BCMD modelling system 0.5b

Matthew Caldwell

January 16, 2015

Contents

1 Introduction

BCMD is intended to be a more portable and robust successor to the BRAINCIRC modelling environment (Banaji 2005). Like BRAINCIRC, its primary purpose is to translate a description of a model in terms of equations and reactions into executable code that can simulate the model's behaviour for specified initial conditions and time courses of parameter changes.

Ultimately, all supported models are converted into a standard differential-algebraic equation representation, of the form:

$$\mathbf{M}\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, \boldsymbol{\theta}, t) \tag{1}$$

where $\mathbf{y}$ is a vector of variables of interest, $\mathbf{M}$ is a constant, possibly-singular, mass matrix specifying relations among the differential terms, and $\mathbf{f}$ is some vector-valued function, possibly having additional parameters $\boldsymbol{\theta}$. (If a row of $\mathbf{M}$ is zero, the corresponding equation in $\mathbf{f}$ is algebraic rather than differential.)

The differential and algebraic equations may be specified explicitly, but it is often convenient to describe some or all of a system in terms of reacting chemical species in one or more spatial or functional compartments, as in this example from the BrainSignals model (Moroz 2013, Appendix C):

$$p_2\ \mathrm{H_m}^+ \xrightarrow{f_2} 4\mathrm{Cu_{A,o}} + 4\mathrm{cyta_{3,r}} \tag{2}$$

We attempt to make such specifications as concise and intuitive as possible, within the constraints of efficient parsing. The following code snippet declares both an explicit differential equation and the above chemical reaction in the BCMD modelling language:

```
# the BrainSignals "v_prel" differential equation
v_p' = 1/t_p*(P_a - v_p)

# the BrainSignals "a_ox" reaction
p2 [H, m] -> 4 [CuA_o] + 4 [cyta3_r] {f2}
```

The modelling language is described in detail in §5, and a number of complete models are supplied in the `examples` directory of the BCMD distribution.

In addition to the model equation system, a model may also include arbitrary intermediate variables and parameters. Default initial values may be specified for these values, or they may be set at any step of a simulation. Unlike BRAINCIRC, BCMD does not distinguish between different types of run: as detailed in §6, the time course of a simulation is just driven by its input file.

2 System requirements

BCMD runs on Linux, Mac OS X and probably other Unix-esque systems, and also on Microsoft Windows with the help of the MinGW development tools.

Models are translated to ANSI C code, so a suitable C compiler is needed to generate the model executable. Because the underlying RADAU5 DAE solver is written in Fortran, a Fortran compiler is also required to build it, and the C compiler must be capable of linking to the Fortran library. The GNU compiler toolchain (`gcc, gfortran, gmake`) is recommended, though other compilers might also work.

The included `configure` script should be able to identify the C and Fortran compilers and identify the appropriate compiler and linker options. The script is generated with `autoconfig`. If for some reason you need to recreate `configure`, the GNU `autotools` will be required; usually this should only be necessary for the package maintainers.

The model compiler is written in Python, and has been mostly tested with Python 2.7, although it may also work with Python 3.x and possibly with earlier versions. The compiler makes use of the PLY lexer/parser package (Beazley 2011); this is included in source code form, so no separate installation is necessary.

The compiler can optionally generate a graph of the model structure in the GraphViz DOT language. To convert this to an actual graphic image, you will need to have the GraphViz tools installed. Note that this is not necessary to be able to run models, but it can be a useful aid to analysing them.

Some additional resources are necessary in order to use the BGUI graphical interface and/or the batch processing utilities. BGUI makes use of the Tkinter windowing toolkit and the matplotlib plotting library, while both sets of tools require the NumPy numerical libraries. These packages may already be available in your Python system, but if not, the easiest way to obtain a complete installation of the whole SciPy stack is via the Enthought Canopy distribution—the free Canopy Express package includes all the required components.

To perform parameter optimisation, you need the OpenOpt framework—see the installation page for information about how to install this. (The preferred option is to use `easy_install`, but getting this to work with Enthought can be a little fiddly.) Ideally you should also add the PSwarm solver—download the PPSwarm_v1_5.zip code archive and build it with Python support enabled. This can be installed globally or placed directly into the BCMD distribution after you have installed the latter—put the built PSwarm python module (e.g. `pswarm_py.so`) into the BCMD `batch/pylib` directory.

In order to support export to (and perhaps in future import from) SBML, the libSBML library (Bornstein et al. 2008) is required, including its Python bindings. (Again, getting this to co-operate with Enthought can be a little fiddly.)

3 Installation

Before attempting to run BCMD, ensure that your environment is set up with all the necessary tools (see the instructions below for Linux, Mac OS X and Windows).

Obtain the code archive from the Computational Brain Modelling group's Software web page at UCL, and unpack it in a location of your choice. Open a terminal window (for Windows users this should be an MSYS window rather than the standard DOS prompt), switch to the top-level `bcmd` directory and type:

```
./configure
make
```

If everything is correctly set up, you should now have a working BCMD installation. If not, examine any error messages produced to identify what went wrong.

3.1 Linux

Most Linux installations will already be furnished with a suitable Python and the GNU C compiler. A Fortran compiler may or may not be included—if you are unsure, it is worth running `configure`

anyway to check. Otherwise, there will usually be a `gfortran` package available via the system software package manager. E.g., for Debian-based distributions like Ubuntu and Mint, use the command:

```
sudo apt-get install gfortran
```

while for RedHat/Fedora use:

```
sudo yum install gcc-gfortran
```

For other distributions, check your documentation on how to search the package repository, or have a look at the GNU wiki.

3.2 Mac OS X

If you don't already have the OS X developer tools installed, you will need to download Xcode from the Mac App Store. The current version at the time of writing runs on Mac OS X 10.8 (Mountain Lion) and 10.9 (Mavericks). Earlier versions can be obtained from Apple's Xcode site provided you have a (free) developer registration.

From within Xcode, install the command line tools by going to **Preferences** → **Downloads** → **Components** and clicking the small **Install** button alongside the **Command Line Tools** item.

Pre-built installers for `gfortran` are available from GNU. I *think* what they install will be sufficient for running BCMD, but have not tried them on a completely clean system. Otherwise, it is possible (if laborious) to build the entire toolchain with the MacPorts package system (`sudo port install gcc48` should be a good starting point). Some other suggestions can be found here.

The version of Python that ships with Mac OS X tends to be a bit behind the times, but is probably sufficient for running BCMD from the command line. To make use of the GUI, you should install a more up to date version of Python 2.x along with Tkinter and SciPy—the easiest way to do this is almost certainly to use Enthought Canopy Express.

3.3 Microsoft Windows

Install an up to date Windows version of Python from python.org (the latest Python 2.x is a safer bet than Python 3), or use the Enthought Canopy Express distribution, which also includes the necessary SciPy packages.

Install the MinGW Windows port of the GNU development tools, including the MSYS shell environment. The simplest way to do this is first to install the mingw-get installer tool, and then use its GUI to install the relevant packages:

- mingw-developer-toolkit
- mingw32-base
- mingw32-gcc-fortran
- msys-base

(The 32-bit versions appear to be the only ones readily available, but they also seem to work fine on 64-bit Windows. I haven't tested this very extensively, though, so please let me know if you encounter any difficulties.)

Some additional configuration steps are needed after installing these tools.

MSYS creates a proxy Unix filesystem within its own install directory. Locate this directory—in my tests it was:

```
C:\MinGW\msys\1.0
```

At the 'top' level here (i.e., immediately within that directory) is a Windows batch file, `msys.bat`, that launches a Bourne-style shell window. This will be your primary medium for dealing with BCMD (at least until some prettier front end gets developed), and is in any case quite a useful thing to have at your fingertips, so you might want to make a shortcut to it somewhere readily accessible, like the desktop.

Double-click `msys.bat`. The shell will start up in a proxy home directory for your user—this will be something along the lines of:

```
/home/Owner
```

in the ersatz file system of the shell, corresponding to something like:

```
C:\MinGW\msys\1.0\home\Owner
```

in the actual Windows file system. Make a note of this directory location. (The full Windows directory structure, above `C:\MinGW\msys\1.0`, is accessible within the MSYS shell using file paths of the form `/c/WINDOWS/`, i.e. with the drive letter, in lowercase, as the first directory in the constructed Unix path.)

MSYS provides a script to help synchronise with the MinGW installation it cohabits with. Type the following in the shell window, and follow the onscreen instructions:

```
/postinstall/pi.sh
```

The MSYS shell initially inherits a translated version of the Windows PATH environment variable, which typically contains all sorts of things we're not interested in, while also lacking some things we are. In particular, we need the Python that was installed earlier to be on the path, and we'd also prefer there not to be anything with spaces in, since that can confound MinGW.

The `pi.sh` script should have set up a mount point for the MinGW installation, and added its `bin` directory to its PATH. In the terminal, type:

```
echo $PATH
```

And look for something like `/mingw/bin` in the result. Make a note of what it is, and also of where your Python installation went. Now create a text file like the following, but with the paths adjusted to match the locations of your MinGW and Python:

```
PATH=/usr/local/bin:/mingw/bin:/bin:/c/Python27
PATH=$PATH:/c/WINDOWS/system32:/c/WINDOWS
export PATH
```

Copy it into the home directory noted above, and rename it to `.profile`. This can be easily done in the MSYS shell:

```
mv yourfile.txt .profile
```

Open another MSYS shell window and `echo $PATH` again to confirm that it now has the right value. Try:

```
python --version
gfortran --version
```

If you get something resembling sensible responses, you should be all set for BCMD.

4 Interfaces

Partly in reaction to BRAINCIRC's reliance on a tortuous fixed directory structure, BCMD does not enforce such an arrangement. The BCMD tools can all be invoked manually from the command line, with detailed control over things like file names, source paths, extensions and so on. There is no notion of a 'current' or 'active' model—each compiled model is just an ordinary program that can be run as you see fit.

However, while this approach provides the greatest flexibility, it can be rather fiddly and tedious, and some additional interfaces are also provided. These are more restrictive, but make it easier to do the most routine tasks.

4.1 Using the Makefile

The low-level wrapper interface is available via a Makefile—a script defining tasks and dependencies which a `make` program then executes to compile and run software—in this case models. Invoking `make` is done at the command line like this:

```
make target
```

where `target` is the name of some task that the Makefile defines. These are typically pattern-based, which is to say the Makefile describes not just how to make one specific target, but all targets of the particular form, such as building a model from its definition. (Note that the `make` command has to be run from the same directory the Makefile is in, though the targets can be in other directories.)

The BCMD Makefile defines several convenience targets to simplify the building and running of models. These targets only cater to relatively simple cases, but they should provide a useful starting point.

In order to simplify the targets, the Makefile assumes a predefined directory structure: model definitions and input files are assumed to live in the `examples` directory, and compiled model files and runtime outputs are placed in the `build` directory. Moreover, given a model called *foo*, the following file name conventions are adopted:

- *foo*`.modeldef` is the name of the (primary) file containing the model definition. (This may, of course, import other files.)
- *foo*`.input` is the name of the input file to use when running the model.
- *foo*`.c` is the name of the generated C code file.
- *foo*`.model` is the name of the compiled model executable.
- *foo*`.out` is the name of the coarse results file from running the model with the input.
- *foo*`.detail` is the name of the detailed results file.
- *foo*`.gv` is the name of the GraphViz model representation.

In addition, several log files (*foo*`.log`, *foo*`.stdout` and *foo*`.stderr`) and intermediate compiler files (*foo*`.tree`, *foo*`.bcmpl`) may be generated in the `build` directory. These are briefly described later, but are usually only relevant when the model fails for some reason.

In order to use the Makefile, place your source files in `examples`, go to the BCMD home directory in a terminal, and type `make build/`*foo*`.detail` (substituting the appropriate model name). You can also use `make build/`*foo*`.model` to create the model executable without running it, or `make build/`*foo*`.c` to only generate the C code without compiling it. If you have GraphViz installed, you can create a

PDF version of the model graph with `make build/`*foo*`.pdf` (this assumes you have already built the model).

By default, the Makefile builds everything in a debug mode that can generate a rather large amount of output (mostly in the *foo*`.stderr` file) and run significantly more slowly. Once you are sure that your model is building and running correctly, you can build in non-debug mode by adding `DEBUG=0` to the make command. (You will first need to remove the previously built files, e.g. with a command like `rm build/`*foo*`.*`. Alternatively, you can remove *all* compiled and intermediate files and libraries with the command `make clean`—but be aware that this will delete any results files you have left in `build`.)

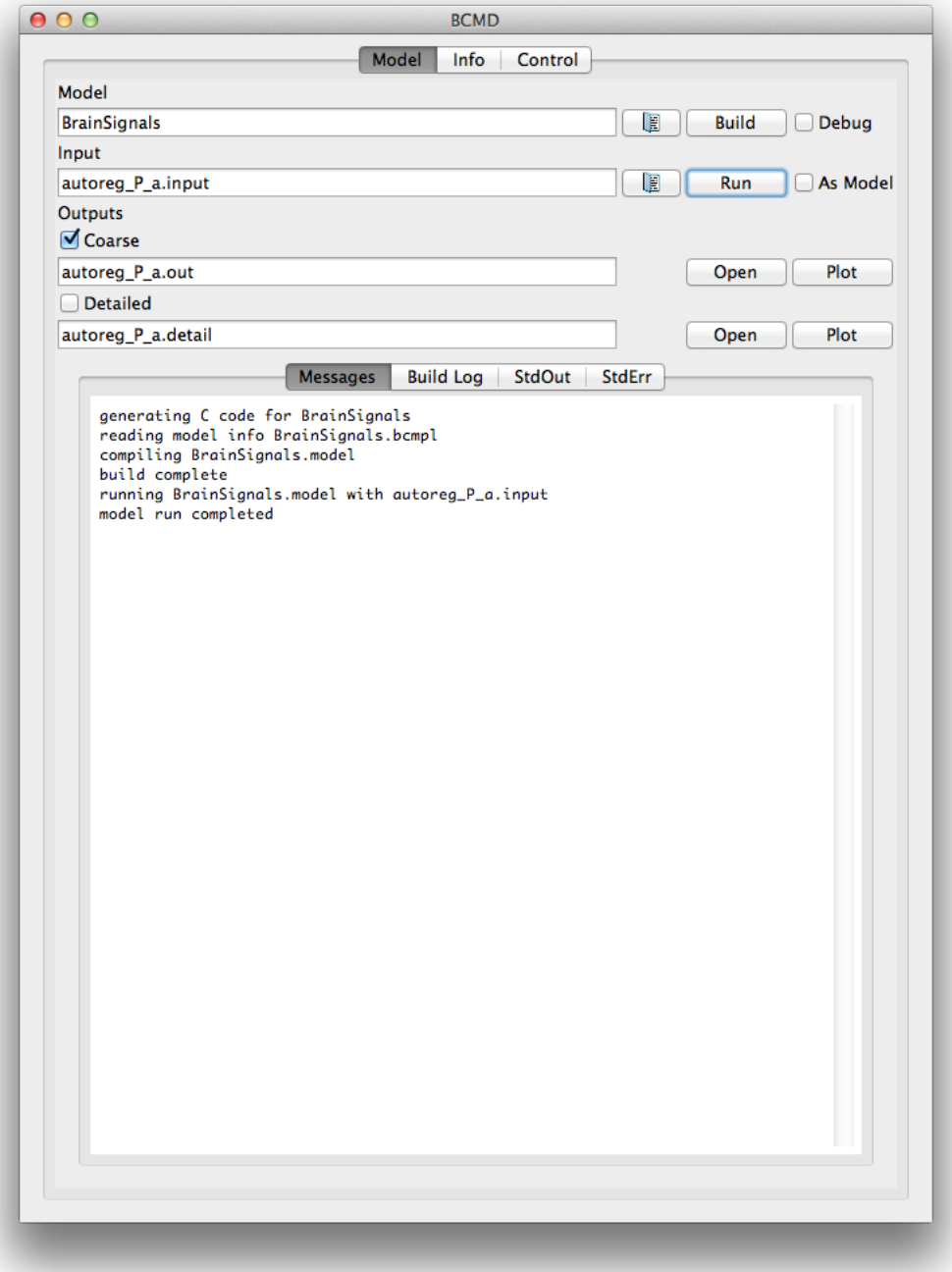

Figure 1: BGUI main interface window

4.2 BGUI

BGUI is a work-in-progress graphical interface for building models and running them with a chosen input file. It provides limited facilities for viewing and plotting the results, though these are mostly useful as an immediate sanity check—more quantitative analysis should be done in a proper numerical environment like R or Matlab. It can also display information about models and provides tools to help with construction of the input files that control a model run.

