probably the dimension that requires the most resolution, or perhaps M_2, but probably not a.

14.3 Zombie binary_c processes

When the grid exits abnormally, or is killed, a *binary_c/nucsyn* process may be left behind in an infinite loop state and as such will take up your precious CPU time. You will have to kill it manually *if you are sure it is not doing anything with another grid process!* (Be careful and do *not* do "killall binary_c" as root. . .)

14.4 Zombie Perl processes

These should die naturally when the grid finishes and the Perl script exits. They are harmless and have never been seen to consume CPU time.

14.5 Setting functions and/or function pointers

In the good old days you had to set up function pointers, e.g. `\&parse_bse`, manually when setting up your grid. However, in the latest version of *flexigrid*, this is set automatically (from `\&main::parse_bse`). However, you can override the settings.

As an example, if you want to set the parse_bse_function_pointer, just set
`$binary_grid::grid_options{parse_bse_function}='my_parse_bse';`
where `my_parse_bse` is your new function name. This will automatically be converted into `\&main::my_parse_bs`
at runtime.

This process is applied to any `grid_option` whose key ends in `_function`. These include `thread_precreate_`
`threads_entry_function`, `threads_flush_function`, `thread_postrun_function`, `thread_prejoin_functi`

threads_join_function, thread_postjoin_function. These are all undef by default, so are ignored (unless you override them with function names), except threads_join_function which is set to binary_grid::join_flexigrid_thread as usual.

15 *Installation*

This section describes in some detail the installation of binary_c and binary_grid. Note that sometimes you will require the latest version of pieces of software, in particular *Perl*. I show you how to do this *without* requiring root permissions on your machine. Should you require packages to be installed that require root permission, you can always set up binary_c and binary_grid on a *virtual machine* which runs as a guest on your operating system, I do this with *Virtualbox* (https://www.virtualbox.org/). This would be my advice if you are running e.g. *Micro$oft Windows*. You will have the root password for your virtual machine, so there are no permissions problems.

You will need some basic tools installed to make everything work: *bash*, *subversion*, *perl*. These are available on all good operating systems, and come by default – or after a simple install – with most flavours of Linux/Unix.

15.1 *SVN access*

By using *binary_c* and *binary_grid* you are part of a community. You are therefore responsible for helping with the never-ending process of improvement and bug fixing.

Binary_c (and its *Perl* modules, such as *binary_grid*) is stored on an SVN server, currently www.astro.uni-bonn.de.
To access the server, you require a username and password, as well as *subversion* (SVN) installed on your computer. In general, you can make up your user name (provided someone else hasn't already taken it). To obtain a password run the command
```
htpasswd -mn <userid>
```
where you substitute `<userid>` for your chosen user name.

Once you can log in, you can either

1. Download the trunk from
   http://www.astro.uni-bonn.de/svn/izzard/binary_c/trunk
   with
   ```
   svn co http://www.astro.uni-bonn.de/svn/izzard/binary_c/trunk
   ```
2. Download your personal branch from
   http://www.astro.uni-bonn.de/svn/izzard/binary_c/branches/username
   with
   ```
   svn co http://www.astro.uni-bonn.de/svn/izzard/binary_c/branches/username
   ```

If you just use the `trunk`, you should not make changes to *binary_c* or *binary_grid*. You are more of a *user* rather than a *developer.* If you want to make many changes, you should obtain your own `branch` instead.

15.2 *Installing* binary_c

Installing *binary_c* is as simple as

1. Go to the `binary_c` directory
2. Run
   ```
   ./configure && make cleanall && make
   ```
3. Test that it worked by running
   ```
   tbse
   ```

binary_c has its own manual, you should refer to that for more details.

15.3 *Installing* binary_grid

binary_grid is all written in Perl (http://www.perl.org/) which is available on almost every modern operating system, although to my knowledge *binary_grid* has never been tested on anything other than Linux (http://www.linux.org/), Solaris (not for some years: does Solaris still exist?!) and MacOS (which is really BSD Unix).

The newest binary_grid uses features from the latest Perl (5.16 or above, currently testing on 5.17.5) so you'll clearly need this installed. Unfortunately most versions of Linux run an older Perl (e.g. Ubuntu 11 is running 5.10). You will also need to have the latest versions of a number of Perl modules (which come from CPAN http://www.cpan.org/). My recommendation is to use *perlbrew* to make your own Perl, *local::lib* to install the modules in your home directory correctly, and *cpanminus* (`cpanm`) to install the modules.

