MODULES and how to use the BRAINCIRC environment to construct models.

Contents

1 Module files

1.1 Comment lines

Module files are the most important files as they contain model processes. They will be read by the parser and translated into code for simulation. The most important thing to know about module files – and the interface in general – is that lines beginning with `//` are comment lines and are ignored by the parser.

1.2 The areas in a module file

Currently there are up to fourteen possible areas in a module file. There are described in more detail in the subsections below, but in brief they comprise...

1. An area containing **chemical reactions** which is started with the line:

   ```
   REACTIONS
   ```

   and terminated with the lines

   ```
   endREACTIONS
   ******
   ```

2. An area containing **algebraic relations** which is started with the line:

   ```
   ALGRELS
   ```

   and terminated with the lines:

   ```
   endALGRELS
   ******
   ```

3. An area containing **differential equations** which is started with the line:

   ```
   DIFFEQS
   ```

   and terminated with the lines:

   ```
   endDIFFEQS
   ******
   ```

4. An area containing **terms in differential equations** which is started with the line:

   ```
   DETERMS
   ```

   and terminated with the lines:

   ```
   endDETERMS
   ******
   ```

5. An area containing **templates** which is started with the line:

   ```
   TEMPLATES
   ```

   and terminated with the lines

   ```
   endTEMPLATES
   ******
   ```

6. An area containing **the names of variables** which is started with the line:

   ```
   VARS
   ```

   and terminated with the lines:

   ```
   endVARS
   ******
   ```

7. An area containing **the names of temporary variables** which is started with the line:

   ```
   TEMPVS
   ```

   and terminated with the lines:

   ```
   endTEMPVS
   ******
   ```

8. An area containing **functions used to initialise variables** which is started with the line:

   ```
   INITFUNCS
   ```

   and terminated with the lines:

   ```
   endINITFUNCS
   ******
   ```

9. An area containing **descriptions of variables** which is started with the line:

   ```
   VDESC
   ```

   and terminated with the lines:

   ```
   endVDESC
   ******
   ```

10. An area containing **constraints** which is started with the line:

    ```
    CONSTRAINTS
    ```

    and terminated with the lines:

```
endCONSTRAINTS
******
```

11. An area containing **the names of parameters** which is started with the line:

    ```
    PARAMS
    ```

    and terminated with the lines:

    ```
    endPARAMS
    ******
    ```

12. An area containing **functions used to set parameters** which is started with the line:

    ```
    PARAMFUNCS
    ```

    and terminated with the lines:

    ```
    endPARAMFUNCS
    ******
    ```

13. An area containing **descriptions of parameters** which is started with the line:

    ```
    PDESC
    ```

    and terminated with the lines:

    ```
    endPDESC
    ******
    ```

14. An area containing **descriptions of processes** which is started with the line:

    ```
    PROCDESC
    ```

    and terminated with the lines:

    ```
    endPROCDESC
    ******
    ```

Any number of these areas may be empty or may not exist in a particular model file. For example a system may contain no explicitly defined differential equations with all ODEs constructed automatically from chemical reactions. The details of what goes in the different areas are now presented.

1.3 Differential equations

Differential equations must have a name, a key variable, a term on the left hand side, and a term on the right hand side.

For example the differential equation:

$$\frac{\mathrm{d}r_1}{\mathrm{d}t} = \frac{G_1(P_a - P_1) - G_1G_2(P_1 - P_2)/(G_1 + G_2)}{K_{V1}r_1}$$

might be coded as:

```
name: r_1_dyn
keyvar: r_1
lterm: 1.0, r_1
rterm: (G_1*(P_a-P_1) - G_1*G_2/(G_1+G_2)*(P_1-P_2))/K_V1/r_1
******
```

The line `keyvar: r_1` is not strictly necessary. If it is omitted, the first variable listed after lterm will be taken as the key variable.