Use of BGUI requires the Python graphical interface toolkit Tkinter, along with the Matplotlib plotting library and the NumPy & SciPy mathematics packages. Generation of model dependency graphs requires the GraphViz tools.

BGUI is launched from the command line like this:

```
python bgui.py
```

(On Unix platforms the script is executable, and may be invoked directly. However, explicitly including `python` is more dependable.)

A number of program settings, such as the current model and input file, are saved to a preferences file and reloaded when BGUI is launched. This *shouldn't* cause problems, but if you encounter any issues it may be worth deleting this file, which is named `.bcmd_prefs` and saved in your user home directory.

The main BGUI window is shown in Figure 1. (The exact appearance will vary from platform to platform). It has a tabbed interface whose three tab panels, **Model**, **Info** and **Control**, are described in the sections below.

4.2.1 Model Window

The area at the top controls the building and running of models. You can either specify a model by name or press the adjacent button (with the indecipherable blue icon) to bring up a file selection dialog to choose a `.modeldef` file. Once selected, press 'Build' to build and compile the model. The checkbox at the right determines whether the model is built in debug mode.

Once a model has been built, you can run it with a specified input file. Again, click the adjacent blue button to choose the file, and then the 'Run' button to run the model with it. (The checkbox on this line determines whether, when you choose a model file, the input file name should be updated to match. Although this may be marginally useful in some simple situations, it is rather non-obvious and will probably be changed in future.)

The next two rows specify whether to generate coarse and/or detailed outputs, and what the names of these files will be. The output names are automatically set to match the input file (and this will cascade from the model file if the 'As Model' checkbox is checked), but can be changed if desired.

The area at the bottom of the window contains several different log consoles, giving messages from different parts of the build and run process. The first, 'Messages', gives short summary messages from BGUI itself. 'Build Log' shows messages received while compiling the model, which may be useful if the build fails. 'StdOut' and 'StdErr' show the two output streams from running the model itself—the latter may be prohibitively large for a sizeable model in debug mode.

(Note that model definitions and input files may be located anywhere you like, provided that you use the file dialogs to specify them. If you type the names directly, BGUI will look for them in the last directory you specified for the corresponding file type. For the moment, build products such as the model executable itself and the resulting data files are always placed in the standard BCMD `build` directory—at some point there will probably be a way to change this.)

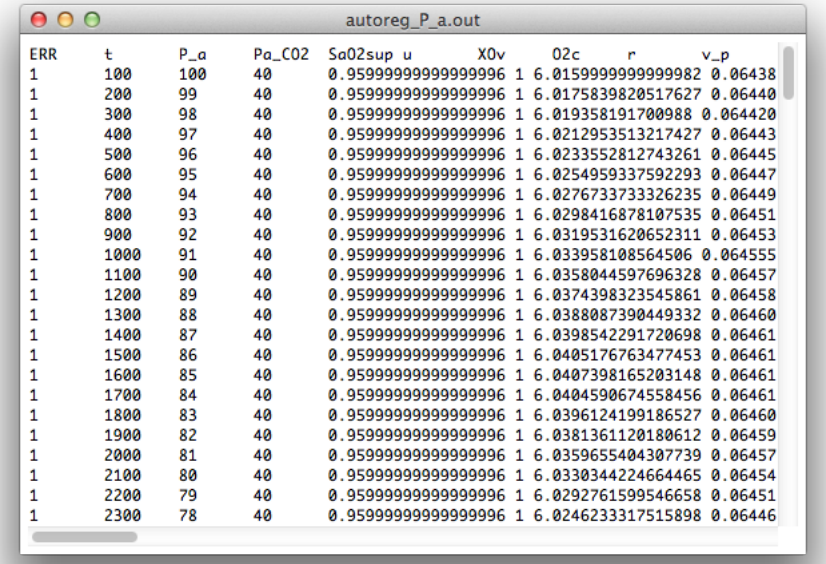

Figure 2: BGUI textual results display

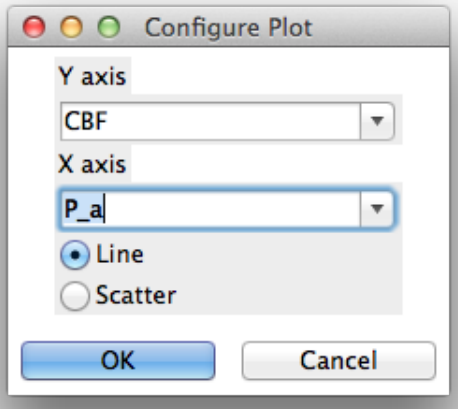

Figure 3: BGUI plot data selection dialog

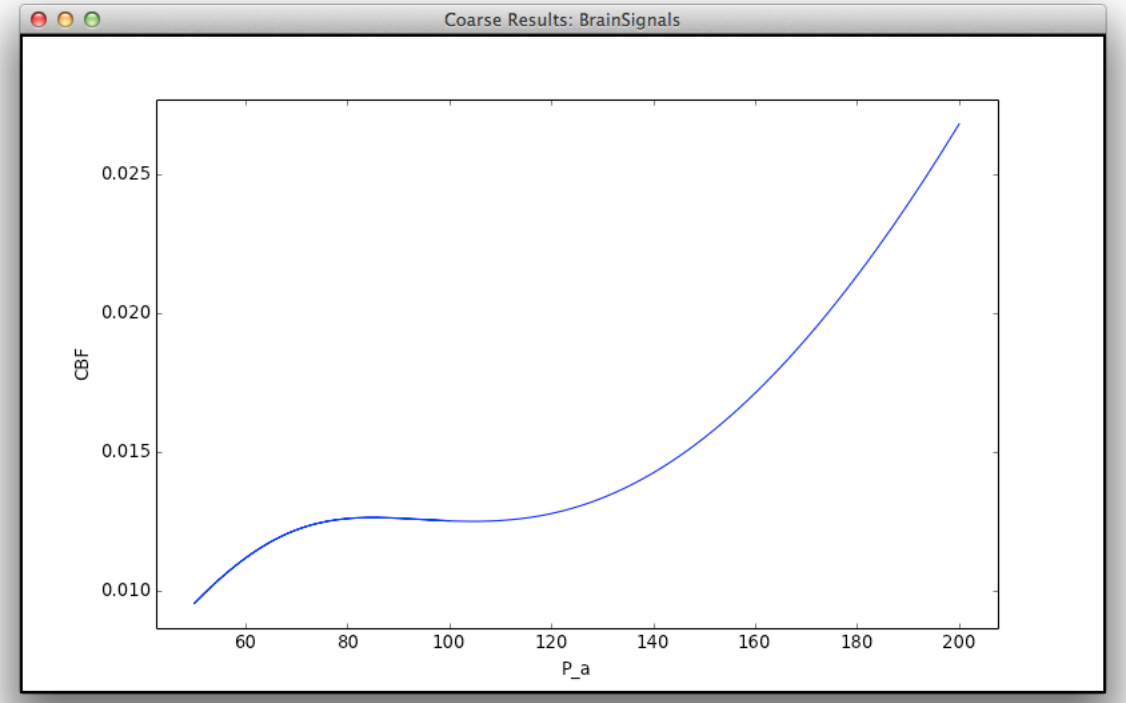

Figure 4: BGUI plot window

Once a model has been successfully run, you can examine the outputs using the buttons to the right of the respective output filename. The 'Open' button will display the tab-delimited results in a simple text window (Figure 2), while the 'Plot' button allows simple plotting. It will first pop up a dialog (Figure 3) allowing you to select the plot type and which column from the result file should be plotted on each axis. (It is not currently possible to plot multiple traces together, nor to plot data directly from the input file.) Once you 'OK' the configuration, the plot itself will appear (Figure 4). Note that both the text and plot windows are self-contained, so you can have several open at once, which is not as useful as plotting together but at least allows some comparisons to be made.

4.2.2 Info Window

The Info window (Figure 5), as the name suggests, provides information about the current model. Note that this is only up to date after the model has been successfully built. The bulk of the information is shown in the large text area that occupies most of the window. This includes summary data about the size of the model, a list of the model equations, some dependency analysis, etc.

The area at the top of the panel allows you to specify the current model, either by entering the name or using a file dialog. This is the same model as on the main Model window—selecting on either tab panel will also change the model on the other.

The 'Parse' button reads the model information generated when the model was compiled. BGUI will do this automatically when it builds the model, so it is rarely necessary to use this button. However, if for some reason you want to load the info for a model without rebuilding it first, you can do so here, provided the model has been built previously and the resulting (`.bcmpl`) info file remains available. The 'Definition' button will display the model definition itself in a simple text window. Note that

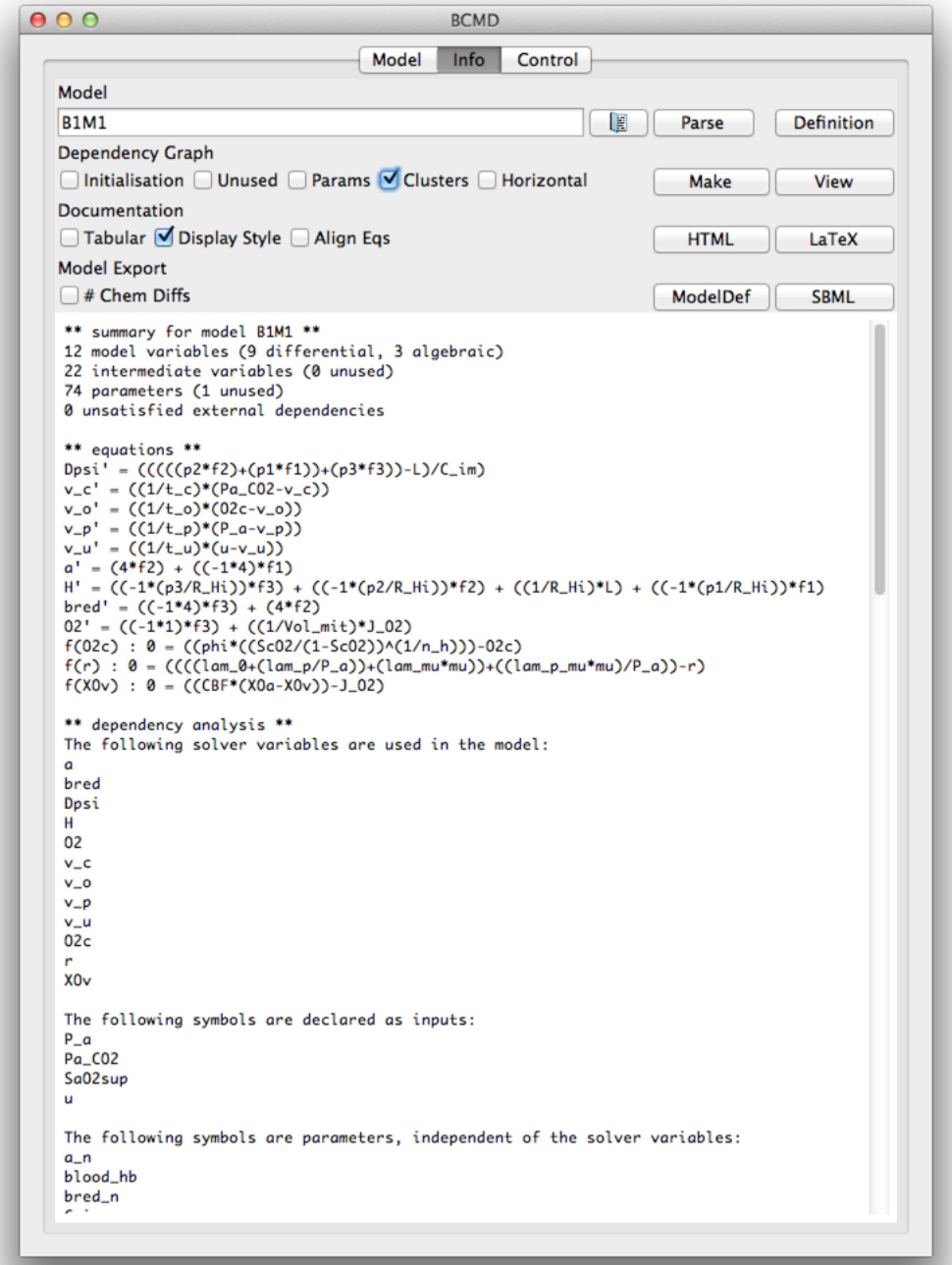

Figure 5: BGUI info pane

this is not a proper text editor and you cannot save changes from here, it is only for viewing.

The next row allows you to configure and build a dependency graph like the one shown in Figure 10. The first four checkboxes specify whether the graph should include: symbols that are only used at initialisation; symbols that never feed back into the model behaviour (often representing output or measurement processes); parameters; and whether symbols should be clustered into subgraphs based on their tags (see §4.4.3 and §5.1.1). The fifth checkbox, 'Horizontal', specifies the principal direction[1] of the graph: if checked, the graph is drawn from left to right, otherwise it is drawn from top to bottom.

Once the settings have been chosen, the 'Make' button creates the graph in PDF and GIF formats—the files are saved in the usual `build` directory and have the same base filename as the model. The 'View' button opens the generated GIF in its own window, but note that this viewing option is *extremely* primitive. In particular, it does not support rescaling, so often only a small portion of the graph can be seen at a time. This is mainly intended for spot-checking that the graph creation worked—for proper examination, open the PDF version in a standard PDF viewer.

The Documentation controls in the next row allow you to generate model documentation. This is based on the model definition, supplemented with any information you have provided in documentation comments (see §5.1.1). The 'HTML' button will generate documentation in HTML format, viewable in any reasonably up-to-date web browser. The generated file is named *modelname*`.html`, and is created in the standard `build` directory. Similarly, the 'LaTeX' button generates documentation in LaTeX format, writing the results to `build/`*modelname*`.tex`. The content of the files generated in the two formats is broadly equivalent, but they are not identical.

The HTML documentation is mostly intended for use as a reference by model developers. It includes a number of implementation details such as source model files, embedded code, symbol types and tag membership and also provides a lot of internal hyperlinks. Chemical reactions are *not* included, and mathematical expressions are shown in the somewhat opaque form the parser produces, with a lot of defensive parentheses—no attempt is currently made to improve their readability, although this will likely be added in future. At present there are no configuration options provided for HTML documentation.

The LaTeX output is aimed more for inclusion in written papers or published supplementary materials, and focuses on presenting formulæ and parameters. The checkboxes to the left allow some configuration of the output, although you may well also require some *post hoc* adjustment to make things like equations fit correctly on the page. The current LaTeX options are:

Tabular If this box is checked, variables and parameters are listed in a `longtable` environment; otherwise they are presented in a `description` list (like the one describing these options). The former may be more appropriate for some kinds of document, but it will usually require more fiddling with the generated LaTeX code in order to make the columns look right on the page. The description format is less economical with space, but will probably need less manual intervention.

Display Style If this is checked, mathematical symbols and expressions are drawn in the 'display' style, otherwise standard inline formatting is used. The latter makes somewhat better use of vertical space, and may lead to better arrangement of text on the page, but will often render complex expressions and fractions unreadably small. Since the generated documentation is mostly a bunch of lists or tables anyway, uniform body text is probably not a primary concern, so enabling display style is recommended.

[1] In the absence of cycles, dependent nodes in the graph are positioned either below or to the right of nodes on which they depend. Even moderately complex models will often include cycles, which will break this ordering, but the overall direction is generally maintained. For example, the graph in Figure 10 is drawn in top-to-bottom order.