You can find *perlbrew* at http://perlbrew.pl/ : please follow the instructions on that page to install the latest Perl on your system. Typically, do the following, but remember you *must* install *perlbrew* with "`-Dusethreads`" otherwise *Perl* will not use threads and *flexigrid* will fail:

1. `wget -no-check-certificate -O - http://install.perlbrew.pl | bash`
   or
   `curl -kL http://install.perlbrew.pl | bash`

2. Run `perlbrew available` to find a list of available versions, you should choose the newest, e.g. `perl-5.18.0`, and then run the following command to install *Perl*:

   ```
   perlbrew -v install perl-5.18.0 -Dusethreads -Duselargefiles
   -Dcccdlflags=-fPIC -Dpager=/usr/bin/sensible-pager -Doptimize="-O3
   -march=native -mtune=native" -Duseshrplib -j 8
   ```

   Note that the final *8* should be replaced by the number of CPUs you wish to use for the build (it is an option passed to *make*).
   The installation process can take a long time, go and have lunch... then:

3. Do what *perlbrew* suggests with your `.bashrc` (or whatever shell initialization script you use) to fix the `PERLBREW_PATH`.

4. Restart your shell (e.g. close your terminal and open a new one, or just run `bash` again) to update your environment.

5. Check you're using the correct Perl with
   `perl -v`
   This should say something like
   ```
   This is perl 5, version 18, subversion 0 (v5.18.0) built for x86_64-linux-thread-multi
   ```
   Note that the `-thread-` is there – without threads, *binary_grid* will not function.

   If you have a previous version of Perl installed by perlbrew, you will have to do something like
   `perlbrew switch perl-5.18.0`

6. If the installation was successful, but you see an older Perl, check that your `$PATH` variable points to the new Perl, and check that `$PERLBREW_PATH` exists.

7. Install *cpanminus* by running
   `perlbrew install-cpanm`

Now you have the latest *Perl* and *cpanm* installed, you can start to install the modules needed for *binary_grid*. (Un?)fortunately, there are many of them, so I have made a script to do it for you

1. From the *binary_c* directory, go to `src/perl` with
   `cd src/perl`
2. Run the install script
   `./install_modules.pl`
3. Wait.
4. Check the output. If there is a failure, it should say. You will have to fix it by looking at the output to see where it went wrong. The most common cause of failure is that you need some kind of development (*-dev* or *-devel*) packages to be installed on your system, e.g. through *apt*, *yum* or *synaptic*. This may require root permission.
5. Your installed modules are usually in (for *perl* 5.18.0)
   `$HOME/perl5/perlbrew/perls/5.18.0-threads/lib/site_perl/5.18.0`
   which should be found automatically by the *perl* installed by *cpanm/perlbrew*. However, if you have problems not finding modules after they have been installed, check that either the `PERL5LIB` environment variable is empty or points to the above directory. Everything *should* work with `PERL5LIB` empty, but all Linuxes are different. (Note that you may already have PERL5LIB pointing to a custom *Perl* modules directory: this should be fine as long as the modules in it are up to date. It is recommended that you let cpanm installed all your *Perl* modules.)

Now you should have everything installed and be able to run a *binary_grid* script.

Troubleshooting

Module not found Check the `PERL5LIB` environment variable. When it is empty, *cpanm/perlbrew* should find your modules automatically, *if* they are installed correctly by *cpanm* (in the *site_perl* directory). You can try installing modules again with `cpanm -reinstall` to force a reinstallation.

Module fails to build Try `cpanm -notests` to not run all tests. Sometimes these fail for spurious reasons, e.g. in the case of *binary_grid* because there are *no tests,* but the module still functions.

Perlbrew build flags I would recommend at least
`-Dusethreads -Duselargefiles -Doptimize="-O3 -march=native -mtune=native"`
although you may wish to use `-O2` if you feel mathematical precision is key to your application (I have never noticed a problem with the above).
I also recommend `-Dcccdlflags=-fPIC` so that modules are position independent and `-Duseshrplib` so that the appropriate threading library is used. (These flags are passed to the C compiler that builds *Perl*.)