1.4 Chemical reactions

Reactions can take a variety of forms. A very simple example is the formation and dissociation of carbonic acid:

$$\mathrm{CO_2} \rightleftharpoons \mathrm{HCO_3^-} + \mathrm{H^+}$$

We can represent this as:

```
name: CO2toH1
type: MA2
left: 1.0, _CO2
right: 1.0, _Hy, 1.0, _BiC
rates: k_CHi, k_nCHi
******
```

The reaction is assumed to be a two-way mass action reaction with forward and backward rate constants k_CHi and k_nCHi. Every reaction is assumed to have a left and a right side, one of which may be empty. The line `type: MA2` tells the parser that the reaction is a two-way mass action reaction.

A more involved example of a reaction might be:

```
name: potpump
type: rateterm
comps: 2
left: 1.0, _ATP, 2.0, _eK, 3.0, _Na
compsleft: Vol_exm, Vol_ecs, Vol_exm
right: 1.0, _ADP, 1.0, _Phos, 2.0, _K, 3.0, _eNa
compsright: Vol_exm, Vol_exm, Vol_exm, Vol_ecs
rates: Vmax_KATP
rateterm: _ATP/(Km_KATPA + _ATP)*(_eK/_K)/(Km_KATPK +
(_eK/_K))*(_Na/_eNa)/(Km_KATPNa + (_Na/_eNa))
******
```

which represents the action of Na^+–K^+–ATPase:

$$\mathrm{ATP_{ic}} + 2\mathrm{K^+_{ec}} + 3\mathrm{Na^+_{ic}} \longrightarrow \mathrm{ADP_{ic}} + \mathrm{P_{i,ic}} + 2\mathrm{K^+_{ic}} + 3\mathrm{Na^+_{ec}}$$

Here there are chemicals involved in two different compartments (ic for intracellular and ec for extracellular), hence the "comps", "compsleft" and "compsright" terms. The "comps" term tells the parser that there are quantities from two compartments taking part in the reaction. The actual compartments in which the quantities reside are listed in the same order as the quantities themselves in "compsleft" and "compsright". `Vol_exm` `Vol_ecs`, etc are the volumes of the compartments in which that chemical resides.

The rate term is supplied, hence the reaction is of type "rateterm". This is the default type for reactions, and hence the line `type: rateterm` could be omitted.

Thus the dynamics generated for extracellular potassium would be:

$$\frac{\mathrm{d}[\mathrm{K^+_{ec}}]}{\mathrm{d}t} = \frac{\mathrm{Vmax_{KATP}}}{Vol_{ecs}} \left(\frac{[\mathrm{ATP}]}{\mathrm{Km_{KATPA}} + [\mathrm{ATP}]} \right) \left(\frac{\frac{[\mathrm{K^+}]_{\mathrm{ec}}}{[\mathrm{K^+}]}}{\mathrm{Km_{KATPK}} + \frac{[\mathrm{K^+}]_{\mathrm{ec}}}{[\mathrm{K^+}]}} \right) \left(\frac{\frac{[\mathrm{Na^+}]}{[\mathrm{Na^+}]_{\mathrm{ec}}}}{\mathrm{Km_{KATPNa}} + \frac{[\mathrm{Na^+}]}{[\mathrm{Na^+}]_{\mathrm{ec}}}} \right)$$

where Vol_{ecs} is the volume of the extracellular compartment, and all quantities are intracellular unless stated otherwise. Note that in this particular example, the parameter $\mathrm{Vmax_{KATP}}$ is listed explicitly in the reaction description. This is useful but not essential. Exactly the same reaction could have been written:

```
name: potpump
type: simp
comps: 2
left: 1.0, _ATP, 2.0, _eK, 3.0, _Na
compsleft: Vol_exm, Vol_ecs, Vol_exm
right: 1.0, _ADP, 1.0, _Phos, 2.0, _K, 3.0, _eNa
compsright: Vol_exm, Vol_exm, Vol_exm, Vol_ecs
rateterm: Vmax_KATP*_ATP/(Km_KATPA + _ATP)*(_eK/_K)/(Km_KATPK +
(_eK/_K))*(_Na/_eNa)/(Km_KATPNa + (_Na/_eNa))
******
```

Note that the reaction is now of type "simp" (also called "ratetermsimp").