Align Eqs If this is checked, the differential and algebraic equations in the model are rendered using an `align` environment, rather than as a sequence of separate `equation` environments. If your equations are all quite short, this *may* look better—the principal difference is that all the equals signs should line up—but if there are long equations that you need to split across more than one line then it becomes quite inconvenient. For that situation, use the normal equation format, then edit the generated file to use a `split` environment for the problematic equations.

The generated file is a complete LaTeX document with a simple `article` wrapper that includes the necessary packages. It should be possible to typeset it directly. You can also easily copy out the bulk of the content into another document of your choice. Note that `ams math` package is required in all cases. In addition, the `mhchem` package is needed if there are any chemical reactions in the model, and the `longtable` package is required if you have exported using the 'Tabular' option.

The Model Export controls allow you to export the model definition in two formats.

The 'ModelDef' button saves the current model as a self-contained BCMD model definition file with standard formatting. This can be useful in cases where the model definition is spread across a number of different files, particularly if you wish to make the model available to someone else. If the model includes chemical reactions, they are included as such in the exported definition. However, it can be useful to see the differential equations that the compiler has constructed from the reactions. If the (admittedly cryptic) '# Chem Diffs' checkbox is ticked, the exporter adds these differential equations *as comments* in the relevant section of the modeldef file. The generated file is named *modelname*`.modeldef`, and is created in the standard `build` directory. Be aware that if your original model definition is in this directory (this is not recommended, but possible), then using this export option will overwrite it.

The 'SBML' button exports the model using the Systems Biology Markup Language (Hucka et al. 2003; Finney and Hucka 2003). The generated file is named *modelname*`.xml`, in the standard `build` directory. Note that this feature is *extremely provisional*. Some BCMD features cannot be easily represented in SBML, and others sort-of can but aren't yet. Export results may not be valid SBML, or may not exactly reproduce the original model. Handle with care!

4.2.3 Control Window

The Control window (Figure 6) provides tools for creating BCMD input files, including explicit parameter setting, data import and/or synthesis and choice of outputs.

The region at the top of the window is used to specify the file to be created. Enter the intended filename in the 'File' field. The button with the indecipherable blue icon is in this case used to choose the *directory* in which to create the named file, rather than the filename itself (since the file may not exist yet). Once the file has been identified, the 'Generate' button will create the file. (Pressing 'Return' or 'Enter' after entering text in the filename field has the same effect as pressing this button.) If the 'Set Input' checkbox is ticked, then generating the input file will also set that file as the current input file on the Model window.

The contents of the generated file are specified in the tabbed region below. There are three separate tabs, corresponding to different conceptual stages of running the model: initialisation, the actual simulation steps, and output generation. These are described in the following sections.

Note that a standard simulation workflow is assumed. Some of the more esoteric capabilities of the input file format—such as sending different data to the coarse and detail output streams, or emitting headers somewhere other than at the top of the file—are not supported here. However, the omitted features are seldom used in practice.

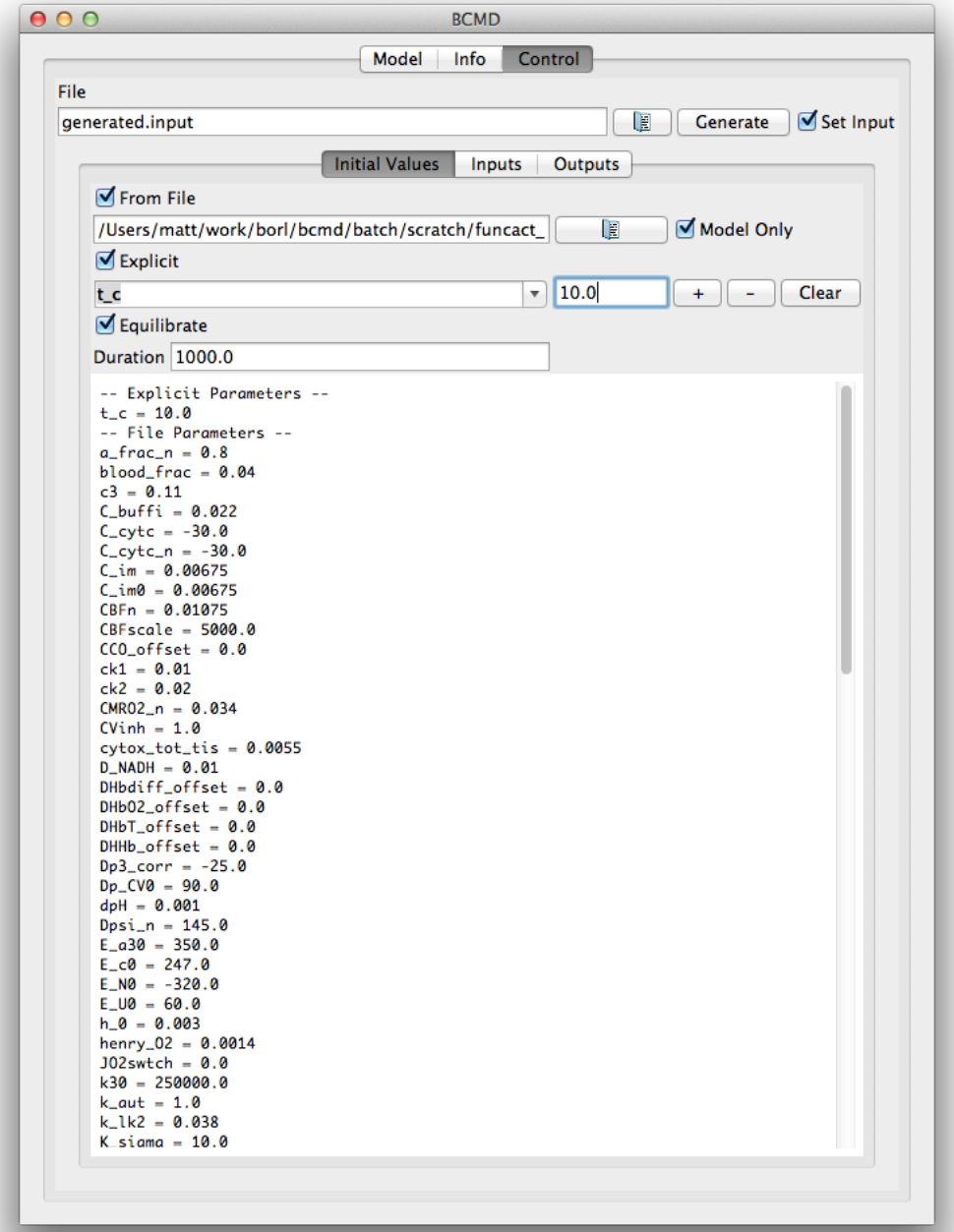

Figure 6: BGUI control/initial values pane

4.2.4 Initial Values

The Initial Values panel (Figure 6) allows the values of model parameters and variables to be set prior to beginning the simulation, and also supports an optional initial equilibration period to allow the values to 'burn in'. Values may be specified individually and/or imported wholesale from a file.

If the 'From File' checkbox is ticked, parameter values will be loaded from the file specified in the associated text field. Use the icon button to select the file via a standard open file dialog. Supported file types are comma-separated value (`.csv`), tab-delimited text (`.txt`) and the old BRAINCIRC (`.dat`) format. For text and CSV, the values can be arranged by rows or by column. Only numeric values can be assigned—any other values in the file will be ignored.

If the 'Explicit' checkbox is ticked, values may be specified manually. The name of the parameter to be assigned can be typed into the associated combo box control (shown containing 't_c' in the figure), or selected from the box's popup menu, which should list all of the symbols in the current model (assuming it has already been built). Enter the value for the parameter in the adjacent text field (containing 10.0 in the figure) and press the '+' button to add it to the list of initial values shown below. The '–' button removes the specified parameter from the explicit assignments, while the 'Clear' button erases all explicit assignments. (Note that when a parameter is set explicitly that also appears in the parameter file, the explicit value takes precedence.)

The 'Model Only' checkbox specifies whether variable and parameter names that are not present in the current model should be included in the generated input file. It may be useful to uncheck this if you wish to generate a file that will be used with several different models with different sets of parameters.

To run the model for a period before beginning the simulation proper, check the 'Equilibrate' checkbox. The length of this equilibration period is specified in the 'Duration' field, in whatever time units the model uses (typically seconds).

The large text area at the bottom of the panel lists all initialisations that will be included in the generated file. It *should* update as you make changes, although there might be a few edge cases where the updating does not happen immediately.

4.2.5 Inputs

The Inputs panel (Figure 7) specifies the actual time steps of the simulation. At present, only a single step sequence is supported—it is not possible to chain together data from different sources in succession, or have different sets of parameters set at different stages. So for more complex simulations it may be necessary to concatenate data files externally before importing. However, many basic cases are catered for.

The upper portion of the panel concerns the 'Time Base' for the simulation—i.e., the actual time points over which the model is evaluated. These may either be imported from an external file—for example, containing real experimental data—or synthesised. The two radio buttons in this section determine which of those options is used.

If 'From File' is selected, then the time base is imported from the file specified in the field below. Use the button with blue file icon button to choose a file—supported file types are CSV and tab-delimited text. When a valid file is chosen, the names of the columns of data in the file are loaded into the combo box control to the right. Use this to choose which column contains the time data. (At some point this may be updated to make an educated guess from the column names, but for the moment you need to choose explicitly even if the file contains only one column called 't' or whatever.)

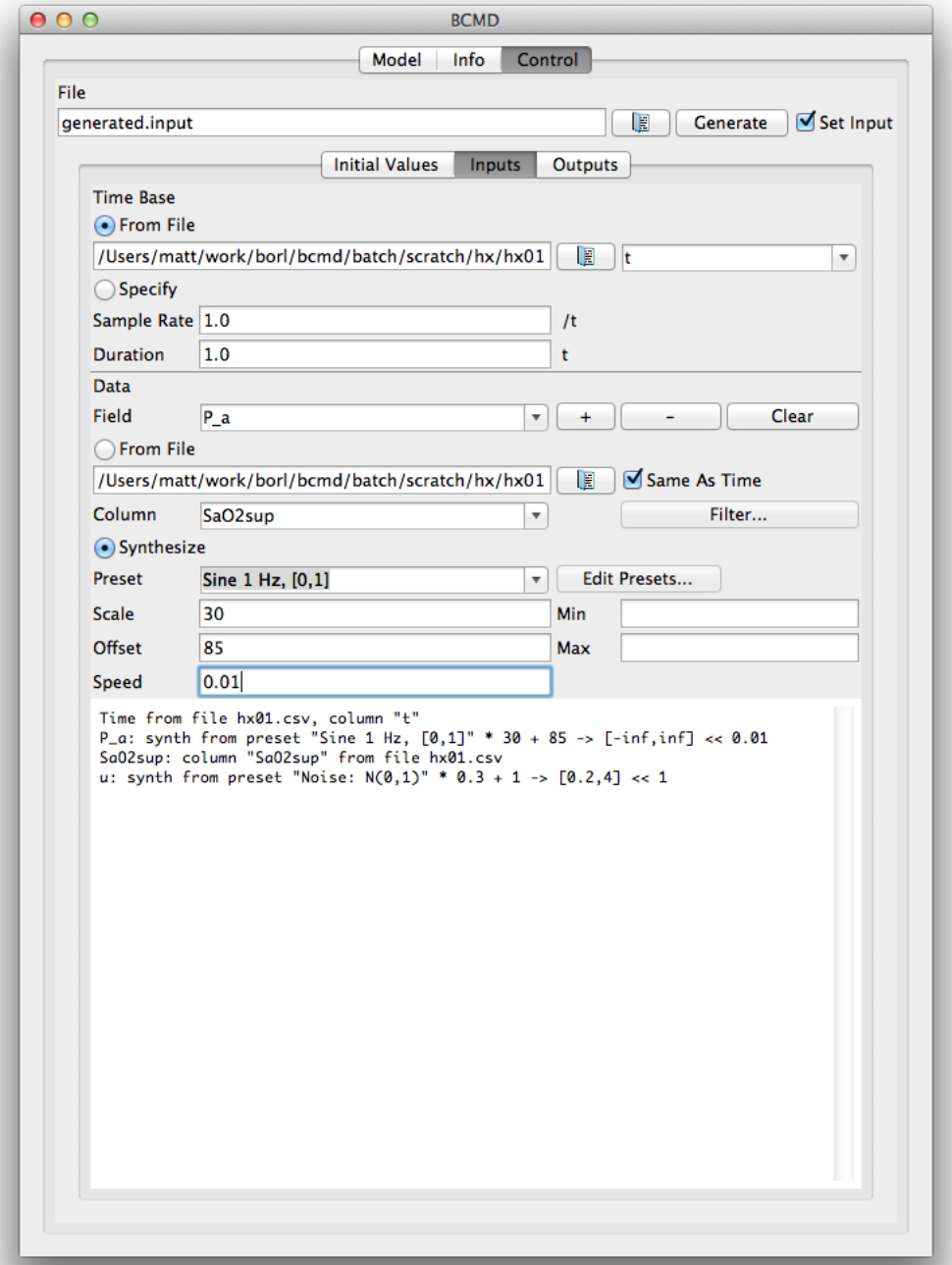

Figure 7: BGUI control/inputs pane

If the 'Specify' button is selected instead, then the time base will be generated from the values given in the 'Sample Rate' and 'Duration' fields. The generated time base always starts from time 0, and the units are whatever the model is defined with.

The central portion of the panel concerns the parameter settings that occur at each time step. As with the time base, these may either be drawn from an external file or synthesised in a number of preset forms. In either case, the field in question must first be chosen using the 'Field' combo box, either by selecting from the pop up menu of model symbols, or by typing any valid identifier of your choice. (You can use a name not in the current model if desired—this may be useful if the same input file will be used with several different models.) The '+' button adds the field to the set that will be included in the generated file—using whatever form has been defined in the sections below. Note that the settings are captured at the point you add them—if you wish to alter the parameters for the signal, you must press '+' again after having done so. The '–' button removes the chosen field, while 'Clear' removes all fields.

If the 'From File' radio button is selected here, the sequence of values for the chosen field will be imported from a file, in the same way as the time base; again, CSV and tab-delimited text are supported. Note that the values for different fields can be drawn from different files, depending on the setting in the 'Same As Time' checkbox: if this is ticked, then a single file is shared for everything—time base and *all* file-based fields. In that case, a file chosen via the selection dialog in either section will immediately apply to both. Conversely, if 'Same As Time' is unchecked, then every field retains its own file source, which can only be changed by re-adding the field using the '+' button. Either way, you must choose which column in the file to use for the field using the 'Column' combo box below.

(The 'Filter...' button is intended for post-processing the imported data. At present, though we have the functionality to do this, there is no user interface to do so. This will be added soon, but in the meantime this button is disabled.)

If the 'Synthesize' radio button is selected, the values to be set are instead generated artificially according to a signal 'Preset', chosen from the pop up menu. Several simple presets are currently supplied:

Noise: N(0,1) Gaussian noise with mean 0 and standard deviation 1.

Noise: U(0,1) Uniformly-distributed noise on the interval [0,1].

Random Walk A drifting baseline generated as the cumulative sum of Gaussian noise with mean 0 and standard deviation 0.1.

Sine 1 Hz, [0,1] A standard sinusoidal wave at 1 Hz, scaled into the interval [0,1].

Saw 1 Hz, [0,1] A sawtooth wave rising from 0 to 1 at 1 Hz.

Square 1 Hz, [0,1] A square wave (50% duty cycle) with frequency 1 Hz and off and on values 0 and 1 respectively.

NIRS mix, [0,1] A mixture of four sinusoids at different frequencies, with added Gaussian noise, scaled into the interval [0,1]. (Based on the simulated signal in Scholkmann et al. (2010).)

(Again, much of the functionality for user-definable presets has been implemented, but this requires a pretty complicated UI that doesn't yet exist, so the corresponding button is disabled for the moment.)

Presets can be adapted somewhat using the settings below:

Scale Specifies a value by which the synthetic signal should be multiplied. (Default: 1)

Offset Specifies a value to be added to the signal. (Default: 0)

Min Specifies a lower bound at which signal values should be clamped. (Default: $-\infty$, i.e. unbounded below)

Max Specifies an upper bound at which signal values should be clamped. (Default: ∞, i.e. unbounded above)

Speed Specifies a multiplier for the time base (or, equivalently, frequency) of the signal. Low values decrease the frequency, high values increase it. This has no effect on the noise presets, since those are time-independent. (Default: 1)

The text area at the bottom of the panel gives a summary of all the selected inputs in a brief shorthand form. See Figure 7 for examples.

