33 JARNAC: a system for interactive metabolic analysis

H.M. Sauro
5A Lynedoch Place, West End, Edinburgh EH3 7PX, U.K.
http://www.fssc.demon.co.uk

Introduction

Jarnac is a Windows (95/98/NT) based interactive environment for studying models of chemical and biochemical networks. It can for example be used to study metabolic systems, gene regulatory networks or signal transduction pathways.

Jarnac supports a text based language that can be used to describe and interrogate metabolic models. This is a similar approach to previous software packages developed by the author, such as Scamp [1,2], iMap and Indigo [2]. Other approaches exist, such as Gepasi [3,4] or DBSolve [5] which employ a Windows interface incorporating menus, buttons and lists, or Talis [6] which is a fully interactive visual environment for studying metabolic systems.

Jarnac was primarily designed for research purposes but could easily be used in a teaching environment owing to its interactive nature. Jarnac incorporates features not found in other metabolic simulation software, most notably the ability to model populations of pathways, and to manipulate both the structure and kinetics of individual pathways at run time.

Methods

Jarnac is built from four main components, a virtual machine runtime engine, a parser and code generator, a symbol table manager and a Windows front-end. The Windows front-end supports the creation and execution of Jarnac script files, running Jarnac interactively (with history management) or in batch mode, and 2D graphical output.

The simplest way to introduce Jarnac is to type commands at Jarnac's interactive command line. Here's an example (Bold characters indicate user input, non-bold Jarnac output):

```
-> println "Hello World"
Hello World
-> a = 5.6
-> println a/2
2.8
```

The symbol -> is the interactive Jarnac prompt and commands issued at the command-line are executed immediately. Help is available either by double clicking on a key-word and issuing a normal 'F1' Windows help or by simply typing a question mark ('?') followed by the name of the command, e.g., `?eigen`.

Jarnac has built-in computational support for dynamic simulation using the LSODA integrator, steady state analysis, simple stability analysis, matrix manipulation and operations, full steady state metabolic control analysis and basic metabolic structural analysis. To support these features Jarnac has a number of predefined types, which include:

Type	Example
Integer	`4, 7, -99`
Double	`1.2, 1E-5, -7.65`
Boolean	`True, False`
String	`"Glucose"`
Vector	`{1, 2, 3}`
Matrix	`{{1, 2, 3}, {6.6, 7.8, -2}}`
Lists	`[1, 6.7, Km, "str", {1.5, 6.7}]`
Networks	`defn cell [J1] S1 -> S2; k1*S1; end`

Some data types, in particular the lists and networks, are objects in the sense that they have additional functions, methods and properties accessed via the '.' indirection operator. For example, the network object type has a function called `sm` which returns the stoichiometry matrix of the network.

```
-> p = defn cell [J1] S1 -> S2; k1*S1; end
-> println p.sm
{{-1 }
 { 1 }}
->
```

The full list of functions and properties is given at the Jarnac web site http://www.fssc.demon.co.uk.

Jarnac incorporates a full programming language with constructs such as repeat/until, while/do, for loops, conditionals and user defined functions. In addition Jarnac provides a comprehensive library for matrix arithmetic manipulation, list manipulation and above all metabolic network support. Jarnac also supports modules which enables functions and data to be encapsulated in module objects.

Results

In the normal course of events a user would probably enter a metabolic model into a script file, run the Jarnac script file from the interactive command line and then interrogate the model interactively. For example, the following text can be entered into a script file using the built-in editor of the Jarnac IDE and saved under the file name, 'linear1.jan'. Note that the rate kinetics specified for each reaction of the network are optional; this means that networks can be defined purely with structural analysis in mind.

```
// Lines such as this are comments
// Define a linear network using the built-in rate law srmm
// srmm models the single substrate Michaelis-Menten model

p = defn Linear
        [J1] $X0 -> S1;  srmm (X0, S1, Vm1, Km1, Km2, Keq1);
        [J2]  S1 -> S2;  srmm (S1, S2, Vm2, Km3, Km4, Keq2);
        [J3]  S2 -> $X1; srmm (S2, X1, Vm3, Km5, Km6, Keq3);
    end;

// Initialise the parameter values
p.X0 = 1.0; p.X1 = 0.0;
p.S1 = 0.1; p.S2 = 0.1;

p.Vm1 = 5.6;  p.Vm2 = 0.7; p.Vm3 = 9.8;
p.Km1 = 0.6;  p.Km2 = 1.2; p.Km3 = 3.3;
p.Km4 = 9.6;  p.Km5 = 10.4; p.Km6 = 0.4;

p.Keq1 = 10; p.Keq2 = 15; p.Keq3 = 20;
```