There are a variety of other common kinds of chemical reactions which can be represented including for example Michaelis-Menten reactions. However all reactions can be represented using the reaction type simp (or rateterm), which gives the user maximum control.

1.5 A complete list of reaction types

Apart from some outdated reaction types, the following types of reaction are understood by the interface. In every case, the "comps" fields are optional - i.e. all reaction types are allowed to contain reactants in different compartments. When a reaction does contain reactants in different compartments, the rate term is assumed to refer to changes in absolute quantities, rather than concentrations. Thus when the differential equations for concentrations are being constructed, these rate terms will be divided by the volumes of compartments in which the quantities reside, to give changes in concentrations. Note that if the reaction does not take place across different compartments, then the rate terms simply refers to the changes in concentrations.

1. **simp (also called ratetermsimp):** The essential for this reaction is a rate term in the "rateterm:" field. There must be no rate constants apart from this rate term.
2. **rateterm:** The essentials for this reaction are a rate term in the "rateterm:" field, and one rate in the "rates:" field. This rate is assumed to multiply the entire rate term. (This form is useful, because in many reactions there is a multiplying term which represents the total quantity of enzyme present – e.g. the Vmax term in Michaelis-Menten reations – and sometimes we might want to access the term divided by this quantity.)
3. **MA1 (One way mass action):** The essentials for this reaction are a single rate in the "rates:" field (let's say $\mathbf{k}_1$) and at least one chemical (let's say $\mathbf{X}$) either on the left or the right. If there are only chemicals on the right hand side then the following dynamics are generated:
$$\dot{\mathbf{X}} = \mathbf{k}_1 \ \cdots \tag{1}$$
i.e. the rate is assumed to be constant supply rate.
4. **MA2 (Two way mass action):** The essentials for this reaction are two rates in the "rates:" field (let's say $\mathbf{k}_1$ and $\mathbf{k}_{-1}$) and at least one chemical (let's say $\mathbf{X}$) either on the left or the right. The rate constants must be listed in the order "left-to-right" first and then "right-to-left". Behaviour is as for **MA1**.
5. **MM1 (Michaelis-Menten with one substrate):** Michaelis-Menten reactions are assumed to be one-way. For this reaction type there must be only one substrate (on the left) and any number of products (on the right). There must be two rates in the "rates:" field. The Vmax must be listed first and then the Km.
6. **MM1sub2 (Michaelis-Menten with two substrates):** As above, except for this reaction type there must be exactly two substrates (on the left) and three rates in the "rates:" field. The Vmax must be listed first and then the Km for the first substrate and then the Km for the second substrate.
7. **MM1sub**n **(**$n = 3, 4, 5$ **Michaelis-Menten with** n **substrates):** The obvious generalisation to 3, 4 and 5 substrates. The most important thing is to ensure that the Vmax is listed first, and then the Kms are listed in the same order as the substrates are listed.

1.6 Algebraic relations

Algebraic relations are simple to represent. There are three entries - the name of the relation, the key variable, and the equation itself. Of course the choice of "key" variable in a general equation is quite arbitrary, but

this is needed for the purposes of confirming that the full set of equations can be solved. An example might be:

```
name: x_rel
keyvar: x
eqn: x*x + y*y - 4.0
******
```

which represents the equation:

$$x^2 + y^2 = 4$$

1.7 Terms in differential equations

The parser constructs differential equations from terms in differential equations which we abbreviate to "DEterms". Occasionally it might be necessary to specify a term which arises from some process explicitly. The syntax for doing this is simple and similar to that for an algebraic relation. For example:

```
name: Yac
keyvar: Y
term: min(2.0*q*(Y_a - Y), q*Y_a)/Vol_c
******
```

tells the parser to add the term $\min\{2q(Y_a - Y), qY_a\}/Vol_c$ to the differential equation describing the evolution of Y.