4.2.6 Outputs

The Outputs panel (Figure 8) controls what will be included in the output data generated when the simulation runs. The same set of fields is used for both coarse and detailed results (see §4.4.2 for an explanation of the difference).

The 'Include Table Header' checkbox, as the name suggests, specifies whether the output files should include a header row giving the names of parameters reported in the corresponding column of the output. This is usually a good idea unless the output is going to be passed to some existing process that does not understand such headers.

The radio buttons below provide a number of different output options:

Model Default Outputs Do not specify any particular output fields, but instead tell the model to output whatever have been defined as the default output fields within the model itself.

Solver Variables Include only the variables that are actually solved, i.e. those in the $\mathbf{y}$ vector in Equation 1. If nothing else has been specified in the model, this will be equivalent to the default outputs, but note that with this option the field list is included explicitly rather than being left up to the model.

Everything Include *all* symbols that are defined in the model. This can result in very large output files, and often also highly redundant since many parameters will not change during a simulation. However, it can be useful for debugging purposes. Again, the field names are included explicitly in the generated file, as derived from the current model.

Specified Below Explicitly choose the fields to be exported, using the combo controls below.

Defaults + Specified As above, but the default outputs of the current model are added to the explicit list automatically.

Assuming the current model has been successfully built, all symbols from the current model will be listed in the combo box popup menu below. You can also type any other (syntactically valid) name of your choice, although obviously the model will be unable to output fields it does not contain. Press the '+' button to add the chosen field to the list of outputs, or the '–' button to remove it. The 'Clear' button will remove all explicitly-specified output fields (but not those inherited from one of the other options).

The 'Model Only' checkbox specifies that only field names that actually exist in the current model should be included as outputs in the generated file. Again, this depends on the current model having already been built.

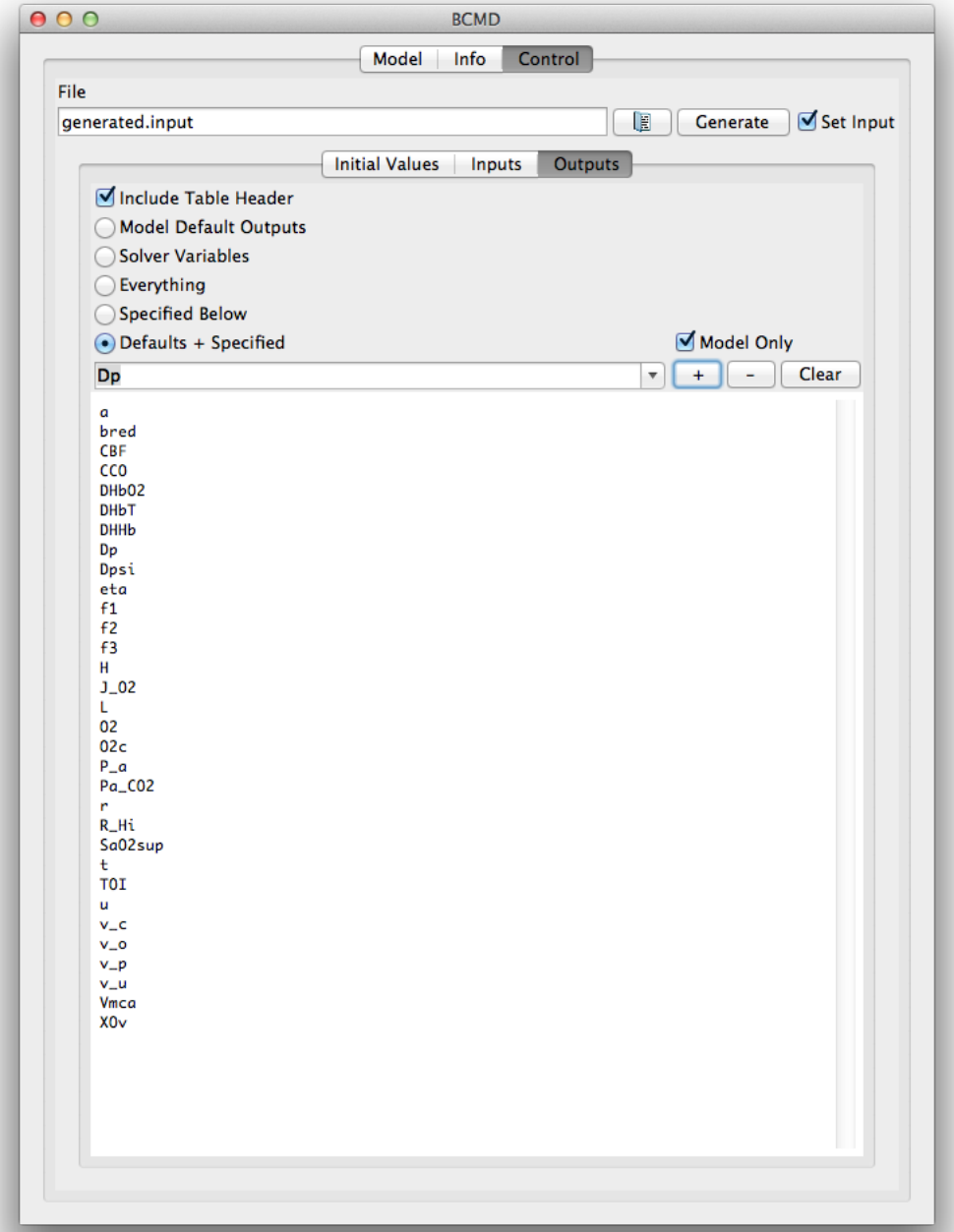

Figure 8: BGUI control/outputs pane

The complete list of fields that will be output is shown in the text area at the bottom of the panel. (Note that in the case of 'Model Default Outputs' this list is implicit, and may vary if the model definition changes or if the file is used with a different model.)

4.3 Batch processing tools

Our computational experiments often involve running large numbers of model simulations under varying conditions. Several scripts used for this purpose can be found in the `batch` subdirectory of the BCMD distribution. Note that these are still very much works in progress, often unfriendly and in some cases undocumented. The current implementations are pretty *ad hoc* and will likely be rationalised in future.

There are two main scripts: `dsim.py` (§4.3.2) manages the running of deterministic or quasi-deterministic batches of simulations, primarily (so far) for the purposes of sensitivity analysis, while `optim.py` (§4.3.3) performs parameter estimation by traditional optimisation[2]. The scripts make use of a plain text *job file* to define the batch task to be performed—these vary between the scripts, but share the same overall format and many common features—these are described in §4.3.1.

The batch tools are run from the command-line. They are unlikely ever to have a GUI in their own right—batch processes are inherently non-interactive—but some tools for constructing the input and job files *may* be added to BGUI eventually.

If the facilities described here don't fit your needs, it may be possible to write a simple wrapper script in Python (or possibly some other language you are happy with) to do what you require. Some fairly trivial examples are included in the `batch` directory. These are not documented—they are just 'quick & dirty' hacks to run some jobs that we needed at some point—but may provide a useful starting point. E.g., have a look at `co2batch.py`, a simple hard-coded batch script that runs a sequence of `dsim` jobs over a directory of data sets.

4.3.1 Shared Features

All three main batch scripts take two arguments, like this:

```
python script.py jobfile datafile
```

The job file specifies details such as what model to use and what variables to simulate, while the data file specifies time series data that will be used when running the job. The latter should be a CSV or tab-delimited text file, with column headers that will be used to identify the variables or inputs. (It is also possible to use a BRAINCIRC `.dat` file as the data source, but hopefully the necessity for that is asymptoting to zero...)

Job files are line-oriented plain text. Blank lines, lines starting with a hash character `#` and lines specifying any details the script doesn't recognise are simply ignored. Lines defining the job start with a keyword, followed by a colon, then one or more details, separated by commas, as in the following snippet:

```
# specify which model to use
model: BrainSignals

# variable to be simulated
# with the distribution of initial conditions
```

[2] A third script, `abcmd.py`, attempts to do parameter estimation via Approximate Bayesian Computation. At present this is not functioning correctly and is not documented here.

```
var: CBF, constant, 0.01075
```

Several fields, like `var` in the above example, allow a *distribution* to be specified. How (or whether) this is actually used varies according to context. The first element (following the variable of parameter name) is the distribution type, followed by some optional parameters (all of which default to 0) and an optional default value (which may be calculated from the parameters if they are given and it isn't). Four types are recognised:

constant, [*value*] No variation, always use the specified value, or 0 if omitted.

uniform, *min*, *max* [, *default*] The value is uniformly distributed between *min* and *max*. The default default is $(min + max)/2$.

normal, *mean*, *variance* [, *default*] The value is normally distributed with specified mean and variance. Default defaults to *mean*.

lognormal, *mean*, *variance* [, *default*] Value follows a log-normal distribution, where the underlying normal distribution has the specified mean and variance. Default defaults to e^{mean}.

Any distribution that is not otherwise specified defaults to `constant, 0`.

While each kind of job has some keywords that are specific to it, the following are used by all three scripts:

model Name of the model to be used for the simulations. This should be just the name itself, without any file extensions, and should follow the rules for model identifiers (numbers, letters and underscores, not starting with a number). Various other job details are constructed from this if they are not independently specified.

var† An output to be simulated in the model. The first element is the name of the variable. A distribution can be specified in subsequent elements—if the job type supports it, this is used for drawing the initial value of the variable at the start of the simulation. A target time series for the variable should usually be provided in the input data file (possibly aliased, see below), although in some cases it is possible not to do so (in which case the results will be evaluated relative to 0).

input† An input for which time series data will be supplied to the model. Data must be provided in the input data file (possibly aliased, see below). The first element is the name of the input parameter or variable. Optional subsequent elements specify its distribution—if the job type supports it, this is used to *perturb* the supplied data (i.e., to add noise so that the inputs are not identical each time).

param† A parameter that can be set in the simulation. The first element specifies the parameter name. Optional subsequent elements specify a distribution, which is used in different ways by the different job types. Note that parameters can be described and then not used—see `param_select` below.

param_file† Specify an optional external file containing parameter details. This can be useful if you have a lot of parameters with the same distributions used in different jobs. The file can be a CSV or TXT file, but is treated in a line-oriented rather than tabular fashion (so different rows need not have matching columns). Each row is interpreted as if it were the right hand side of a `param` entry in the job file, i.e. the parameter name followed by an optional distribution. If a parameter in this file is also explicitly specified in the job file, the job file version overrides the settings in the parameter file.

param_select† Specify the parameters that will actually be used in the job. Each element on the right hand side is a parameter name. If this keyword is not present, or if any

element is an asterisk `*`, then all parameters for which a distribution has been specified, in both the job file and the parameter file, are included.

alias† Specify that an input or variable time series appears in the input data file under a different name. The first element is the name in the model, the second is the name in the file. If not specified, variables are assumed to have the same name in both, with one exception:

IMPORTANT: For historical reasons involving the support of BRAINCIRC files, internal processing of the independent variable (usually time) in these scripts uses a different proxy name. If you explicitly specify times in your input data file (almost always the right thing to do), you need to include an alias for it *even if it seems to have the right name already*. Use a line like this:

```
alias: t, t
```

Yes, this is obviously stupid. It will probably be sorted out in future, but for now just go with it.

program The model executable. Defaults to `build/`*modelname*`.model`. You should only need to specify this if you're doing something unusual.

work Name of a directory in which to place work and result files. The given directory is created inside the standard `build` directory. If not specified, a new directory is created at `build/`*modelname/timestamp*.

model_io Name of a directory in which to place the simulation input and output files. This directory is placed inside the work directory, above. By default it is called `model_io`.

init Specifies a BCMD or BRAINCIRC input file to use as the initial steps of every simulation in the batch. This sequence is simply prepended to the generated one—this means the steps are executed *before* any parameter setting specific to the batch job is performed.

Any keyword can occur multiple times in the job file without error. Those marked above with a † symbol are *intended* to support multiple occurrences, and all specified values are used—clearly it may be necessary to simulate more than one output or vary more than one parameter. For the remaining, unmarked, keywords, only the first occurrence is used—subsequent appearances are legal but ignored.

All the batch processing scripts depend on the `model_bcmd.py` module, which implements a class conforming to the ABC-SysBio model interface. This wraps up the process of invoking a BCMD model with given inputs, and then reading and returning the results. It also handles parallelisation, allowing it to run some specified number of models simultaneously, which should be faster on multicore systems (this is only supported for `dsim` and `abcmd` at present, not for `optim`).

Most of the business of constructing the step sequences for the model input files is handled by the `steps.py` module, which can also be invoked from the command line to help when constructing inputs of your own (see §6.2). Reading data from TXT, CSV and BRAINCIRC files is handled using the `inputs.py` module. Some other data conversion utilities are `descs2csv.py` and `dat2csv.py`.

Finally, the `distance.py` module provides a number of distance metrics (or quasi-metrics), each in two forms, one conforming to an ABC-SysBio interface and the other in a simpler functional form. Each calculates a scalar value that in some way quantifies the distance between two vectors of the same length, say $\mathbf{a}, \mathbf{b} \in \mathbb{R}^N$. The following measures are currently supported:

euclidean The Euclidean or L^2 distance is the most common distance metric, calculated as:

$$\text{euclidean}(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\| = \sqrt{\sum_i a_i^2 - b_i^2}$$

manhattan The Manhattan, taxicab or L^1 distance is the sum of the absolute component differences:

$$\text{manhattan}(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_1 = \sum_i |a_i - b_i|$$

mean The mean or RMS distance is the mean squared componentwise difference between the vectors:

$$\text{mean}(\mathbf{a}, \mathbf{b}) = \sqrt{\frac{\sum_i a_i^2 - b_i^2}{N}}$$

(This is literally just a scaled version of the Euclidean distance, but it's kept around for historical reasons.)

loglik The log-likelihood distance is based on the probability that the difference between the two vectors is attributable to Gaussian noise with some specified standard deviation, σ (by default 1):

$$\text{loglik}(\mathbf{a}, \mathbf{b}) = \frac{N}{\sqrt{2\pi\sigma^2}} - \sum_i \frac{a_i^2 - b_i^2}{2\sigma^2}$$

cosine The cosine distance gives the cosine of the angle between the two vectors:

$$\text{cosine}(\mathbf{a}, \mathbf{b}) = \frac{1}{2} - \frac{\mathbf{a} \cdot \mathbf{b}}{2\,\|\mathbf{a}\|\,\|\mathbf{b}\|}$$

(The factor of $\frac{1}{2}$ scales the result into the range [0,1].)

angular The angular distance gives the angle between the two vectors:

$$\text{angular}(\mathbf{a}, \mathbf{b}) = \frac{1}{\pi} \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\,\|\mathbf{b}\|}\right)$$

(The factor of $\frac{1}{\pi}$ scales the result into the range [0,1].)

4.3.2 Deterministic Batches & Sensitivity Analysis

The `dsim.py` script runs multiple simulations with different combinations of parameter values, where the values are drawn from a set of defined 'levels' according to a predefined scheme. This is nominally 'deterministic' because the sequence of simulations is fully determined before any runs take place and does not change according to the results of individual simulations. Some job modes are indeed wholly deterministic and will run identically each time. However, in probably the most common usage, that of Morris-style trajectories for sensitivity analysis, there is nevertheless an element of randomness. It is also possible to add random perturbations to the input data.

The script is run with Python from the Unix or MSYS command line:

```
python dsim.py [options] jobfile datafile
```

(On Unix platforms the script is executable, and can probably be run directly. However, explicitly including `python` is more dependable.)

The following command line options are available:

-h, --help Print a usage message and exit.

--version Print the program's version number and exit.

-r FILE, --results FILE Specify a name for the output file name. By default, the file is named `results.txt`.

-b DIR, --build DIR Specify the location of the build directory (this is where the model executable is expected to be). The standard BCMD `build` directory is used by default.