Rebuild modules for new Perl Let's say you had an old *perl* built with perlbrew, have installed a shiny new *perl* version with perlbrew, and now you want to rebuild all your existing modules. Try this after the install, but before switching to the new *perl*, replacing `5.xx.0` with the new *perl* version:
`perlbrew list-modules | perlbrew exec -with 5.xx.0 cpanm`

A Grid options (grid_options hash)

The following summarises the `%grid_options` hash elements. Please note that some options are (perhaps) no longer used and are left in this manual as a reference guide only.

`alarm_procedure` When a signal is captured (usually on a timeout) the alarm_procedure tells us what to do. If 0 then the grid exits. If 1 it tries to restart on the next star (although this may be buggy it is the default!). Really you should fix the problem if there is a timeout because it should not happen.

`always_reopen_arg_files` If 1 then argument log files are always reopened when new arguments are sent to binary_c. Can be very expensive in I/O operations, i.e. slow.

`always flush binary_err` Set to 1 (default) to force the flushing of the `stderr` channel out of *binary_c*. If you don't do this, stderr output will pile up and eventually stop the grid from functioning. Do not change it unless you *know* there will be no `stderr` output from *binary_c*.

`arg_checking` Sometimes, an argument to binary_c will not be recognised (e.g. if you set something in a bse_option which is not compatible with your binary_c, perhaps because you haven't enabled some feature in binary_c) and the grid will give errors (and/or crash) which are rather cryptic. Instead of this, you can set arg_checking to 1 to have strict checking of each argument that is passed to binary_c. This is, however, rather slow (because it requires a lot of I/O) so the default is 0.

`args` Not technically an option, but is the string of arguments which is passed to *binary_c/nucsyn*. Useful for logging or debugging.

`binary` The duplicity. If binary is 1 then the grid runs binary stars, if 0 then single stars.

`cache_binary_c_output` If this is 1 (the default) then output from binary_c is cached in a local array before being processed through `tbse_line()` calls. This should be faster, because it means less switching from one process to another, but its use means that *binary_c* is out of sync with `tbse_line()`, so debugging is more difficult.

`colour` If `colour=1` then verbose output (see `vb`) is in colour, using the `Term::ANSIColor` Perl module. This is recommended.

`disable_signal` You can disable a specific signal with this hash, e.g. to disable `INT` signals, set `$binary_grid::grid_options{disable_signal}{INT}=1;`

`exit_on_eval_failure` When each star is run, it is inside a *Perl* `eval` construct. If there is an error (i.e. `$@` is not undef) *and* `exit_on_eval_failure` is defined then the grid is stopped. Otherwise, the grid goes on in the hope that the error does not persist (possibly dubious!).

`force_local_hdd_use` Checks to see if the disks to which you are outputting data are local or remotely mounted. If the latter, binary_grid refuses to work. You should (of course) always use local disks for your output – it is much faster. (Defaults to 1 on Unix systems.)

`flexigrid` Hash used to set up the flexigrid, see Section 7.

`grid_defaults_set` Logic control variable, set to 1 if grid_defaults has been called.

`lastargs` Very similar to log_args but at a slightly different place in the code. Should be avoided, use `log_args` instead.

`libpath` Path to libraries to be included in the calling of the *binary_c/nucsyn* executable. This harks back to the days whens *binary_c/nucsyn* was built as a set of shared libraries. These days it is built statically, so libpath is usually an empty string (and hence is ignored).

`log_args` If this is set to 1 then the arguments passed to *binary_c/nucsyn* are saved in a file in `/tmp/` (usually `/tmp/binary_c-args` or similar, with separate files for each thread). This is very useful when the grid freezes as you can immediately see which stellar system has caused the problem and then rerun it manually.

`maxq_per_thread` The maximum number of stars allowed to sit on the thread queue, per thread. You do not want the thread queue to become indefinitely large because when that happens the resources to store the queue (RAM) will become significant. Default is 10.

`newline` If `vb=1` then the value of this option is used as the end of line character. You can use either "`\x0d`" (just a carriage return) or "`\n`" which is the carriage return with a line feed (the usual concept of "new line").

`nice` A string containing the nice command used to run *binary_c/nucsyn*. Usually no nice is used, so this is '`nice -n +0`'. You can leave this as it is and run '`nice grid.pl`' if you want the whole grid to be "niced".

`no_signals` Set to 1 to disable Perl signals. Useful for working with Condor.