1.8 Templates

Templates are rules for constructing processes, and are currently able to construct chemical reactions and terms in differential equations. An example of the latter kind of template might be

```
name: bloodvars
addstr: ac
vars: _var1, _var2
keyvar: _var1
term: min(2.0*q*(_var2 - _var1), q*_var2)/Vol_c
******
```

Here the template is called "bloodvars" and takes two variables (templates can take up to 100 input variables), here called `_var1` and `_var2` (when defining templates it is a good idea to give these variables unusual names, so that there are no possible conflicts with the names of model variables.) Now suppose that in the model descriptor file for this model, in the section `TRANSVARS` (see below), there is a line of the form:

```
bloodvars: x, y
```

This will tell the environment to construct a DE term

```
name: xac
keyvar: x
term: min(2.0*q*(y - x), q*y)/Vol_c
******
```

The field `addstr:` is functionally unimportant and only tells the environment how to construct the name of the DE term. The field `keyvar:` simply translates. The convenience of templates is that if a number of variables undergo very similar processes, this can be captured with a single line in the model descriptor file. For example the line:

```
bloodvars: x1, y1, x2, y2, x3, y3, x4, y4, ...
```

tells the environment to construct DE terms for each pair of variables.

An example of a template for a reaction might be

```
name: outflow
category: reaction
addstr: out
vars: _var1
type: MA1
twoway: 1
left: 1.0, _var1
right:
rates: k1
******
```

A line of the form

```
outflow: A1
```

in the TRANSVARS area of the model descriptor file would lead to the construction of the reaction:

```
name: A1out
type: MA1
twoway: 1
left: 1.0, A1
right:
rates: k1
******
```

1.9 Variables and their initialisation

The parser computes which quantities are variables and which are parameters in the model. However if a quantity is found to be a variable, then it needs to be initialised. Variables can be initialised in three ways - either with a number, or with a reference to some existing parameters, or with a function which describes how the intitial value is to be computed. The first way is not a good idea. The second and third possibilities are essentially equivalent. The three possibilities might look like:

```
gCa1    1.0
gCa1    2.0*gCa1n
gCa1    gCa1_fcn
```

If the variable is initialised in the second way, then there must be a parameter called `gCa1n`. In the example shown `gCa1` is initialised to twice the value of `gCa1n`. If the variable is initialised in the third way, then somewhere there must be the function `gCa1_fcn` for example:

```
double gCa1_fcn()
{
  return gCa1_init;
}
//////////
```

Usually the third way is used if you want to use some complicated initialisation function. If you wish to have control over the initial values of variables without editing the module file, then you should initialise variables in the second or third ways – by initialising with reference to some parameters (e.g. `gCa1_init`).

1.10 Temporary variables

Temporary variables are useful because they allow you to code up a model in much the same way that you might find it in a paper. For example instead of writing:

```
rterm: k_E*P_ic*(2.0*G_1*(P_a-P_1) + ...
```

as the right hand side of a differential equation, we could write:

```
rterm: k_E*P_ic*(I1 + ...
```

as long as we had defined `I1` as a temporary variable

```
I1   2.0*G_1*(P_a-P_1)
```

in the temporary variable area.

1.11 Parameters

The parser computes which quantities are variables and which are parameters in the model. If a quantity is found to be a parameter, then the parser needs to know how it's value will be set. Exactly as with variable initialisation, a parameter can either have a set value or have a value determined by a function. There are three possibilities illustrated which might look like:

```
k0      1.0
k0      sqrt(k1)*exp(k2)
k0      k0_fcn
```

If a parameter is defined in a module file in the first way, then this does not mean that it's value is set to the value in the module file. The fact that there is a number simply tells the parser to expect a number. The actual value of the parameter will be set from a parameter value file when a simulation is run.

If a parameter is defined in the second way, then the value of the parameter is a function of some other parameters. (These, in turn, may be given numerical values or set with reference to other parameters. Of course, you have to make sure there is no recursion in such chains of definition.)

If a parameter is defined in the third way, then we expect a function - for example:

```
double k0_fcn()
{
  return sqrt(k1)*exp(k2);
}

//////////
```

in some module file.