-o DIR, --outdir DIR Specify a directory for working files and results. The default is a directory *modelname/timestamp* inside the specified build directory.

-p, --perturb Enable perturbation of the input data (see below).

-d, --dryrun Assemble the data and construct the set of jobs, then write all the configuration details to `stdout` and exit without simulating.

The job files for `dsim` are conventionally given the file extension `.dsimjob`, although this is not enforced. In addition to the common features described §4.3.1, the following keywords are supported:

job_mode What kind of job to run. This should be one of:

- `single`: Each selected parameter is varied across its range individually, while all other selected parameters are kept at their default value. Scales $\mathbf{O}(np)$, for p parameters and n levels.
- `pairwise`: Each pair of selected parameters is varied jointly across their ranges while all other selected parameters are kept at their default. Scales $\mathbf{O}(n^2p^2)$, for p parameters and n levels.
- `cartesian`: All selected parameters are varied jointly. That is, every possible combination of the available levels of all parameters is tried. Scales $\mathbf{O}(n^p)$, for p parameters and n levels.
- `morris`: All parameters are varied by single steps in random sequence. This is an implementation of the Morris 'elementary effects' sampling scheme for sensitivity analysis (Morris 1991; Campolongo et al. 2007), currently limited to single-level steps. A single 'trajectory' (of $p + 1$ steps) samples sequential changes to all parameters. Note that `dsim` only runs the simulations, it does not perform the analysis—see below. Scales $\mathbf{O}(kp)$, for p parameters and k trajectories.

(Default: `single`)

divisions The number of levels into which the parameter range should be divided. Simulated values are drawn from these discrete levels. Division is performed in *quantiles* of the distribution specified for the parameter. In most cases you will probably want to specify a `uniform` distribution, in which case all levels are equidistant. For `normal` and `lognormal` distributions, steps close to the mean are closer together. (Both these distributions have infinite support, so obviously we don't span the whole range—instead, the range is divided into *divisions* + 2 quantiles and the non-finite endpoints discarded.) Parameters specified as `constant` are not divided. (Default: 10)

nbatch The number of model instances than can be run in parallel. This is not guaranteed to make things faster, but may do so, especially for processor-bound jobs on multicore hardware. Try setting this to the number of cores in your machine. (Default: 1)

beta The number of repetitions of each job to run. This is ignored unless input perturbation was enabled when `dsim` was launched. With perturbation, it may make sense to run repeat simulations, since the runs will be non-identical. (Default: 1)

timeout Maximum time in seconds to allow an individual simulation to run before forcibly terminating it. While model runs are typically fast, there can sometimes be regions of the parameter space that lead the solver to get stuck for long periods, perhaps indefinitely. Such cases make the whole batch unacceptably slow, so after awhile we abandon the attempt. (Default: 30)

npath The number of Morris trajectories to run. Ignored for other job types. (Default: 1)

path_start Where to start Morris trajectories. Normally each trajectory starts from a different random point in the (discretized) parameter space. If instead this is specified as `default`, then the starting point is at the default value for each parameter. This is generally a bad idea, as it limits the sampling to a very small part of the parameter space, but it can be useful in some circumstances. (Confusingly, but rightly, `default` is *not* the default setting here.)

save_interval Since `dsim` batches can take a long time, we may want to save intermediate results as we go along. On the other hand, saving itself takes time, so doing it very often will just slow things down even more. This setting specifies how many batches (sets of `nbatch` parallel simulations) to run between each save. Set to 0 for no intermediate saves. (Default: 100)

All these `dsim`-specific keywords are treated as single-valued: the first occurrence in the file is used, subsequent instances are ignored.

A `dsim` job may require very many simulation runs. As well as potentially taking a *really* long time, simulation results are collated in RAM, so for very large batches the Python interpreter will eventually run out of memory. The order of complexity for each job mode is noted in the description above. Be especially wary of the exponential scaling of the `cartesian` mode—this can get out of hand very quickly. Avoid using `param_select:*` in this mode for all but the tiniest models.

Simulation results are generated in a tab-delimited text file, by default called `results.txt`. The structure is straightforward but slightly opaque. The actual simulation data is in the lower right quadrant of the table. The top section contains information shared between all jobs, while the lower left quadrant specifies job details:

- The first column, `job`, specifies the simulation number. This starts at zero and increments for each new set of parameters to be simulated.
- The second column, `rep`, specifies the repetition number of the job. This again starts at zero. It will only be non-zero if perturbation is enabled and `beta` > 1 given in the job file.
- The third column, `species`, identifies the output variable for which the row contains data.
- The next p columns give the values of all selected parameters in the job. These values will be identical in all rows for the same `job`. The column heading is the parameter name.
- The remaining columns give the output value at each time point of the simulation. The column heading is the time point index, starting at zero, prefixed by `t`: `t0`, `t1`, `t2`...
- The top row is a header specifying column names.
- The next row gives values for the simulation independent variable, here always referred to as `t`. These are in the time point columns `t0`, `t1`, `t2`... The `species` column has the value `t`. All other columns contain the 'missing data' value `NA`.

- The next few rows give the input signals shared by all jobs. The `species` column contains the input variable name, with the suffix `_in` appended. The actual input values are in the time point columns `t0`, `t1`, `t2`... All other columns contain the 'missing data' value `NA`.
- The next few rows give the target output values for each output variable. The `species` column contains the output variable name, with the suffix `_out` appended. The actual input values are in the time point columns `t0`, `t1`, `t2`... All other columns contain the 'missing data' value `NA`.
- The remaining rows report the actual simulation results for the specified `job`, `rep` and `species`.

In addition, some information about the job configuration is written to a file called `dsim.info`.

Analysis

The raw simulation data is useful, but often (e.g., for sensitivity analysis) you just want to look at changes in the relationship of the generated signals to their target values. The script `postproc.py` computes a variety of distance metrics for each simulated output and also some summary statistics about them.

The script is invoked from the command-line like this:

```
python postproc.py DIR
```

(On Unix platforms the script is executable, and can probably be run directly. However, explicitly including `python` is more dependable.)

The script takes no optional arguments. The required argument `DIR` specifies the path to a directory that either contains `dsim` results itself, or else contains a number of subdirectories containing `dsim` results. In the latter case, it processes each such subdirectory separately. It does not recurse into lower level subdirectories, and it will skip any subdirectories that do not contain the expected result files.

`postproc` generates three output files:

- `brief.txt` contains some very minimal information extracted from the full `dsim.info` file. This is used by some other scripts, and also to quickly identify what's going on in the directory (since postprocessed directories typically all look the same). At present the only content is the name of the source data file that was used to drive the simulations.
- `distances.txt` contains distance metrics for every output signal in the `results.txt` file. All six of the distance measures described above are calculated for every simulation signal, where the distance is relative to the target output in the top part of the results file. The structure of `distances.txt` is similar to `results.txt`, but it contains no time series data or global rows. Each row corresponds to one output signal from one simulation, identified by the `job`, `rep` and `species` columns. The simulation parameters are recorded in subsequent columns as before. Finally, there are columns for each distance metric, named `dist_L1`, `dist_L2`, `dist_Mean`, `dist_LogLik`, `dist_Cosine` and `dist_Angle`.
- `elementaries.txt` summarises the distance information by parameter. The first three columns identify the species simulated and the parameter varied. The third column states how many valid signals contributed to the summary statistics—distances cannot be calculated for outputs containing non-numeric values (NaN or infinite), so the effective number of contributing signals is typically smaller than the total number of simulations run. Three summary values are given for each distance metric: μ (column name suffix `_mu`) is the mean distance change resulting from a step change in the parameter; σ (column name suffix `_sigma`) is the standard deviation of the distance changes; $\mu^\star$ (column name suffix `_mu_star`) is the mean *absolute* distance

change, which captures some information about direction that can give misleading results for the ordinary mean (Campolongo et al. 2007).

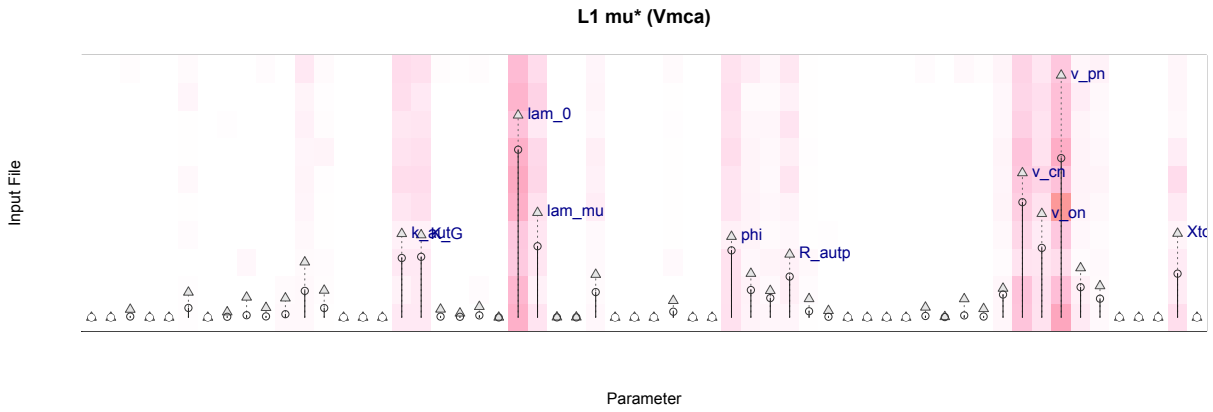

Figure 9: A sensitivity 'heat map', illustrating differences in parameter sensitivities with different input data.

The script `aggregate.py` merges `postproc` output from multiple directories—this can be useful when analysing batches run with different input files. Invoke it like this:

```
python aggregate.py DIR
```

There are no optional arguments. The required argument `DIR` specifies the path to a directory that contains a number of subdirectories containing batch results on which `postproc.py` has already been run. Like `postproc`, it does not recurse into lower level subdirectories and will skip subdirectories that do not contain useful results.

The script is not at all sophisticated—it assumes that the results are all compatible and makes no attempt to check this or conform them if they aren't. In essence, it just concatenates the contents of the `elementaries.txt` files from each result set, prefixing an additional column containing the input file as recorded in the `brief.txt` file. It does not try to enforce uniqueness, so if you run it with more than one directory that was run with the same input then you may get non-unique rows in the output. The merged results are saved to a file called `aggregate.txt` in the directory specified as `DIR`.

One the main reasons for running `aggregate.py` is to generate data for use in R, to plot the kind of sensitivity 'heat map' shown in Figure 9. The function that does this is called `heat_cross()`, and it can be found in the script `util.R` in the `R` directory of the BCMD distribution. The function takes a large number of arguments to configure the plot, but all of them have defaults. In the simplest case, executing `heat_cross` in the directory contain the `aggregate.txt` will load the data and plot the results for one species—the plot in Figure 9 was generated this way. At some point the full function will be documented here, but in the meantime have a look at the R code, which has relevant comments.

4.3.3 Optimisation

The `optim.py` script attempts to fit the value of one or more model parameters to make the model output resemble a specified target as closely as possible. Optimisation is performed using the OpenOpt Python library (http://openopt.org), preferably using the PSwarm solver (Vaz and Vicente 2009, http://www.norg.uminho.pt/aivaz/pswarm/). Installing these dependencies is discussed briefly in

§2, but detailed instructions for all platforms are currently beyond the scope of this document. See the linked sites for more information.

Consider the model as a function $\mathbf{f}(\mathbf{x}, t \mid \Theta, ...)$, where Θ denotes the subset of parameters we're interested in. For specified inputs $\mathbf{x}, t$, let the target output be $\mathbf{y}$. If $\|\mathbf{x} - \mathbf{y}\|_d$ denotes distance by some chosen metric d, then the 'cost' associated with a given parameter combination is

$$g(\Theta) = \|\mathbf{f}(\mathbf{x}, t \mid \Theta, ...) - \mathbf{y}\|_d \tag{3}$$

and our goal is to determine parameters $\hat{\Theta}$ that minimise g. (Note that g is scalar.)

If our models were simple, we wouldn't need BCMD. For most realistic uses, $\mathbf{g}$ tends to be grossly non-convex and minimisation is a questionable proposition (see Boyd and Vandenberghe 2004). If you can re-pose your problem in a different, explicit and preferably convex form, such as a linear regression against a prepared body of output data, you are strongly advised to do so. Non-convex global optimisation is, frankly, a mug's game: tedious to set up, computationally expensive, and almost guaranteed to give you the wrong answer to what is probably the wrong question. Alas, sometimes there just aren't any better options available. If you absolutely *must* do this, `optim` may help; but please be cautious when interpreting the results.

The script is run with Python from the Unix or MSYS command line:

```
python optim.py [options] jobfile datafile
```

(On Unix platforms the script is executable, and can probably be run directly. However, explicitly including `python` is more dependable.)

The following command line options are available:

-h, --help Print a usage message and exit.

--version Print the program's version number and exit.

-b DIR, --build DIR Specify the location of the build directory (this is where the model executable is expected to be). The standard BCMD `build` directory is used by default.

-o DIR, --outdir DIR Specify a directory for working files and results. The default is a directory *modelname/timestamp* inside the specified build directory.

-d, --dryrun Assemble the data, write the configuration details to `stdout` and exit without running anything.

-w, --wetrun Assemble the data, write the configuration details to `stdout` and do a single test run of the model without subsequent optimisation.

-D, --debug Run in debug mode, which logs various pieces of information to `stderr` along the way. If this is specified in addition to **--wetrun**, the single model invocation does not occur, but the parameters that would be used are printed.

The job files for `optim` are conventionally given the file extension `.optjob`, although this is not enforced. In addition to the common features described §4.3.1, the following keywords are supported:

job_mode The kind of OpenOpt job to run. See the OpenOpt documentation for details, such as they are. This should be one of:

- `GLP`: A global optimisation. This is almost always going to be what you want. Strictly speaking, the whole concept of 'global optimisation' is kind of dubious, but anyway.
- `NLP`: A non-linear local optimisation.

- `NSP`: A non-smooth local optimisation. This is much the same as NLP, but with some different default settings.

(Default: `GLP`)

solver Which optimisation method to use for the problem. Different solvers are supported for the different job modes—seee the OpenOpt documentation for more information. Supported solvers for `GLP` jobs include the following:

- `pswarm`: A particle swarm optimisation method. Requires the PSwarm library to be installed, including Python support.
- `galileo`: A genetic algorithm based method. Built into OpenOpt, no external libraries required.
- `de`: A two-array differential evolution algorithm. Built into OpenOpt, no external libraries required.

Other solvers may also work, depending on the nature of the problem and whether you have the relevant external code installed. Pswarm seems to be a good choice for our problems, being somewhat less prone to getting stuck in local minima, but it is often worth trying different solvers and comparing results. (Default: `pswarm`)

distance The distance function to minimise. Value is the actual *name* of the function to call. At present, only functions in the `distance.py` module are supported, there is no way to specify custom functions elsewhere. One of: `euclidean`, `manhattan`, `loglik`, `mean`, `cosine` or `angular`; see §4.3.1 for definitions. (Default: `euclidean`)

max_iter The maximum number of optimisation iterations to run—fewer will be used if convergence is achieved sooner. This is primarily intended for `pswarm`, although other solvers may also support it. Iterations in this case are cycles of polling and searching, not model executions. (Default: 1000)

steady Length of equilibration period at start of each model run (i.e., after setting parameters but before commencing input sequence), in model time units. Set to 0 for no equilibration. (Default: 1000)

timestep If times are not explicitly provided in the data file, you can optionally choose to specify a default interval and let `optim` assign the times. In this case, the simulation will start at time 0 and proceed with uniform steps of the specified duration. (No default: if not specified, times must be in the data file.)

weight† A weighting to apply to the distance metric for an output variable. If you are jointly optimising multiple simulated outputs (i.e., if there is more than one `var` declared), the total cost function g is a weighted sum of the individual signal distances:

$$g(\Theta) = \sum_{v \in \mathbf{V}} w_v \left\| \mathbf{f}_v(\mathbf{x}, t \mid \Theta, ...) - \mathbf{y}_v \right\|_d$$

Here, $\mathbf{V}$ is the set of output variables, while $\mathbf{f}_v(\cdot)$ and $\mathbf{y}_v$ are the simulated and target signals for variable v. The `weight` keyword allows specifying w_v. The right hand side should contain the variable name as the first element and the numeric weight second:

```
weight: CBF, 0.5
```

(Default: 1)

As noted by the † symbol, `weight` is multivalued—you may need to specify weight for several variables. All other `optim`-specific keywords are treated as single-valued: the first occurrence in the file is used, subsequent instances are ignored.

