Jarnac - Scamp II
Quickish Introduction Guide

Version 1.19
Last revised 15 November 2001

Jarnac is a Windows 95/98/NT/2000 based interactive language for numerical analysis. It has in addition special facilities for modelling and studying the behaviour of integrated cellular systems such as metabolic pathways, signal transduction circuits and gene regulatory networks. The program's target audience is broad, and includes students, teachers, government funded researchers and researchers in the biotechnology and pharmaceutical industries.

Quick Start Guide

The main area of user interaction is at the console window, this is located at the top left of the screen. This is where you enter commands and instructions for Jarnac to obey. The lower panel is the built-in editor where you can write or edit script files – a list of commands or programming constructs for Jarnac to execute.

Operations can be performed with Jarnac in two ways: you can enter commands at the console for immediate execution, or, you can run Jarnac script files. At the console you can type things like, `sin (pi) + log10(1000)`, hit return and Jarnac will return an immediate answer to you. On the other hand you may have a whole set of things you wish to calculate in which case it might be convenient to put all the instructions into a file, a so-called script file, and get Jarnac to run the script file.

Here are four important console commands you should be aware from the very beginning:

`run filename`	Run a script file, eg `run calvin`
`edit filename`	Load the file, filename, into the editor window
`dir`	Display directory listing of current directory, eg `dir`
`cd path`	Change current directory, eg `cd glycolysis`

The `run` command is used to run Jarnac scripts files. Scripts can be written using any editor that can save ascii text files, this can include nodepad, WinEdt or more simply Jarnac's built-in editor.

Current Directory

Jarnac maintains an internal variable called the current directory. All operations involving external files, for example, running script files, opening export files, saving data objects, etc., will use the current directory as the search path. The current directory can be examined using the predefined variable 'sys.path'. Setting the current directory can be done either by assigning a new path string to the path variable (eg `sys.path="c:\MyModels"`) or using the system command, 'cd', e.g. `cd c:\MyModels`, note that the command `cd` does not require the path to be put in quotes, this is because `cd` is a console command rather than a Jarnac language statement (See Jarnac Command and Language Statements).

Simple Examples from the Console Window

In the following console extracts, the symbol, `->` is the Jarnac prompt. Everything after the `->` is **user input**, every line without a `->` is **Jarnac output**. I assume that at the end of each line the user has hit the return key.

```
->sys.path
c:\MyModels
->sys.path="c:\MyModels\liver"
->sys.path
c:\MyModels\liver
->
```

Numeric handling

```
->1/2 + 10
10.5
->a = 2
->b = 7.5
->a*b
15
->sin (a*b)
0.650287840157117
->m = {{1,2,3},{7,5,2},{8,7,6}}
->1/m
{{-1.778   -1  1.222}}
 { 2.889    2 -2.111}
 {    -1   -1      1}}
```

```
->mt = tr (m)
->invmt = 1/mt
->
->println "pi times 2 is", pi*2;
pi times 2 is 6.2832
```

Running Script Files

You can write script files using any basic ascii editor but more conveniently you can use the built-in editor that comes with Jarnac. To start a new script file click on the new button in the editor tool bar. Now enter your commands into the lower panel, for example enter the following lines (Note the semicolons at the end of each line):

```
H = 1E-4;
pH = -Log10 (H);
println "The pH is ", pH;
```

Now save the file to some convenient location using the save-as button in the tool bar, and call the file `myfile.jan`. To run this script you can either type the command `run myfile` at the console or click on the run button in the tool bar.

Loading existing script files into the editor is simply a matter of typing `edit myfile` at the console or clicking on the load file button on the editor tool bar. Save any edit by using the save button on the editor tool bar.

Script files should have the extension `.jan`. Commands such as `run` and `edit` will search for file names of the form `x.jan`. This means it is possible to omit the extension and simply use `run x` and Jarnac will implicitly add the extension. Thus to edit a file called `mymodel.jan` one can type the abbreviated form, `edit mymodel` – although `edit mymodel.jan` is equally acceptable – and Jarnac will implicitly add the `.jan` extension for you. This is simply a convenience for the user.

Metabolic Models

Metabolic Models can be defined using a similar syntax to that used in Scamp command files. Pertinent points include: Boundary (fixed) metabolites are indicated with a dollar character proceeding the name of the metabolite (eg $X0); Reactions can be named using a square bracket syntax (eg [J1]), useful if you want to refer to the flux of a reaction; Kinetic rate laws come immediately after the reaction spec, if only a structural analysis is required, it is not necessary to enter a full kinetic law, a simple `... -> S1; v;` is sufficient. The `->` symbol is used to separate the reactants from the products.

Example:

```
DefaultModel Branch
   [MainFeed]    $X0 -> S1;  Vm*X0/(Km + X0);
   [TopBranch]    S1 -> $X1; Vm1*S1/(Km1 + S1);
   [BottomBranch] S1 -> $X2; Vm2*S1/(Km2 + S1);
end;
```

Note that there is no need to pre-declare the metabolite names shown in the reactions or the parameters in the kinetic rate laws. Strictly speaking, declaring the names of the floating species is optional, however this feature is for more advanced users who which to define the order of rows that will appear in the stoichiometry matrix. For normal use there is no need to pre-declare the metabolite names. See the glyucol.jan file for an example where metabolic names are predeclared.

Initialisation of Model Values

Initialising a default model is very simple:

```
X0 = 3.4;
X1 = 0.0;
S1 = 0.1;
Vm = 12; Km = 0.1;
Vm1 = 14; Km1 = 0.4;
Vm2 = 16; Km2 = 3.4;
```

Steady State and Metabolic Control

To evaluate the steady-state (make sure the model values have been previously initialised) enter the following statement at the console.