It is fine for a quantity to be defined as both a parameter and a variable, as the parser will decide which definition to use. This feature exists because the interface has been designed to make putting together submodels easy.

1.12 Constraints

Constraints are essentially an error checking mechanism. A constraint consists of a variable name, a condition and an action to be carried out when the condition is met. An example of a constraint might be:

```
gKpot1: gKpot1 < 0.0: gKpot1 = 0.0
```

This constraint tells the integrator that the variable `gKpot1` must never go below zero. If it does then it must be reset to zero. Such constraints can lead to warnings during integration, when variables temporarily cross into disallowed regions. These can often be ignored, but if a model stops running, then they might tell you why. All quantities which figure in chemical reactions are automatically constrained to be positive.

1.13 Descriptions

Module files can also store descriptions of parameters, variables, and processes. A description of a parameter might look like:

```
V_Ca
a parameter in the relationship between membrane potential and the
probability of a calcium channel being open
latex: V_{Ca}
units: mV
******
```

A description of a variable might look like:

```
gCa1
conductivity of calcium channels in the proximal VSM segment
latex: gCa_1
units: $\chem{mM\cdot s^{-1}\cdot mV^{-1}}$
******
```

A description of a process might look like:

```
gCa1rel
How the conductivity of calcium channels in the proximal VSM segment
depends on the membrane potential and pH
name: calcium channel conductivity in the proximal VSM segment
site: VSM membrane in proximal segment
******
```

Descriptions are not strictly necessary to a model. Latex names are only needed if latex documents are to be generated from the module files. They may be left empty. Units may also be left empty.

1.14 General considerations

1. If a differential equation has X as its key variable, then X cannot be a key variable in any other process. For example it cannot feature as a substrate in a reaction.
2. If an algebraic relation has X as its key variable, then X cannot be the key variable in any other process. For example it cannot feature as a substrate in a reaction.
3. A variable may occur in any number of chemical reactions and occur as the key variable in any number of DEterms, as long as it satisfies conditions 1) and 2) above.
4. It is quite irrelevant which module file different parts of a model reside in as long as they reside in module files in the same directory. However it is a good idea to have related quantities in the same module file.

2 Model descriptor files

Model descriptor files are called `models/modeldescs/<model_name>/alldata.dat` and essentially contain lists of all processes which are to be incorporated into the model in question. They contain the following areas:

1. **Algebraic relations**. For example the area

   ```
   ******** ALGRELS ********

   qrel
   ******** endALGRELS ********
   ```

   tells the parser that the model contains a single algebraic relation called `qrel`. The parser will look for a relation by this name in the module files and includes it in the model.

2. **Differential equations**. For example the area

   ```
   ******** DEQS ********

   r_1_dyn
   r_2_dyn
   ******** endDEQS ********
   ```

   tells the parser that the model contains two differential equations called `r_1_dyn` and `r_2_dyn`. The parser will look for two differential equations of these names in the module files and include them in the model.

3. **Terms in differential equations** of the form:

   ```
   ******** DETERMS ********

   ******** endDETERMS ********
   ```

   The principle is the same as for algebraic relations and differential equations.

4. **Chemical reactions**.

   ```
   ******** REACS ********

   ******** endREACS ********
   ```

   Again the principle is the same – the names of all chemical reactions to be included in the model must be placed in this area.

5. **An output parameter and variable area**

   ```
   ******** OUTPUT_PARAMS ********

   output_params:
   output_vars:

   ******** endOUTPUT_PARAMS ********
   ```

This area is an anachronism and can be ignored for new models.

6. **Templates (transport variables)**

```
******** TRANSVARS ********


******** endTRANSVARS ********
```

This area is used to construct DEterms en masse (see the section on templates above). It is useful when a number of variables undergo very similar processes. Historically it has been used for representing convective transport, hence the name.