At the interactive command line the following conversation takes place between Jarnac and the user:

```
-> run linear1
-> println p.cc (p.J1, p.Vm1), p.cc (p.J1, p.Vm2), p.cc (p.J1, p.Vm3)
0.1238 0.8469 0.0204
-> println p.S1, p.S2, p.J1, p.rv[1];
0.1000 0.1000 0.0000 3.3600
-> p.ss.eval
-> println p.S1, p.S2, p.J1;
6.2332 0.4973 0.4472
->
```

The command 'run' is a Jarnac command used to execute a script file. The words, cc, rv etc. are methods and properties associated with the network object defined in linear1.jan. For example, cc is a built-in method to compute control coefficients and rv is a built-in vector property of the network for supplying the rates of reaction. The statement, p.ss.eval is used to compute the steady state of the network model, p.

One tedious aspect of the above conversation is the requirement to prefix all network references with the variable name 'p.'. This can be useful when there are many models defined in a single session and one needs to distinguish between them. However if you can guarantee only a single model in a session then it is possible to use the so-called default model for which one is allowed per Jarnac module. For example:

```
// Default Model

DefaultModel Cell
  [J1] $X0 -> S1; .....

end;

Vm1 = 2.3; Vm2 = 7.8; ....

S1 = 0.1; S2 = 0.1;
```

The same conversation as previously illustrated above now takes a simpler form, i.e all the references to 'p.' are avoided:

-> **run linear1**
-> **println cc (J1, Vm1), cc (J1, Vm2), cc (J1, Vm3)**
0.1238 0.8469 0.0204
-> **println S1, S2, J1, rv[1];**
0.1000 0.1000 0.0000 3.3600
-> **ss.eval**
-> **println S1, S2, J1;**
6.2332 0.4973 0.4472 ->

Discussion

A novel aspect of Jarnac and still currently under development is its ability to work with populations of networks. Consider again the model in linear1.jan. The network was assigned to the variable p. A special method of networks is `clone` which, as the name implies, creates an exact copy of a network. Thus is is possible to issue the following command:

```
-> q = p.clone
```

The variable ‘q’ now holds an exact copy of p. Note that the statement ‘q = p’ does not carry out the same operation but merely assigns a reference of p to q.

Alternatively, by using the list data type it is possible to create a large pool of networks, for example:

```
-> pop = []
-> for i = 1 to 100 do Pop.Append (p.clone)
```

The above example illustrates a number of ideas. First a variable called Pop is introduced and an empty list assigned to it. A loop is then set up to construct a list made up of 100 cloned copies of p.

Access to the list is simply a matter of using array indexing. Thus to compute the steady state of the 49th clone, one simply issues the statement:

```
-> Pop[49].ss.eval
```

Several methods are available to manipulate networks, including adding, deleting or changing both structural and dynamic aspects of the network.

Future Developments

The future development of Jarnac lies to four broad areas:

- Integration with a visual network designer such as Talis. This would allow structural and dynamic patterns in a metabolic system to be identified visually.
- Allow networks to be built up from a library of networks. The example below illustrates a possible syntax for this:

```
// Define a basic enzyme model
defn enzyme (S1, S2)
  [J1] S1 + E => ES; k1*S1*E - k2*ES;
  [J2] ES -> S2 + E; k3*ES;
end;

// Build a simple linear chain from the basic
// Enzyme building block
p = defn Pathway
      [J1] $X0 -> S1; Enzyme (X0, S1);
      [J2] S1 -> $X1; Enzyme (S1, X1);
    end;
```

- Expose the internal functionality of Jarnac through a COM interface so that Jarnac services could be accessed by other applications. Also, disconnect the user interface from the runtime system to allow remote access across the internet.
- Allow other developers to add enhancements by adding additional modules written in C/C++ or some other suitable development language.