Parameters to be optimised should be given a `uniform` distribution—the bounds of this are used as box constraints on the parameter value. Input perturbation is not supported—a distribution can be provided for each `input` specification, but it will be ignored.

At present, result reporting is extremely basic. The solver may emit various messages as it progresses. Once the job completes, `optim` prints a short summary to `stdout`, reporting success or failure and the final values of the optimised parameters. More extensive reporting may be added in future.

4.4 Direct invocation

4.4.1 Compiling

The BCMD compiler is run with Python from the Unix or MSYS command line:

```
python bparser/bcmd.py [options] model [model ...]
```

(On Unix platforms the script is executable, and can probably be run directly. However, explicitly including `python` is more dependable.)

The following command line options are available:

-h, --help Print a usage message and exit.

--version Print the program's version number and exit.

-i PATH Append PATH to default model search path.

-I PATH Replace the default model search path with PATH.

-n NAME, --name NAME Specify the name of the model. By default the first model file name is used, with no extension.

-o FILE Specify output file name. (This is currently unused as the script does not invoke the C compiler itself.)

-d DIR Specify the directory into which generated files will be placed (default: .)

-u, --unused Do not calculate apparently-unused intermediate variables. This may be somewhat more efficient, but means that these values cannot be output at runtime.

-g, --debug Include debug outputs in generated model code. This can be useful for debugging a model, but will make it run more slowly.

-t [FILE], --tree [FILE] Write the parse tree to a text file. FILE is optional; if not specified, the default is *modelname*`.tree`

-p [FILE], --processed [FILE] Write the whole compilation data structure to a text file. FILE is optional; if not specified, the default is *modelname*`.bcmpl`

-G [FILE], --graph [FILE] Write the model structure to a GraphViz script. FILE is optional; if not specified, the default is *modelname*`.gv`

-U, --graphxunused Exclude apparently-unused variables and parameters from the exported graph.

-N, --graphxinit Exclude initialisation-only variables and parameters from the exported graph.

-C, --graphxclust Exclude cluster subgraphs from the exported graph.

-S, --graphself Include direct circular dependencies (ie, graph edges that loop back to the same node) in the exported graph.

-v LEVEL, --verbose LEVEL Set the level of detail logged to stderr during compilation (0–7, default: 3)

-Y, --yacc Run a dummy parse to rebuild the parse tables.

The compiler looks for model definition files in the directories specified in the model search path. By default it looks in the current directory and, if present, in a subdirectory called `models`. It first looks for the specified name exactly as supplied, then tries with the `.modeldef` extension appended (so it is not necessary to include that). The same search order applies to files imported inside the model definition with the `@import` directive.

If you are invoking BCMD via the default Makefile targets, the logging output is redirected to `build/`*modelname*`.log`—this is the first place to look if you encounter a problem compiling. If there is an error in your model definition it will usually at least tell you what line it's on and what parse token caused the error. This doesn't necessarily diagnose the problem but it should help at least a little bit.

The generated C code file must be compiled and linked with the RADAU5 library. The `%.model` target in the Makefile takes care of this (and does not assume you are using the default `build` directory). If you prefer to compile without using this target, check the Makefile for the requisite compiler options, which should have been identified by the `configure` script.

See §4.4.3 below for more information on the graph generation options.

4.4.2 Running

The compiled model program can be run from the command line. The following options are available:

-h, --help Print a usage message and exit.

-v, --version Print the model version information and exit.

-m, --model Print the model name and exit.

-s, --symbols Print a list of the symbols (i.e., the variable and parameter names) used in the model, and exit.

-i FILE, --input FILE Specify an input file with which to run the model. If no input is specified, the model is run for a single time step of 1000 time units (typically seconds), with all parameters initialised to their default values.

-o FILE, --output FILE Specify an output file for coarse results (see below). If this is not specified, the results are printed to `stdout`.

-d FILE, --detail FILE Specify an output file for detailed results (see below). If this is not specified, detailed results are not printed.

-N, --NaN Initialise all working data (variables and parameters) with NaN values instead of 0. This may be useful in diagnosing problems with the model. (May not work on all platforms.)

As described in §6, a model run is controlled by an input data file, which specifies time steps and parameter changes. For each time step in the input, the RADAU5 solver is invoked. During this invocation the model equation system may be evaluated at many intermediate time points. The

calculated values at these time points are reported in the *detailed* output. The values at the end of the time step, after RADAU5 returns, are reported in the *coarse* output. The two sets of output are sent to different destinations and can be configured by the input file to report different data.

At present, results data are written in tab-delimited text format, usually with a header row (though that can be set by the input file). No internal plotting facilities are provided, nor are they likely to be—that sort of thing is much better handled in R or Matlab. (For very simple cases, the results can be plotted with BGUI.)

If you are running the model via the default Makefile targets, the coarse outputs are saved to the file `build/`*modelname*`.out` and the detailed ones to `build/`*modelname*`.detail`. In addition, the standard output and error streams are redirected to the files `build/`*modelname*`.stdout` and `build/`*modelname*`.stderr`. If the model was built in debug mode (the Makefile default), the latter file may be very large.

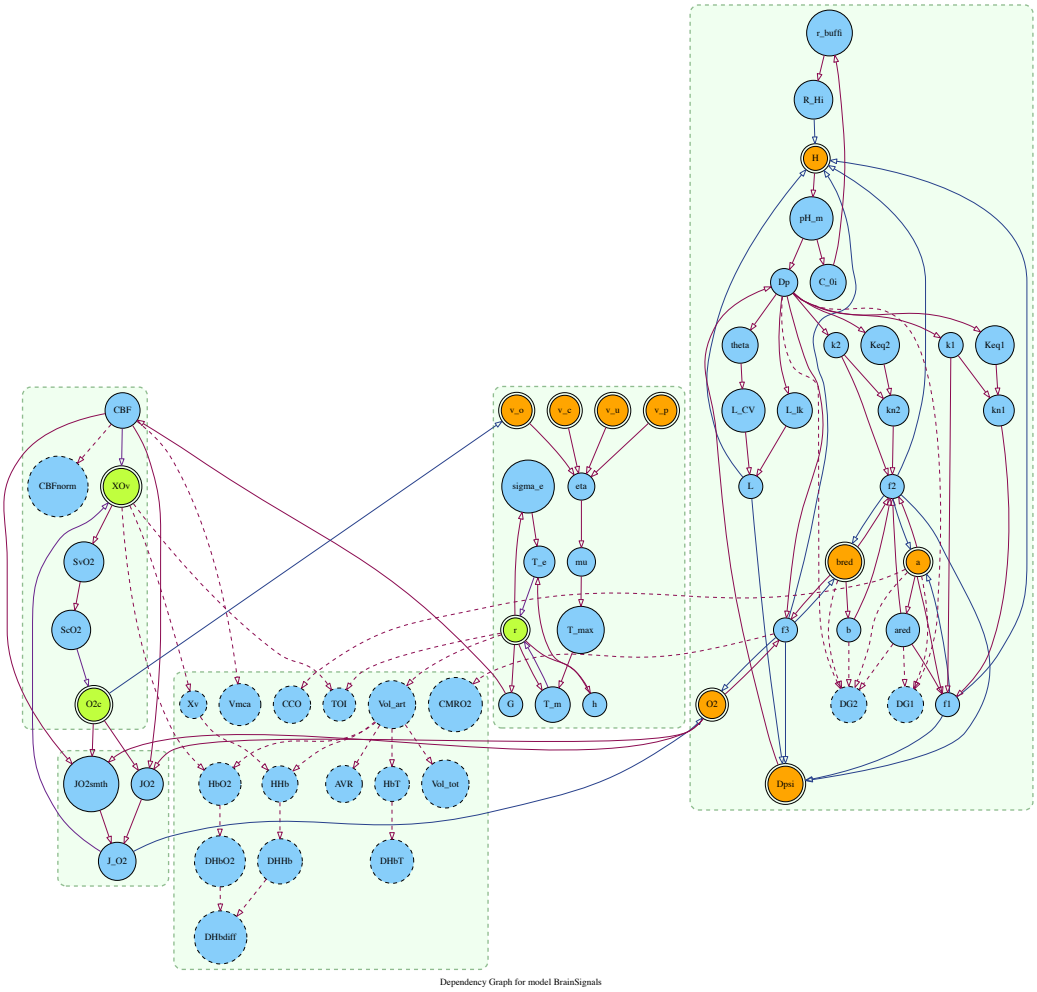

Figure 10: Dependency graph of the BrainSignals model, with parameters omitted. Variables are clustered into subgraphs according to tags in the model.

4.4.3 Info

The compiler tools include a module for extracting some useful information about a model after it has been compiled, generating model documentation and generating graphs of its structure like the

one in Figure 10. This is invoked automatically to print some summary information when compiling, and is also used by BGUI, so it is usually not necessary to call it directly. Nevertheless, it is sometimes useful to do so, running it from the command line like this:

```
python bparser/info.py [options] file
```

The `file` argument should be the name of the `.bcmpl` parser information file generated when a model was compiled (so compilation must have included the `-p` option). Both BGUI and the Makefile place these files in the `build` directory.

The following options are available:

-h, --help Print a usage message and exit.

-d DIR Specify output directory (default: .)

-t [FILE], --text [FILE] Specify output of model documentation in plain text format. FILE is optional, if omitted the output file is called *modelname*`.txt`

-l [FILE], --latex [FILE] Specify output of model documentation in LaTeX format. FILE is optional, if omitted the output file is called *modelname*`.tex`

-H [FILE], --html [FILE] Specify output of model documentation in HTML format. FILE is optional, if omitted the output file is called *modelname*`.html`

-m [FILE], --modeldef [FILE] Specify output of a consolidated model definition. FILE is optional, if omitted the output file is called *modelname*`.modeldef`

-S [FILE], --sbml [FILE] Specify export of model to SBML format. FILE is optional, if omitted the output file is called *modelname*`.xml`

-a Generate all major graph variants, using their default names.

-n NAME, --name NAME Specify a model name instead of deriving it from the file.

-U [FILE], --graphxunused [FILE] Produce graph with unused elements omitted. FILE is optional, if omitted the output file is called *modelname*`_no_unused.gv`

-N [FILE], --graphxinit [FILE] Generate graph with initialisation dependencies omitted. FILE is optional, if omitted the output file is called *modelname*`_no_init.gv`

-P [FILE], --graphxparam [FILE] Generate graph with parameters omitted. FILE is optional, if omitted the output file is called *modelname*`_no_params.gv`

-X [FILE], --graphcore [FILE] Generate graph excluding both initialisation and unused elements. FILE is optional, if omitted the output file is called *modelname*`_core_only.gv`

-C [FILE], --graphxclust [FILE] Output graph without clustering. FILE is optional, if omitted the output file is called *modelname*`_no_clusters.gv`

-F [FILE], --graphfull [FILE] Output graph with everything included. FILE is optional, if omitted the output file is called *modelname*`.gv`

-R, --graphself Include direct circular dependencies in graphs.

-s, --summary Print summary information to `stdout`.

-x, --nograph Suppress all graph output.

-v LEVEL, --verbose LEVEL Set output verbosity (0–7, default: 5)

At present there are no formatting options for the documentation files produced when invoking this module from the command-line. The default formatting is always used. If you wish to apply different LaTeX formatting, you must access these via the BGUI Info panel (§4.2.2).

Most of the graphing options should be reasonably self-explanatory. Initialisation dependencies are those that only occur before the solver is first called, typically to set initial values and calculate derived parameters. Unused variables are intermediates that do not feed back to the behaviour of the model variables, often used for output processes. Parameters are values that have no dependence on model variables. Clusters are boxes that group together symbols sharing the same primary tag (§5.1.1)—if tags have not been assigned, then omitting clusters has no effect.

It is possible to generate more than one graph in a single run, and indeed doing so is necessary to obtain some variations: to avoid having options for every possible combination, there is some interaction between types. For example, if producing both parameterless and clusterless versions, the latter will also exclude parameters.

The graph generation features produce `.gv` files containing a specification of the graph in the GraphViz DOT language. To convert these to actual graphics, you need to have the GraphViz tools installed. These provide a range of different transformation options, for which you'll need to examine the GraphViz documentation. However, the following command provides a reasonable starting point for most purposes:

```
dot -Tpdf -o model.pdf  model.gv
```

This generates a PDF version of the graph described in `model.gv` (substitute the actual name as appropriate).

For the moment, no control is provided over the appearance of the generated graphs—the styling for different element types is as always as shown in Figure 10. This is likely to change in future.

5 Model definition

Models are defined in one or more plain text files. Definitions can contain any of the following elements:

- differential equations
- algebraic equations
- chemical reactions
- constraints
- arbitrary intermediate variables and parameters

In addition, files can contain comments (which begin with a `#` symbol) and some other directives for doing things like importing other files (which begin with an `@` symbol). The elements can be defined in any order and none are compulsory—although obviously a definition that doesn't give rise to a meaningful equation system will not run sensibly (and in some cases might cause compilation to fail, though we hope to catch most such cases eventually).

5.1 Comments and whitespace

Comments allow the inclusion of arbitrary text in a model definition file, to provide human-readable notes and documentation[3]. They begin with a hash character `#`, may start anywhere on a line and then continue to the end of that line. Comments are essentially ignored and in general do not contribute to the model definition (though there is at least one minor exception to this, described below).

Most other model elements must start at the beginning of the line. Conceptually, they occupy a single line in the file, but continuation across multiple lines is allowed using indentation: any line that starts with whitespace characters is considered to be a continuation of the previous line (with any tail comments stripped off). That is, something like this:

```
x = (a                # some comment about a
     + b)             # some comment about b
```

is equivalent to:

```
x = (a + b)
```

The amount of indentation is unimportant, so different lines can be indented differently. This can be useful for keeping track of complex nested parentheses, for example.

Otherwise, whitespace is ignored.

5.1.1 Documentation comments

Comments beginning with a double hash `##` are recognised as a distinct type. Like normal comments, they have no effect on the model itself (with one exception, noted below), but they provide additional information which is used when automatically generating model documentation (§4.2.2 and §4.4.3).

Several special cases of documentation comment, described below, are specified by their first non-comment character. Otherwise, doc comments simply contain text that will be included *almost* verbatim in the generated documentation.

The exact treatment varies with the documentation generator. The HTML generator escapes HTML special characters in the doc comments, so you can use the $<$, $>$ and & characters freely, but you cannot embed links or HTML styling. On the other hand, the LaTeX generator does not escape special characters, so it is possible to embed LaTeX commands, but also quite easy to accidentally break your document with a misplaced underscore or ampersand. Plain alphanumeric text should produce sensible output for all generators, but of course that may not suit your purposes.

An empty doc comment denotes a paragraph break.

Attachment comments

All doc comments are attached to either a model symbol—a parameter or variable—or to the model as a whole. (At present it is not possible to attach comments to an equation or chemical reaction.)

[3]The BCMD modelling language is designed with a philosophy almost diametrically-opposed to that of some other languages, such as SBML. We assume that you know what you want to do, and attempt to let you do it as directly and concisely as possible. Therefore, much of the model structure is implicit. You are not required to specify what everything is or how you expect it to behave, and there is no support for automatic checking or conversion of things like units. Consequently, while it is much easier to do things *at all* in a BCMD model than in a bureaucratic environment like SBML or CellML, it is also quite easy to get things horribly wrong. We recommend structuring your model code as cleanly and intuitively as you can, making liberal use of comments and spacing to aid readability.