`operating_system` Saves details of the current operating system (from *rob_misc.pm*)

`parse_bse_function` Used to create the `parse_bse_function_pointer` (see FAQ 14.5).

`parse_bse_function_pointer` In your grid script you need to define a subroutine (also known as a function) to be used for data parsing. Usually this is called `parse_bse` (see FAQ 14.5).

`prog` The *binary_c/nucsyn* executable name, usually '`binary_c`'. See also `rootpath`.

`progenitor` Not really a grid option, but stores the system information (M or M_1, M_2, a, P, e, p etc.) – useful for debugging or logging.

`progenitor_hash` As progenitor but stores each item in a hash.

`repeat` The number of times *`binary_c`* is called for each grid point, default is 1. This is useful for increasing the resolution in situations where Monte-Carlo methods are used such as the random supernova or white dwarf kicks.

`rootpath` The directory in which the *binary_c/nucsyn* code source and executable reside. See also `prog` and `srcpath`.

`single_star_period` The orbital period (days) given to single stars (default 10^{50} d).

`starting_snapshot_file` Filename of a snapshot to be loaded before the grid is started.

`snapshot_file` Filename used to save the status of the grid in a snapshot so it can be restarted. Defaults to `/tmp/binary_c-snapshot` (where /tmp/ is set by the '`tmp`' grid option).

`srcpath` Location of the binary_c source files. Usually this is just `rootpath` with '`/src`' appended.

`suspend_files` An anonymous array (set with `[...]`) containing filenames that should be watched. If one exists, the grid is suspended to disk using the snapshotting mechanism (see snapshot_file).

`tmp` Location of temporary files. Usually this is */tmp* (on Unix/Linux) and certainly it should be a local disk. If you have multiple users on one PC, they cannot *all* use */tmp*.

`thread_max_freeze_time_before_warning` The maximum time a thread loops its calls to `tbse()` before a warning is issued (seconds, default 10).

`thread_precreate_function` Used to create the `thread_precreate_function_pointer` (see FAQ 14.5).

`thread_precreate_function_pointer` A pointer to a function to be called just before `threads->create` is used to start a thread. Ignored if `undef`.

`thread_prejoin_function` Used to create the `thread_prejoin_function_pointer` (see FAQ 14.5).

`thread_prejoin_function_pointer` A pointer to a function called just before a thread is `joined`. Ignored if `undef`.

`thread_presleep` A thread waits for this length of time (in seconds) before it starts in order that all other threads have time to start before any output is logged to the screen. As such this is just to keep output pretty and should be small (default 1 second).

`thread_postjoin_function` Used to create the `thread_postjoin_function_pointer` (see FAQ 14.5). Ignored if `undef`.

`thread_postjoin_function_pointer` A pointer to a function called just after a thread is `joined`. Ignored if `undef`.

`thread_postrun_function` Used to create the `thread_postrun_function_pointer` (see FAQ 14.5). Ignored if `undef`.

`thread_postrun_function_pointer` A pointer to a function to be called just after a `thread->create` call (and just after `$@` is checked for a thread-creation error). Ignored if `undef`.

`threads_entry_function` Used to create the `threads_entry_function_pointer` (see FAQ 14.5). Ignored if `undef`.

`threads_entry_function_pointer` A pointer to a function called from inside a thread just before flexigrid() is called. Ignored if `undef`.

`threads_flush_function` Used to create the `threads_flush_function_pointer` (see FAQ 14.5). Ignored if `undef`.

`threads_flush_function_pointer` A pointer to a function called from inside a thread just after flexigrid() returns. Ignored if `undef`.

`threads_join_function` Used to create the `threads_join_function_pointer` (see FAQ 14.5). Ignored if `undef`.

`threads_join_function_pointer` When using a threaded grid you have to define a subroutine (also known as a function) pointer which points to the function to be called after each thread have finished. This function is responsible for "joining" the threads, which means it collects the results from the thread and adds them to the global results. This is usually a function called `join_thread` and is set with a line in grid-xxx.pl similar to:

```
$binary_grid::grid_options{'threads_join_function_pointer'}=\&join_thread;
```

`threads_stack_size` The stack size (in MBytes) for each of the Perl threads. Default is 32 (MBytes).

`timeout` When possible the grid sets an alarm (a Linux/UNIX signal) which means that warnings are given and perhaps the grid stopped after `timeout` seconds. Set to zero to ignore, usually 30 is enough. This assumes your *binary_c/nucsyn* process takes less than `timeout` seconds to run a system so if you have problems with this please check your system to make sure you have the CPU time you need.