Availability

Jarnac is currently available free of charge. Executables and documentation can be obtained from http://www.fssc.demon.co.uk under the biotechnology heading. Enquiries can be made directly to the author at HSauro@fssc.demon.co.uk.

Examples

The following is a complete Jarnac example Script:

```
// Oscillating pathway from Heinrich et al '77

// Example illustrating two approaches to generating time
// course data. Also illustrating the graph statement.
// The graph stmt accepts a matrix argument. The first column is
// treated as the x coordinate and the remaining columns the
// y coordinates however many there are.

DefaultModel
     [J1]  $Xo -> S1;   vo;
     [J2]   S1 -> $X1;  S1*k3;
     [J3]   S1 -> S2;   (k1*S1-k_1*S2)*(1+c*S2^q);
     [J4]   S2 -> $X2;  S2*k2;
end;


Xo = 1.0; X1 = 0.0; X2 = 0.0;
S1 = 1.0; S2 = 1.0;

// set vo = 7.5 for stable system
// set vo = 8.0 for oscillating system
vo = 7.5;

c = 1.0; q = 3;
k1 = 1; k_1 = 0; k2 = 5; k3 = 1;

// Generate the data using a single call
// Args: Start Time, End Time, Number of Points, Compute list
m = sim.eval (0.0, 25, 2600, [Time, J1-J2, S1, S2, S1/S2]);
graph (m);  // Graph the data

// Comment the above two lines and uncomment these to get a phase plot
//m = sim.eval (0.0, 25, 2600, [S1, S2]);
//graph (m);

// ... or generate the data with more control using OneStep
t = 0.0;
hstep = 25/1000;
for i = 1 to 10 do
    begin
    S3 = S1/S2;
    println t, J2, S1, S2, S3;  // or ..., S1/S2;
    t = sim.OneStep (t, hstep);
    end;
```

And a second example illustrating some steady state analysis:

```
// Simple steady state analysis including estimating
// some control coefficients and confirming the
// summation theorem

DefaultModel
    [J1] $X0 -> S1; k*(k1*X0 - k2*S1);
    [J2] S1 -> $X1; k3*S1;
end;

k = 1.0;
k1 = 1.2;
k2 = 3.4;
k3 = 0.4;
X0 = 1.0;
// Compute steady state
ss.eval;

// Now calculate some mca coefficients
C1 = CC (J1, k);
C2 = CC (J1, k3);
sum = C1 + C2;

// Concentration control coefficients:
CS11 = CC (S1, k);
CS12 = CC (S1, k3);
```

```
println "   J1     S1";
println J1, S1, nl;
println "    C1    C2    Sum";
println C1, C2, Sum, nl;
println "   CS11   CS12";
println CS11, "  ", CS12, nl;

println "Elasticities for first step:", nl;
println "ee(J1,X0):", ee (J1, X0), "  ee(J1,S1): ", ee (J1, S1), nl;

println "Elasticities for second step:", nl;
println "ee(J2,S1):",  ee (J2, S1), "  ee(J2,S1): ", ee (J2, X1), nl;
```

References

1. Sauro, H. M. (1993) SCAMP: a general-purpose simulator and metabolic control analysis program *Comp. Appl. Biosci.* **9**, 441–450.

2. Scamp, iMAP, Indigo: http://www.fssc.demon.co.uk

3. Mendes, P. (1993) GEPASI: A software package for modelling the dynamics, steady states and control of biochemical and other systems. *Comp. Appl. Biosci.* **9**, 563–571.

4. Gepasi: http://www.ncgr.org/software/gepasi/

5. DBSolve: http://websites.ntl.com/ igor.goryanin/

6. Talis: http://www.fssc.demon.co.uk