`->ss.eval`

This statement will return a value indicating how effective the computation was, essentially it returns the norm of the rate of change vector (i.e sqrt (Sum of dydt)). The closer this is to zero the better the approximation to steady state.

Once a steady state has been evaluated, the values of the metabolites will be at their steady state values, thus `S1` will equal the steady state concentration of `S1`.

The fluxes through the individual reactions can be obtained by either referencing the name of the reaction (eg `J1`) or via the predefined rate vector `rv[]` which is part of the model object. The advantage to using the rate vector is that the individual reaction fluxes can be accessed by indexing the vector (see example below).

```
->println J1, J2, J3
```

```
3.4, ...
->for i = 1 to 3 do println rv[i]
3.4, ...
->
```

To compute control coefficients use the statement:

`cc (Dependent Measure, Independent parameter)`

The dependent measure is an expression usually containing flux and metabolite references, for example, `S1, S1+S2, J1`. The independent parameter must be a simple parameter such as a Vmax, Km, ki, boundary metabolite (X0) or a conservation total such as CSUM1. In the current release it is not yet possible to use a floating metabolite as an independent parameter. Examples include:

```
cc (J1, Vmax1)
println cc (J1, Vm) + cc (J1, Vm1) + cc (J1, Vm2)
cc (J1+J2+J3, Vmax1)
cc (J1, X0)
cc (J1, CSUM1)
```

To compute elasticity coefficients use the statement:

`ee (Reaction Name, Parameter Name)`

For example

```
ee (J1, X0)
ee (J1, S1)
```

Since cc and ee are built-in functions they can be used alone or as part of larger expressions. thus it is easy to show that the response coefficient is the product of a control coefficient and the adjacent elasticity by using:

```
R = cc (J1, X0)
println R - cc (J1, Vm) * ee (J1, X0)
```

To obtain the conservation matrix for a metabolic model use the model method, `conserve`.

```
->p=defn cycle [J1] E + S1 -> v; [J2] ES -> E + S2; v; [J3] S2 -> S1; v; end
->p.conserve
     E S1 ES S2
C1{{ 0  1  1  1}
C2 { 1  0  1  0}}
->
```

Th result given above indicates that the conservation relations, `S1 + ES + S1` and `E + ES` exist in the model. As a result, Jarnac would generate two internal parameters `CSUM1` and `CSUM2` corresponding to these two relations.

Note that the above example declares a model slightly differently. The model is defined as a so-called named model, in this case, named 'p'. To reference model properties and methods, the property or method must be proceeded with the name of the model, eg `p.S1 = 2.3;`

Time Course Simulation

There are two methods for evaluating the time evolution of a metabolic model, these are, `sim.OneStep` and `sim.Eval`. `OneStep` is used to compute a single time step between two time points while `Eval` is used to compute the model over a range of time steps. The advantage of `OneStep` is that the user has more control over the course of the simulation. `OneStep` takes two arguments and returns a double value.

```
tnext = sim.OneStep (tStart, tStep)
```

`OneStep` takes the start time point of the simulation and the step size required to simulate over. For convenience `OneStep` returns the new time value, i.e. `tStart + tStep`. Example:

```
t = 0;
repeat
  t = sim.OneStep (t, 0.1);
  println S1;
until t > 10;
```

Since this takes a number of lines to describe it would probably be better to enter this into a script file and run the script file. We can do the same type of simulation using `sim.eval`, with the advantage that it can done with a single line of code:

```
m = ss.eval (0, 10, 1000, [S1])
```

`sim.eval` returns a matrix which in the example is 1000 rows deep (corresponding to the value of the third argument, the required number of points) and one column wide (corresponding to the number of items in the fourth argument). The fourth argument is a list containing the desired outputs, a number of examples include:

```
m = ss.eval (0, 10, 1000, [S1, J1, J2, J2])
m = ss.eval (0, 10, 1000, [S1, J1+J2])
m = ss.eval (0, 10, 1000, [Time, S1, S1*Log10 (J1)])
```

Note that the special variable `Time` is available which represents the independent time variable in the model. The first argument equals the start time point, the

second argument the end time point and as mentioned before the third argument represents the required number of output points.

To visualise the output in the form of a graph one simply needs to pass the matrix variable returned by sim.eval to the command, graph. Thus:

```
m = ss.eval (0, 10, 1000, [Time, S1, J1, J2, J3])
graph (m)
```

or if only the graph is required:

```
graph (ss.eval (0, 10, 1000, [Time, S1, J1, J2, J3]))
```

To selectively graph particular items from the generated matrix use the syntax shown in the following example:

```
graph (m, [2, 3])
graph (m, [1, 2, 4])
```

The list argument indicates which columns of m to graph. The first column specified in the list is always considered the independent variable (x axis). This means that the statement, `graph (m, [1,2,3,4])` is equivalent to `graph (m)`.

Other Network Objects of Interest

I have already mentioned one object that is part of the metabolic model, the rate vector, `rv[]`. There are however a number of other predefined objects associated with a metabolic model which might be of interest. Of particular interest are the stoichiometry matrix, `sm` and the jacobian matrix, `jac`. These are returned to the user as matrices and can thus be treated like any other matrix type.

```
println sm
println jac
```

The jacobian is evaluated at the current state of the model, be it in steady state or not.

Jarnac Command and Language Statements

When text is entered at the console window Jarnac first checks to see if the first word in the text is a command. If Jarnac detects a command then the entire line is passed to the command processor for decipherment and execution by the interactive console processor. If the console process does not recognise the entered text as a command then the input to passed instead to the Jarnac

language parser where an attempt is made to parse the line into the Jarnac internal machine language. If this is successful, the Jarnac console process passes control over to the Jarnac virtual machine where execution of the request is carried out.