7. **Shared module files**. This area is important if your model is going to use processes which belong to another model. For example the area:

```
******** MODFILES ********

adenkin.dt
ECdiffBiC.dt
******** endMODFILES ********
```

tells the parser that the model uses module files `adenkin.dt` and `ECdiffBiC.dt` which live in the `models/modules` directory (and are presumably also used by other modules). This is particularly useful when constructing submodels and models which share components with other models. Basically any module file written for a particular model can be used by any other in this way. But you should take care to remember that a file is shared. If the file lives in the `models/modules` directory, it is assumed to be shared. It is in general recommended (though not necessary) to keep shared module files in a directory different from any model's native module directory.

3 Parameter value files

Parameter value files are files which simply store a list of the values of all settable parameters. A complete file for a short model might look like:

```
// model parameters
xx_init 1.0
yy_init 1.0
zz_init 1.0
sigma 10.0
rho 28.0
beta 2.6667
```

(This is in fact the parameter value file for the lorentz model). Note that this file *must* contain the values of all settable parameters – any values given in module files are only used by the parser to ascertain that the parameter is a settable parameter. In addition, the file *may* contain values of other parameters which are not parameters in the model (for example, so that parameter value files may be shared between different submodels which share some processes).

4 Input files

If you wish to run simulations which involve changing some parameter values during the simulation, then you need to create input files. A typical (short) input file might look like:

```
//input file for the lorentz model
steady: none
chosen_param: sigma
******
100.0, 10.0
10.0, 11.0
100.0, 10.0
```

This input file is read at the start of simulation, and the parameter sigma is altered during the simulation. For the first 100 seconds it is kept at the value 10, then increased to 11 for 10 seconds, and then reduced to 10 again for another 100 seconds.

Input files contain two parts separated by a line consisting of

```
******
```

The top part – the header – can contain a number of different options. The second part – the parameter change data – contains the actual values of the parameters to be changed.

Some options which can be put in the header are:

1. `chosen_param:` If you want to run a simulation where various parameter values are changed at various points, then you need to enter the names of these parameters here. If you do not want to change any parameters, then this line can be left out.
2. `time_step:` It is possible to run simulations with a fixed time step. In this case, the parameter change data part of the file will not have an initial column containing time-values.
3. `reset:` This allows some parameter values to be reset before the start of a simulation. For an ODE model this should not really be necessary, since the values can be changed in the parameter value file. However one might not wish to alter the parameter value file. For a DAE model (i.e. one which contains some algebraic relations) it is in general a good idea to change parameter values for a particular simulation using the `reset:` option. For example the line:

   ```
   reset: r_01, 0.015, 0.012, r_02, 0.0075, 0.006
   ```

   in the header resets parameter `r_01` from 0.015 to 0.012 and parameter `r_02` from 0.0075 to 0.006.
4. `steady:` This option allows you to control whether you get transient behaviour or not. The default behaviour is *to allow the model to run for a while* and thus to lose transient behaviour at the start of a simulation. If you want the transient behaviour, then you should have

   ```
   steady: none
   ```

   in the header of the input file.

5. `perturb:` This tells the simulator whether to add small random noise at each integration step. This can for example stop some (usually degenerate) systems getting trapped on subspaces which are actually not attracting. The default is no noise. If you want noise you should put

   ```
   perturb: yes
   ```

   in the header.

6. `loop:` This tells the simulator whether you want the data to be read once or many times. For example, if you want to simulate the behaviour of the model using a highly sampled blood-pressure wave form, you can input the data and ask the simulator to loop back to the start n times where n is an integer. The default is 1 (no looping), but if you wanted to loop say 10 times, then you could input:

   ```
   loop: 10
   ```

   in the header.