Comments are usually expected to be attached to whatever is defined immediately after them in the file:

```
## Some lines of information about the variable x,
## perhaps describing its relationship to a and b.
##
## A second paragraph of information about x.
x = a + b
```

However, it is sometimes inconvenient to position them in this way, and we may also want to attach the current comments to a number of separate entities—this is often the case for *tags* (see below) for example. To allow this, doc comments that begin (after the hash characters themselves and any whitespace) with an at character `@` are taken as specifying one or more symbols to which the most recent comment(s) should be attached. E.g.:

```
## This parameter has no effect on simulation behaviour,
## it is purely for output purposes.
## @ hue saturation brightness
```

Here, the same comment would be attached (separately) to each of the three symbols listed.

An `@` comment with no targets tells the compiler to attach the comments to the model itself rather than to any specific symbol within it.

Note that you can attach comments to symbols that you have not yet defined, but doing so *creates* the symbol—that is, if you attach a comment to a symbol, then that symbol exists in the model, even if you never subsequently define it. It won't actually *do* anything in that case, and will have no material effect on the model. However, we often use counts of variables and parameters as a rough metric of model size, and spurious unused symbols could be misleading in that regard.

Tags

A doc comment beginning (after the hash characters and any whitespace) with a plus sign `+` is considered to define one or more keywords or *tags* to associate with a model symbol. These might be used, for example, for classifying symbols according to the processes in which they participate. Tags are whitespace separated, and should follow the rules for variable identifiers (§5.4.1).

```
## + horizontal mysterious adults_only marks_the_spot
x = a + b
```

Symbols can have any number of tags attached to them, but the *first* one declared for each symbol is considered its 'primary' tag. The graph generator makes special use of this tag: if clustering is enabled (as it is by default), all symbols with the same primary tag are clustered together into the same subgraph. (See Figure 10 for an example.)

Units

A doc comment beginning (after the hash characters and any whitespace) with a tilde character `~` specifies what units the variable or parameter is measured in. This is purely informational, and is not interpreted or used in any way by the model. LaTeX markup is allowed in the units, but will obviously look odd when generating HTML or plain text documentation.

```
## X is the speed of an Arcturan Mega-Donkey's soul
## ~ furlongs/fortnight
```

Name	Default Appearance
`tigers`	*tigers*
`__tigers__`	*tigers*
`tiger_cubs`	$tiger_{cubs}$
`_X_y_z_abc_d_`	$X_{y,z,abc,d}$

Table 1: Examples of the default symbol name rendering by the LaTeX documentation generator.

```
x = a + b
```

LaTeX

A doc comment beginning (yes, you guessed it, after the hash characters and any whitespace) with a dollar sign `$` specifies a LaTeX math mode fragment to be used to represent a symbol in generated LaTeX documentation. The fragment substitutes for the plain text symbol used in the model definition itself. For example, you could do something like this:

```
## A small amount. Less than this we just don't care.
## $ \varepsilon
## @ epsilon
```

Within generated LaTeX documentation, the model symbol `epsilon` would then be rendered as ε rather than *epsilon*. It is not necessary to provide a closing dollar sign, but you may do so if you wish.

If you *don't* specify a LaTeX fragment explicitly, the LaTeX documentation generator gives special treatment to symbol names containing underscores. First, any underscores at the beginning and end of the symbol are removed. Anything before the first internal underscore is treated as the main symbol, and everything after is placed as a subscript. If there are multiple underscore-separated parts to the subscript, these become comma-separated in the subscript. Some examples are shown in Table 1. This rendering may be sufficient to your needs, in which case it is fine to rely on it. Note however, that there is no attempt to ensure uniqueness—if you have symbols in your model that differ only by leading or trailing underscores, they will be rendered identically in the documentation.

5.2 Compiler directives

Compiler directives begin with an at character `@`. At present there are six of these, described below.

5.2.1 @import

This directive tells the compiler to include other files into the current model. The `@import` keyword is followed by one or more model names, delimited by whitespace. E.g.:

```
@import submodel shared_params
```

Note that at present the names must follow the rules for variable identifiers (§5.4.1), and thus cannot include a filename extension. (This will probably change in future.) The compiler searches for these files in the same way as for source files specified on the command line (§4.4.1).

5.2.2 @version

This directive specifies a version number or string to be associated with the model. This is optional, but can be useful for keeping track of changes. If the version is a number, or if it conforms to the variable identifier rules (§5.4.1), then it can follow the directive without quoting. Otherwise surround it in double quotes. E.g.:

```
@version 1.2
@version "1.2 (simplified)"
```

The version string for a compiled model can be queried by running the model with the `--version` option (§4.4.2).

5.2.3 @independent

This directive specifies the name of the independent variable used in the equation system. By default, this is `t`. Note that most of our models do not include an explicit dependence on the independent variable and don't need to refer to it by name, but it may be useful to do so for cosmetic or readability reasons. E.g.:

```
@independent x
```

5.2.4 @output

This directive specifies which variables should be output when no other output specification has been made in the input file, or when the input file specifies `*` (see §6). By default, all solver variables are included in this output.

The `@output` keyword is followed by one or more variable or parameter names, delimited by whitespace. Multiple such directives can be included in a file, and all variables specified in all of them will be output. E.g.:

```
@output x y z
@output v w x
```

Note that the independent variable (by default `t`, but see the `@independent` directive above), is always included as the first column of the default output. Each symbol is only added to the output once, so in the above example the second `x` will be ignored. Symbols that are not actually declared in the model are also ignored.

5.2.5 @input

This directive specifies which variables are intended to be used as model inputs. Note that this exists largely for information purposes and is not binding—you can still set any parameters you want in the input file.

The `@input` keyword is followed by one or more variable or parameter names, delimited by whitespace. Multiple such directives can be included in a file, and all variables specified in all of them will be marked as inputs. E.g.:

```
@input x y z
@input v w x
```

Any symbols not actually declared in the model are ignored.

It is expected that this will be used for additional model checking in the future. However, at present it only really affects the summary information generated by the compiler, and the order in which symbols are displayed in some parts of the BGUI interface.

5.2.6 @extern

This directive identifies variables that are *referenced* in the current model, but which are expected to be *defined* elsewhere. This is really only relevant when models are broken down into several different files that may be combined in different variations. Once again, this is purely informational.

The `@extern` keyword is followed by one or more variable or parameter names, delimited by whitespace. Multiple such directives can be included in a file, and all variables specified in all of them will be marked as potentially external. E.g.:

```
@extern x y z
@extern v w x
```

When the model is compiled, any symbols not actually referenced in the model are ignored. Symbols that are defined in the model (i.e., assigned a value or expression, rather than merely referenced) are also ignored. Whatever remains is treated as an ordinary parameter, but is reported as external in generated graphs and documentation.

The idea here is that when you break a model up into separate parts, you mark important connecting elements as external. If the submodels are compiled together, the external dependencies are satisfied and the symbols become just an ordinary part of the joint model. On the other hand, if you compile the submodels singly or in incomplete combinations, the unsatisfied `@extern` dependencies will be reported.

5.3 Embedded C

Unlike BRAINCIRC, the expressions in a BCMD model definition are not literal C code and do not get directly included in the generated C file. However, it is possible to add C code fragments by surrounding them with the special delimiters `[**` and `**]`. Everything between these will be copied verbatim into the generated file.

Since the resulting code is unchecked, this should be done sparingly, but it can be useful for defining functions that are not included in the standard C libraries. E.g.:

```
[**
/* Simple Heaviside step function. */
double heavi(double s)
{
    return (s < 0.0) ? 0.0 : 1.0;
}
**]
```

Note that arbitrary variable names or `#define` constants created in embedded C code *cannot* be directly accessed from within BCMD expressions. Functions to be invoked within BCMD expressions should have a return type of `double`, which should also be the type for any parameters taken.

5.4 Expressions

5.4.1 Variable identifiers

Variable identifiers are placeholders for any numerical quantity in the model, including chemical species, parameters and compartments. Identifiers are made up of alphanumeric characters and underscores, and must not start with a number. There is no explicit length limit. Thus, `x`, `CO2` and `unreasonably_long_parameter_name` are all valid identifiers, but `5HT` and `O2%` are not.

Variables do not need to be declared before use. They are automatically created the first time they are used, and their type is determined from context. If they appear in several different equations with conflicting requirements, an error will be reported.

A variable that is used but never given a value in the model definition is automatically initialised with the value `0`. All variable values may be overridden at runtime.

Note that BCMD variable names *do not* map to C variable names. You cannot access them directly from embedded C code.

5.4.2 Numbers

All numbers in BCMD are ultimately handled in double-precision floating point form. It is not necessary to include a decimal point for type distinction purposes. Numbers can be specified in 'engineering' notation if desired. The following are all valid numbers:

```
9
-1.0
1.2e-5
2E6
```

5.4.3 Mathematical operators

Arithmetic is performed with the usual operators: `+`, `-`, `*` and `/`, while `^` signifies exponentiation. Addition and subtraction have lower precedence than multiplication and division, exponentiation has higher, and all these operators associate to the left. Precedence can be overridden using parentheses, e.g.:

```
(a + b)/c
```

5.4.4 Comparisons and conditionals

Comparisons between numerical values are made using the following operators `==`, `!=`, `>`, `>=`, `<` and `<=`, which have their conventional interpretation. The result of a comparison is a logical (true or false) value; unlike in C, these cannot be freely mixed with numerical values. They have meaning only in the context of the *conditional operator*, `?:`, which is equivalent to an *if ... else ...* construct:

```
(a == b) ? c : d
```

In this example, the whole expression takes the (numerical) value `c` if the test (`a==b`) is true, otherwise it takes the value `d`.

5.4.5 Function invocation

A function call is denoted with parentheses in the usual way: `function(arg1, arg2, ...)`. All functions are assumed to return a numerical value and take numerical arguments. Because of this, you cannot use a function call on its own in the condition term of a conditional operator expression:

```
# this is not allowed
sometest(a) ? b : c

# but this is
sometest(a) != 0 ? b : c
```

All the functions of the C standard maths library (from `<math.h>`) are included by default.

5.5 Equations

Differential, algebraic and arbitrary temporary variable equations have similar forms. All use the single equals character = to declare the relation. The right hand side in all cases is a mathematical expression, of arbitrary complexity, plus an optional double-quoted label. The left hand side determines the type of the equation.

The following are syntactically valid examples of each of the three types:

```
temp = a + b * c / sqrt(d)          "intermediate"
diff' = temp * diff                 "differential"
alg : 0 = 2 * alg - diff            "algebraic"
```

The quoted labels are effectively cosmetic, included primarily for consistency with BRAINCIRC. They can safely be omitted, and in most cases probably should; use comments instead.

The different forms of the left hand side are explained below.

5.5.1 Intermediate variables

Intermediate or temporary variables are the simplest kind of equation. They specify an expression that is used within the model, or that you might want to output when simulating, but without declaring a variable for which the system is to be solved (i.e., a member of the **y** vector in Equation 1). The right hand side in this case is simply the name (identifier) to be used to refer to the expression:

```
name = expression
```

See §5.9 for a discussion of when intermediate variables get evaluated.

5.5.2 Differential equations

A differential equation defines one row of the general matrix equation in Equation 1. This means that it declares both a variable in the **y** vector, and also a row in the mass matrix **M**. The former is done by the first term on the left hand side; the second by that taken together with any further left hand terms. In the simplest (and thankfully most common) case, there is only the one term, and the expression will look like this:

```
name' = expression
```

(Note the apostrophe character to indicate differentiation.)

More complex cases specify linear relationships between the differential terms, corresponding to off-diagonal elements in **M**. They are specified like this:

```
name' + w1 name1' + w2 name2' + ... = expression
```

i.e., as a weighted sum of differential terms on the left hand side. The weight of the first term is always implicitly 1 and must not be included. The other weights must (for the moment at least) be numeric literals (and can be omitted if 1). For negative terms, use a - operator instead of +. E.g.:

```
u' + 2 v' - w' = u + x / y
```

All left hand side terms must be differentials, and the secondary ones must each also occur elsewhere in the model as the primary term in a differential equation in their own right (otherwise **M** cannot be constructed).

5.5.3 Algebraic equations

Like differential equations, algebraic equations define a row of the general matrix equation in Equation 1, but in this case the corresponding row of **M** is set to 0. Thus the natural form of an algebraic relation would be like this:

```
0 = expression
```

However, it is also necessary for the equation to declare a variable in **y**. Since in many cases this will be ambiguous, we do not attempt to have the parser guess the variable from the expression, but instead require it to be specified explicitly on the left hand side, in this form:

```
name : 0 = expression
```

In some cases it is more intuitive to represent the equation part in the form `expr1 = expr2` rather than `0 = expr2 - expr1`, so this is also supported. E.g.:

```
T_v1 : T_v1 = sigma_v1 * h_1
```

5.6 Initial values

Initial values for variables and parameters may be specified by using a Pascal-style operator `:=` for assignment:

```
name := expression
```

Apart from the change of symbol, the syntax is identical to that for temporary variables, with arbitrary expressions, function calls etc allowed on the right hand side, as in this example from the Ursino-Lodi model definition:

```
T_m1n := T_max1n*exp(-pow(fabs((r_1n-r_m1)/(r_t1-r_m1)), n_m1))
```

Initialisation differs from intermediate variable definition in that the expression is *only* evaluated at the beginning of the simulation, *even if it includes dependencies on other parameters or solver variables*. See §5.9 for a full discussion of this.

5.7 Constraints

Constraints represent simple bounds on the values taken by a variable. There are nominally two forms, *hard* and *soft*, although the latter are not yet properly implemented in the model code.

5.7.1 Hard constraints

Hard constraints are declared like this:

```
variable > expression
```

(the operators <, <= and >= can also be used). Note that this is a declaration of the desired state rather than the test for violation as in BRAINCIRC. I.e., the above statement asserts that `variable` must always exceed `expression`. The constraint need not be constant.

The constraint is 'hard' in the sense that is enforced: if the variable goes across the barrier value, it is set back to it. (It should be noted that this means the strict constraints < and > are not properly observed, since we can't meaningfully set the variable value to an infinitesimal offset from the boundary.)

By far the most common constraint is non-negativity (`variable >= 0`). This is automatically imposed for all species in chemical equations, so does not need to explicitly declared in such cases.

5.7.2 Soft constraints

Soft constraints are declared like this:

```
~ variable > expression
```

i.e., the same as hard constraints but prefixed with a tilde character ~.

Soft constraints are not enforced, and at present there is no way of outputting them, so they basically do nothing. The intention is that violations will be noted and it will be possible to output them and potentially act accordingly (e.g., by prematurely halting a simulation run). This is not currently at the top of the 'to do' list.

5.8 Chemical reactions

Reactions are declared in a sort of caricatured chemical equation form. Reactant and product concentrations are essentially just normal variables, but with some additional notation to distinguish the context. Each reaction participant is enclosed in square brackets, like this:

```
[chemical]
```

Optionally, the chemical may be associated with a particular compartment:

```
[chemical, compartment]
```

Both elements (if present) must be single identifiers. If a compartment is present, it is assumed to refer to a variable or parameter that holds the compartment *volume*, so that the actual chemical concentration in the compartment is $chemical \div compartment$. (When no compartment is given, that is equivalent to one of unit volume.)

It is important to note that a chemical present in two different compartments is actually considered as two different chemicals. If there is a flux between the compartments, this should be specified

as a reaction. If you need to refer to the species quantity in a mathematical expression other than a chemical reaction, the implicitly-created species variable name consists of the species name and the compartment name joined by an underscore. Moreover, the value of this variable is an *absolute quantity* rather than a concentration—for the latter you must divide by the volume. E.g.:

```
# entry of O2 into mitochondrial compartment
-> [O2, mit] {J_O2}

# intermediate variable for the mitochondrial O2 concentration
mitO2 = O2_mit / mit
```

(The way this is structured is frankly confusing, and may be made more intuitive in future.)