`vb` Verbosity level. With `vb=1` you get some output about the grid, i.e. the number of stars run, the number to go, the current M, or M_1, M_2 and a, an estimate of the time taken, time remaining and the time at which the grid is expected to finish. You may also get information about memory usage (if available). Make sure your terminal window is quite wide to accommodate the information! If `vb=2` you will get additional information about each system. This is very useful for determining which stellar system is causing the code to freeze and/or crash. See also `nmod`, `colour` and `log_args`. Some verbose logging is only switched on if `vb=3`, although this should be considered experimental.

`weight` The probability is weighted by this value.

B *Binary_c/nucsyn options (*bse_options *hash)*

The most important options are listed here, however you should consult *binary_c* for the main options list because these options are (mostly) just passed to *binary_c*. The full list of options is given in `src/setup/parse_arguments.c`. *BSE* refers to Hurley et al. (2002).

acc2 Bondi-Hoyle accretion rate parameter (default 1.5)

alpha_ce Common envelope ejection efficiency, default 1.0.

bb CRAP parameter, default 0.

BH_prescription Black hole mass as a function of CO core mass prescription. Default 0 (*BSE*), can be 1 (Belczynski).

delta_mcmin Shift in the core mass used to calculate the third dredge up efficiency (default 0).

eddfac Eddington limit multiplier (default 1e6, i.e. no Eddington limit).

extra A string which is appended to the arguments for *binary_c*. You should not use this form if you can avoid it, but it might be useful in some cases.

lambda_ce Common envelope envelope binding energy parameter (default -1, i.e. set according to fitting functions).

lambda_ionisation Amount of recombination energy used to eject the common envelope (default 0.0, only acts if lambda_ce (above) is -1).

lambda_min Minimum value of the third dredge up efficiency λ (default 0).

max_evolution_time Maximum time for stellar evolution, default 13.7 Gyr.

minimum_envelope_mass_for_third_dredgeup Minimum envelope mass for third dredge up (default 0.5).

qcrit_GB_method Method for choosing q_{crit} on the giant branch.

sn_sigma SN kick velocity dispersion.

superwind_mira_switchon Mira period at which the superwind switches on in TPAGB stars, default is 500 days as in Vassiliadis and Wood (1993).

tidal_strength_factor Modules the tidal timescale, default 1.0.

tpagbwind TPAGB wind prescription. Default 0 is Karakas et al. (2002).

tpagb_reimers_eta η for Reimers on the TPAGB.

vw_mira_shift Shifts the superwind Mira switchon (default 0.0).

vw_multipler Multiplier for the VW93 wind (default 0.0).

wr_wind Wolf-Rayet (massive star) wind prescription, default 0 is *BSE*.

wr_wind_fac Multiplier for WR winds, default 1.0.

z The metallicity (default 0.02).

References

Hurley, J. R., Tout, C. A., and Pols, O. R. (2002). Evolution of binary stars and the effect of tides on binary populations. *MNRAS*, 329:897–928.

Izzard, R. G., Dray, L. M., Karakas, A. I., Lugaro, M., and Tout, C. A. (2006). Population nucleosynthesis in single and binary stars. I. Model. *A&A*, 460:565–572.

Izzard, R. G., Glebbeek, E., Stancliffe, R. J., and Pols, O. R. (2009). Population synthesis of binary carbon-enhanced metal-poor stars. *A&A*, 508:1359–1374.

Izzard, R. G., Tout, C. A., Karakas, A. I., and Pols, O. R. (2004). A New Synthetic Model for AGB Stars. *MNRAS*, 350:407–426.

Karakas, A. I., Lattanzio, J. C., and Pols, O. R. (2002). Parameterising the third dredge-up in asymptotic giant branch stars. *PASA*, 19:515–526.

Kroupa, P. (2001). On the variation of the initial mass function. *MNRAS*, 322:231–246.

Kroupa, P., Tout, C., and Gilmore, G. (1993). The distribution of low-mass stars in the Galactic disc. *MNRAS*, 262:545–587.

Vassiliadis, E. and Wood, P. R. (1993). Evolution of low- and intermediate-mass stars to the end of the asymptotic giant branch with mass loss. *ApJ*, 413:641–657.