4.1 An example

A typical simulation file might start:

```
reset: r_01, 0.015, 0.011, r_02, 0.0075, 0.0055
chosen_param: P_a, Pa_CO2, SaO2sup
time_step: 1.0
******
1.077e+002   2.850e+001   9.866e-001
1.074e+002   2.851e+001   9.858e-001
1.054e+002   2.773e+001   9.908e-001

etc.
```

Three parameters – `P_a`, `Pa_CO2` and `SaO2sup` – are altered in this simulation at each time step. The time steps are of fixed duration (1 second). Before the simulation starts, two parameters, `r_01` and `r_02` are reset to new values. Further the system is allowed to settle. This file could equally be written with four columns of data:

```
reset: r_01, 0.015, 0.011, r_02, 0.0075, 0.0055
chosen_param: P_a, Pa_CO2, SaO2sup
******
1.0  1.077e+002   2.850e+001   9.866e-001
1.0  1.074e+002   2.851e+001   9.858e-001
1.0  1.054e+002   2.773e+001   9.908e-001

etc.
```

where now the first column contains the time data. Yet another way of representing the same data would be:

```
reset: r_01, 0.015, 0.011, r_02, 0.0075, 0.0055
chosen_param: P_a, Pa_CO2, SaO2sup
timestep: timestamp
******
1.0  1.077e+002   2.850e+001   9.866e-001
2.0  1.074e+002   2.851e+001   9.858e-001
3.0  1.054e+002   2.773e+001   9.908e-001

etc.
```

where now the first column is the time-stamp for the data, rather than how long the simulation should last for those parameter values.

5 Generating a new model: a little tutorial

Let's imagine you want to create a new model called `trial`. This will contain a trivial model consisting of one O.D.E:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -kx \tag{2}$$

where k is a parameter.

You should be able to create a new model by following the following steps:

1. Generate the skeleton of the model, either using the "create new model" button on the interface ("manage models" section), or by typing

   `src/lib/P_gennewsys trial`

   on the command line. This will create the following useful files:

   - A module file (`models/modules/trial/trial.dt`) (see Section 1).
   - A model descriptor file (`models/modeldescs/trial/alldata.dat`) (see Section 2).
   - A parameter value file (`models/pvals/trial/trial_pvals.dat`) – see Section 3.
   - An input file (`models/inputs/trial/trial_default.dat`) – see Section 3.

2. Now select the model (if you are using the interface) by clicking "select" in the model area and choosing "trial".

3. Now add the differential equation to the module file. This involves putting this text

   ```
   name: x_eq
   lterm: 1.0, x
   rterm: -k*x
   ******
   ```

   in the section `DIFFEQS` in `models/modules/trial/trial.dt`. The whole section might now look like:

```
DIFFEQS
name: x_eq
lterm: 1.0, x
rterm: -k*x
******
endDIFFEQS
******
```

4. Close the module file. Now if you are using the interface, hit the "parse" button ("manage models" section). This will warn you that you are going to change the file - click continue. You will notice that the parser has added some bits to the file which explain what are variable and parameters. It has added information that there is one variable `x` to the variable section and that it will be initialised by a parameter `x_init` so that the section looks like:

   ```
   VARS
   x x_init
   endVARS
   ******
   ```

   It has also added information that there are two parameters in the model `x_init` and `k` whose values will be set by hand, so that the `PARAMS` section looks like:

   ```
   PARAMS
   x_init 0.0
   k 0.0
   endPARAMS
   ******
   ```

   If you are not using the interface then add these bits to `models/modules/trial/trial.dt` by hand.

5. Now edit the model descriptor file `models/modeldescs/trial/alldata.dat`. Add `x_eq` to the `DEQS` section so that the whole section looks something like:

   ```
   ******** DEQS ********
   // the ODEs
   x_eq

   ******** endDEQS ********
   ```

   Close the model descriptor file.

6. Initialise the parameters by editing `models/pvals/trial/trial_pvals.dat`. You could put in the following lines into this file.

   ```
   x_init 1.0
   k 1.0
   ```

7. Your model is ready to run! If you are using the interface, click on "make" and if there are no errors, then "run". When it finishes running (which should be immediate) click "select" in the "plot variable" field, select "x" and click "gnuplot". You will see a plot of "x" decaying away. If you are working from the command line. Type "make trial", and then "make dat1". Then run the script "./makeplot" if you want to see the output – or look directly at the output file `outdat/dat`.

8. If you want to create a more sophisticated input file, then edit the file `models/inputs/trial/trial_default.dat` (have a look at Section 4 for what you might put in here.)