Reactions are specified with the right arrow operator `->` for a one-way reaction, or the double arrow operator `<->` for two-way (the latter is essentially just a shorthand for two opposing one-way reactions). The terms on each side consist of some number of of participants, each with an optional weight expression, separated by `+` signs. Finally, there are one (for one-way reactions) or two (for two-way) reaction rate expressions in curly braces. E.g.:

```
4 [a, comp1]  <-> 2 [b] + X [c, comp2] {k1} {MA:k2}
```

The weights can be any mathematical expression, although if it's very complex it's probably clearer to make it an intermediate variable. In the one-way form, one or other side can be empty, denoting the supply (left hand empty) or removal (right hand empty) of reactants. E.g.:

```
# a supply reaction from the BrainSignals model
-> [O2, Vol_mit] {J_O2}
```

Each reaction rate may be specified explicitly in full, using any mathematical expression, or it may be prefixed with a label (followed by a colon) indicating a standard form. Currently, two such forms are supported: *mass action*, indicated by the prefix `MA`, and *Michaelis-Menten*, with the prefix `MM`. (If some other unrecognised prefix is used, the rate term is treated as explicit.)

Note that the in the case of two-way reactions, the rate term in each direction is completely distinct. They are not required to be of the same type—if you wish them to be, that type must be specified in both terms.

5.8.1 Mass Action kinetics

For mass action kinetics, the reaction rate is dependent on the concentrations of the reactants. The dependence is assumed linear by default, but you can optionally specify a power law expression for each reactant.[4]

At least one rate expression *must* be supplied, to indicate the proportionality, while additional expressions may be included for the powers. The latter are applied to the the reactant terms in left-to-right order; chemicals for which no power is supplied are assumed to contribute linearly.

E.g., the following reaction with a mass action rate specified as:

```
2 [A] + [B] -> [C] {MA : k, 2}
```

is equivalent to this one where the rate is specified explicitly:

[4]This differs from BRAINCIRC, which would automatically raise each concentration to the power of the stoichiometry. That is a valid approximation in the case of elementary reactions, but does not apply more generally. Since it would be difficult to distinguish between cases automatically—and since such rate laws are always empirical anyway—BCMD requires the power to be specified appropriately.

```
2 [A] + [B] -> [C] {k * A^2 * B}
```

5.8.2 Michaelis-Menten kinetics

For Michaelis-Menten kinetics, the reaction is assumed to occur via the formation of an intermediate complex with an enzyme, present at low concentration. The enzyme is not consumed and is not included explicitly in the chemical equation, but instead enters via a limiting rate value V_{max} and complex breakdown/formation ratios for each substrate, K_{m1}, K_{m2}, ... These values must be supplied in the rate term definition, starting with V_{max} and then the K_m expressions in the same order as the reactants appear on the relevant side of the reaction:

```
n1 [S1] + n2 [S2] -> [P] {MM : Vmax, Km1, Km2}
```

There must be the same number of K_m expressions as substrates.

The above reaction could also be defined in explicit form, but it would be quite unwieldy:

```
n1 [S1] + n2 [S2] -> [P]
      {(Vmax * S1^n1 * S2^n2) / ((Km1^n1 + S1^n1) * (Km2^n2 + S2^n2))}
```

5.9 Evaluation

In general, variables are evaluated according to their dependencies. Those that depend on the solver variables get reevaluated at every step. Those that only depend on parameters get reevaluated when parameters are changed. Those with no dependencies at all (i.e., which have been assigned a constant numeric value in the model) are never reevaluated, although they may of course still be changed at runtime by direct assignment in the input file.

In many cases, this is sufficient to produce the correct behaviour. However, there are occasions when you want to specify an initial value that depends on other variables—as a first guess—without tying it to changes in those variables later. This is why we have a distinct initialisation operator, `:=`, that allows a complex expression to be evaluated only at startup, while some other expression—or none at all—comes into play subsequently. It is *possible* to specify both initialisation and general evaluation expressions with the = operator, in which case the parser will try to guess which is which, but this is generally a bad idea.

Note that, because of the updating rules above, there is no practical difference between the two operators for constant assignments. But it is suggested that in those cases the `:=` be used, as it usually better expresses the model intent.

6 Input specifications

The course of a simulation is controlled by an input file, which specifies the simulation time steps and parameter changes, and can also optionally reconfigure the outputs. The input file format is simple, fairly concise and not at all friendly, designed for relatively easy parsing rather than easy writing. The parser is also pretty unforgiving.

The ostensible benefit of this unification is that everything you might ever want to do with a model is done in one place and in a consistent way, rather than having all sorts of special cases littered around the place. However, that doesn't necessarily mean it's easier to use. *Au contraire, mon frère*—at least for some things it is undoubtedly harder.

In future there will hopefully be more sophisticated tools to make creating and managing these files a breeze. The BGUI interface (§4.2) provides some help in this direction, though the functionality is not yet complete. There is also a rather basic command-line script, `steps.py`, that will check and attempt to fix BCMD input files, and will also translate BRAINCIRC input and parameter value files into the BCMD format. (See §6.2 below.)

6.1 File format

Input files are plain text and line-oriented. The first character of each line defines its function. The rest of the line is usually a whitespace-separated list of tokens that are interpreted according to the line type.

6.1.1 Comments

As with the model definition files, comments are denoted with a hash character `#`. Comment lines are discarded without further ado. Comments should start at the beginning of the line—tail comments are not supported; while they may accidentally work in some cases, you should not rely on this.

6.1.2 Header

The first non-comment line in the file must be a header line, indicated with an at sign `@`. It contains only one token, which must be a positive integer specifying the number of time steps in the simulation:

```
@ 10
```

This can be slightly fiddly to work out by hand. Fortunately the `steps.py` script will fill this in automatically if it's absent or incorrect.

If the input file does not define at least the specified number of steps, the model will exit with an error. If it contains more, the excess steps will be skipped.

6.1.3 Output control

Lines beginning with a greater than character `>` specify what variables should be output, while those beginning with an exclamation mark `!` control the output of a header row containing the variable names. The number of repetitions of the character determines the streams affected: if the row starts with a single instance of the `>` or `!` character, it applies to the coarse output stream; if there are two, it applies instead to the detailed stream; if there are three it applies to both. E.g.:

```
# enable header output to detail file
!!
```

Because headers are enabled by default, there is an additional control sequence, `!0`, to switch off headers on both streams.

Note that although headers are typically only used at the start of a file, it is possible to specify headers at a later point. The header is printed only at the beginning of the first time step after enabling.

For output specifications, the token after the `>` sequence must either be an asterisk, `*`, or an integer specifying the number of output fields. In the first case, the model's default output fields are used (see §5.2.4). Otherwise, the remaining tokens list the model variables and parameters to be output:

```
# specify three variables to be written to both outputs
>>> 3 t x lambda
```

If the output field count is 0, all output is suppressed. This can be useful for running the model for awhile to eliminate transients—the `steps.py` script does this for resets and steadying when translating BRAINCIRC input files.

The output specification is allowed to include names that are not present in the model. In that case the corresponding columns are simply not written—there is not a blank column in the file for the missing data.

By default (if no output specifications are included in the input file) the solver variables (t and the $\mathbf{y}$ vector in Equation 1) are output, and headers are written for both the coarse and detailed outputs.

6.1.4 Specifying variable and parameter assigments

Lines beginning with a colon `:` specify what variables and parameters, if any, will be set on subsequent steps. The specification remains in force until another one is given. At least one such line must be included before any actual steps are defined. As with the output specification, the first token after the `:` is an integer specifying the number of fields that will be set, and subsequent tokens are the names of those fields:

```
# set these parameters until further notice
: 3 a b c
```

The count can be 0, in which case the steps are run without making any changes. As with the output specification, the list can include names that do not appear in the model. The corresponding values are then simply ignored.

6.1.5 Absolute steps

Lines beginning with an equals sign `=` specify an absolute time step. All subsequent tokens on the line must be numbers. The first two specify the start and end times of the step, and the remaining ones specify the values to be assigned to whatever variables and parameters were configured by the most recent `:` line. There must be valid numeric values for all fields, even ones that are not in the model and hence won't really be used.

```
# a single absolute time step setting 3 params
= 0 100 0.1 -50 1e-4
```

RADAU5 doesn't like steps of zero duration, but is ok with steps backward.

As a special case, if the start time and end time are both 0, the parameter setting occurs as normal, but no attempt is made to run the model—we just proceed to the next step. This is particularly useful for initialisation.

6.1.6 Relative steps

Lines beginning with a plus sign `+` specify a relative time step. These are basically the same as absolute steps, but instead of start and end times they have only a single value, the step duration:

```
# a single relative time step setting 3 params
+ 100 0.1 -50 1e-4
```

If there has been no previous time step in the file, the start time defaults to 0. In that situation, this relative step and the earlier absolute version would be functionally equivalent. Otherwise, each relative step starts where the previous one finished.

6.1.7 Multiple steps

Lines beginning with an asterisk character * define a step that will occur multiple times. In this case, both the time steps and value changes are relative. The first token after the * must be a positive integer, specifying the number of repetitions. The second token specifies the duration of each step. The remaining tokens specify incremental changes to the prevailing set of variables and parameters:

```
# do this 100 times
* 100 1 0.001 -0.5 1e-6
```

The final values of this set of steps would be the same as the previous single relative step example, but they would have been reached in 100 small increments.

If no previous values have been set in the input, the start time *and all initial field values* default to 0. This is probably not what you want. Moreover, there is currently a potential crashing issue if you specify a multiple time step immediately after changing the fields being set. It is therefore advisable to always have at least one = or + step before starting a multiple sequence.

6.1.8 Unrecognised lines

Lines beginning with other characters than those detailed above are, at present, simply ignored. It probably isn't a good idea to rely on this behaviour.

6.2 `steps.py`

The `steps.py` script (found in the `batch` directory) is not very sophisticated, but it can help with the production of input files for use with BCMD models. It will parse BCMD input files, report any errors it finds and attempt to write out a 'fixed' version of the file. It will also parse BRAINCIRC input and parameter values files, and attempt to generate a corresponding BCMD input file. (It may well fail if it encounters things that are not properly described in the BRAINCIRC documentation.)

Its interface is pretty much non-existent. Invoke python on it from the command line, passing the names of the relevant BRAINCIRC (`.dat`) and BCMD files (`.input`). E.g.:

```
python steps.py file1.dat file2.input file3.dat
```

It will print errors and warnings to the standard error stream, and print a valid BCMD input file containing all the steps in all the supplied files, in the order you list them, to standard out. Typically you will want to redirect the latter to a new input file:

```
python steps.py file1.dat file2.input file3.dat > merged.input
```

BRAINCIRC parameter value files generate a single zero-duration (i.e., not run) assignment step that sets all the specified values. Autoregulation input files generate downward and upward multiple-step sequences, and normal input files generate the appropriate sequences of explicit steps. Parameter reset sequences and a steadying step are generated as required.

7 Implementation notes

7.1 RADAU5

Like BRAINCIRC, BCMD makes use of the RADAU5 solver library (Hairer and Wanner 1996). This solver layer is reasonably well separated and a different engine could be substituted if necessary; however, while there are numerous other general ODE solvers available, not many support differential-algebraic systems of the kind used in our models—RADAU5 appears still to be the most common underlying implementation for such systems. Replacing RADAU5 would likely require restricting model definitions to pure ODE form, which would be rather limiting for our purposes. But this might be added as a compiler option in future, so that models that do not need the DAE features can be built with a different back end.

7.2 Code structure

The Python code for the BCMD compiler is contained within the `bparser` directory, and is organised as follows:

- The underlying LALR parsing is performed by the PLY package, the source for which is in the `ply` subdirectory.
- `bcmd_lex.py` defines the patterns used for lexical analysis, while `bcmd_yacc.py` defines the language grammar and provides functions to convert the parser productions into a very basic *abstract syntax tree* (AST) made of Python tuples.
- `ast.py` walks the AST, gathering a pile of information about the structure of the model, doing some validation, determining dependency order and transforming chemical reactions into ODEs. Most of the 'expert knowledge' in the system, to whatever extent there is such a thing, is contained in this file, with the result that some of it is a bit opaque, notably the parts relating to reaction rates.
- `codegen.py` uses the results of the previous analysis to generate the C code for the model executable. A significant amount of this C code is unchanging boilerplate, and most of that is piped wholesale from the template files in the `templates` subdirectory.
- `info.py` uses the results of the AST analysis to log some useful information about the model structure and to generate a GraphViz representation of it.
- `logger.py` provides utility functions for writing log messages with different levels of urgency.
- `bcmd.py` is the interface program, which invokes the above components according to the arguments passed it on the command line.
- The various `doc_*.py` modules implement export of model information to various file formats.

SBML support, such as it is, is implemented with the aid of libSBML (Bornstein et al. 2008).

The generated C code is a simple non-interactive command line program of a pretty standard form: process command line arguments, set up the model environment, run it, exit. There are two sources of complication: managing the simulation time course and communicating with the RADAU5 solver. The latter is somewhat simplified by a wrapper library (in `src/radauwrap`) that deals with the more tedious parts of setting up and calling the Fortran code. In order to conform to the restrictions of the solver interface, we make unfashionable use of global data arrays.

The simulation time course is parsed from an input file (described in §6) by the big ugly function `load_inputs()`, creating a data structure that defines time points, parameter changes and outputs.

The parser is very simplistic, and the file format is designed for easy parsing rather than friendliness, so it is terse and rather brittle. At some point tools will be provided to make it easier to work with.

References

Murad Banaji. *The BRAINCIRC model*, 2005. URL http://www.medphys.ucl.ac.uk/braincirc/index.html.

David M Beazley. PLY: Python Lex-Yacc. 2011. URL http://www.dabeaz.com/ply/.

Benjamin J Bornstein, Sarah M Keating, Akiya Jouraku, and Michael Hucka. LibSBML: an API library for SBML. *Bioinformatics (Oxford, England)*, 24(6):880–881, March 2008. doi: 10.1093/bioinformatics/btn051. URL http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btn051.

Stephen P Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. ISBN 978-0-521-83378-3. URL http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf.

Francesca Campolongo, Jessica Cariboni, and Andrea Saltelli. An effective screening design for sensitivity analysis of large models. *Environmental Modelling & Software*, 22(10):1509–1518, October 2007. doi: 10.1016/j.envsoft.2006.10.004. URL http://linkinghub.elsevier.com/retrieve/pii/S1364815206002805.

A Finney and M Hucka. Systems biology markup language: Level 2 and beyond. *Biochemical Society transactions*, 31(Pt 6):1472–1473, December 2003. URL http://eutils.ncbi.nlm.nih.gov/entrez/eutils/elink.fcgi?dbfrom=pubmed&id=14641091&retmode=ref&cmd=prlinks.

Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. Springer, second revised edition edition, 1996.

M Hucka, A Finney, H M Sauro, H Bolouri, J C Doyle, Hiroaki Kitano, , the rest of the SBML Forum, A P Arkin, B J Bornstein, D Bray, A Cornish-Bowden, A A Cuellar, S Dronov, E D Gilles, M Ginkel, V Gor, I I Goryanin, W J Hedley, T C Hodgman, J H Hofmeyr, P J Hunter, N S Juty, J L Kasberger, A Kremling, U Kummer, N Le Novere, L M Loew, D Lucio, P Mendes, E Minch, E D Mjolsness, Y Nakayama, M R Nelson, P F Nielsen, T Sakurada, J C Schaff, B E Shapiro, T S Shimizu, H D Spence, J Stelling, K Takahashi, M Tomita, J Wagner, and J Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics (Oxford, England)*, 19(4):524–531, March 2003. doi: 10.1093/bioinformatics/btg015. URL http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btg015.

Tracy Moroz. *Computational modelling of brain energy metabolism and circulation in the neonatal animal model*. PhD thesis, UCL, October 2013.

Max D Morris. Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33(2):161–174, 1991. URL http://www.tandfonline.com/doi/abs/10.1080/00401706.1991.10484804.

Felix Scholkmann, Sonja Spichtig, Thomas Muehlemann, and Martin Wolf. How to detect and reduce movement artifacts in near-infrared imaging using moving standard deviation and spline interpolation. *Physiological Measurement*, 31(5):649–662, March 2010. doi: 10.1088/0967-3334/31/5/004. URL http://stacks.iop.org/0967-3334/31/i=5/a=004?key=crossref.663fb075a88b3ede75a7b9080e36220d.

A Ismael F Vaz and Luís N Vicente. PSwarm: A hybrid solver for linearly constrained global derivative-free optimization. *Optimization Methods and Software*, 24(4-5):669–685, 2009. URL http://www.tandfonline.com/doi/abs/10.1080/10556780902909948.