Jarnac 2.0 - Reference Guide

Version 2.0
Last revised 7 June 2005

Jarnac is an interactive and interpreted language for modeling integrated cellular systems, such as metabolic, gene networks and signal transduction circuits.

This document is a summary of the current state of Jarnac. Since Jarnac is still under development the specifications outlined here may change at a future date.

Summary of Capabilities

- Includes a General Purpose Scripting Language
  - Rich variety of data types, Floats, Complex, Matrices, Lists, etc.
  - Conditional Statements, if/then/else
  - Loop constructs, repeat, while and for
  - User Functions
  - Exception Handling
  - File Input/Output, Graphical Output
  - Modules for building libraries of functionality
- Support for Metabolic Modeling
  - Represent chemical networks using a familiar chemical notation
  - Optionally build chemical networks using a visual tool
  - User specified rate laws or use built-in rate laws
  - Compute Jacobian, elasticities and control coefficients
  - Perform null space and elementary mode (Metatool) analysis
  - Automatic detection of conserved cycles
  - Access to Reder matrices
  - Compute simple equilibrium reactions
  - Runtime structural manipulation of metabolic models
  - Maintain and simulate populations of metabolic models
  - Compartmentation
  - Perform time course, steady state or continuation analysis
  - Export/Import models in SBML XML standard format

Contents

1 Brief Introduction

The purpose of Jarnac is to help people model and study the internal networks and dynamics of living cells or more ambitiously to model interacting multi-cellular systems. It does this by enabling a user to describe the many and varied chemical processes and interactions that go on in cells using a syntax familiar to the average biologist or chemist. Thus Jarnac allows a researcher to describe cells in terms of metabolites, enzymes, effector molecules and so on and avoids the user having to write down differential equations or work out whether there are conservation relations or not. The mathematical and technical side is all done in the background which enables a researcher to concentrate on scientific questions rather than technical details.

To support the ability to model cellular systems, Jarnac implements a rich scripting language to control, build and manipulate models. In addition to controlling cellular models, Jarnac can also manipulate a variety of data types, from simple integers, to floating point numbers, to vectors, matrices and so on. The Jarnac scripting language supports the usual language constructs, such as looping, conditionals, user functions and modules. Modules and user functions are a powerful feature that allow users to extend Jarnac's capability. Jarnac is also an interactive environment, thus a user issues commands or executes scripts at a console window with the results being immediately returned to the user for inspection. This rapid feedback of results, enables a user to quickly learn how to use Jarnac and helps them get 'closer' to the biological problem.

The following sections summarise the main capabilities of Jarnac, it's scripting language and it's support for modelling cellular systems.

For distributed access, Jarnac supports two modes for external access.

Sockets Jarnac exposes a standard BSD socket which has a direct link to the Jarnac parse engine. This makes it simple to send scripts to a copy of Jarnac running on a remote machine. The socket interface provides a simple authentication mechanism to prevent obvious abuse.

SBW Jarnac implements a Systems Biology Workbench (SBW) interface to it's core simulation engine and scripting interface. This allows Jarnac to be controlled from many other languages including C/C++, Java, Python and Perl.

SBW is the recommended mode for external access.

2 Sample Script

To give you an idea of what a Jarnac script looks like, here is a simple example which defines a metabolic model and then carries out a time course simulation of the model. Note that all lines beginning with the symbols `//` are comment lines and are ignored by Jarnac.

```
// Define the pathway model
p = defn Pathway
      // Declare the floating and boundary species
      var S1, S2; ext X0, X1;

      // Declare each reaction and its associated rate law
      J1: X1 -> S1; k1*X0 - k2*S2;
      J2: S1 -> S2; Vm*S1/(Km + S1);
      J3: S2 -> X1; k3*S2^n;
    end;


// Now initialise all the parameter and variables
p.X0 = 1.2; p.X1 = 0.0;
p.S1 = 0.001; p.S2 = 0.001;
```

```
p.k1 = 1.2; p.k2 = 5.6;
p.Vm = 10.5; p.Km = 0.4;

p.k3 = 5.6; p.n = 3.5;

// Define some convenient variables
TimeStart = 0;
TimeEnd = 10;
NumberOfDataPoints = 100;

// ... and perform the simulation
m = p.sim.eval (TimeStart, TimeEnd, NumberOfDataPoints, [<p.Time>, <p.S1>, <p.S2>, <p.J1>]);

// eval returns a matrix which we can pass directly to the graph method
graph (m);
```

3 Mode of Operation

Jarnac can work in three modes:

- Interactive Mode
- Batch Mode
- Socket Batch Mode

Interactive Mode In interactive mode, a user types commands at the console and Jarnac executes the commands immediately. For example:

```
->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
```

Note that the symbol, `->` is the Jarnac interactive 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.

Batch Mode In batch mode a user creates a file containing a list of commands, a so-called script file. Script files can be generated either using an external editor or more conveniently by using the built-in Jarnac editor (See fig. 1). To execute a script file, the user simply issues the command `run ScriptFileName` at

the interactive prompt where `ScriptFileName` is the name of the script file or clicks on the green run script button.

```
// Example script file
count = 0;
for i = 1 to 100 do
    begin
    x = urnd();  // Generate a random number, 0 -> 1
    if x > 0.9 then
       count = count + 1;
    end;
println "Proportion of random numbers greater than 0.9 is", count/100;
```

Socket Batch Mode The socket batch mode is similar to the batch mode, however instead of a script being read from a file, the script is accepted form a standard BSD socket. This allows Jarnac to be run from remote machines or be controlled from other languages, eg Java, VB or Python.

SBW Mode SBW requires the installation of the Systems Biology Workbench. SBW is a broker system which allows applications to communicate with each other and carry out remote procedure calls. Jarnac can act as a service provider for SBW. Two examples where Jarnac is used in this mode include JDesigner and cellDesigner.

4 Getting Help

There are a number of ways to obtain help:

- Use the Windows html-help. This is accessible by using the F1 function key. Or double-click on a text item and help will attempt to search for the word in the help file, if the text is found the appropriate help window is automatically displayed.
- Use the quick help feature. If you know the name of the function, method etc. then you can get quick help by appending the name of the item to a question mark character at the interactive console. For example if you wish to obtain help on the function, `clone`, then issuing the command, `?clone` at the console will yield a help message on `clone`.
- Use the class viewer on the tool bar. The class viewer shows in tree structure form, all the methods, functions, properties and objects currently active in the system together with help information.
- Type the help command at the interactive console window.

5 Built-in Editor

The user dialog window shown in figure 1 shows two main panels within the window, an upper console panel where a user enters commands and receives output, and a lower panel which supports a multi-tab editor.

The editor supplied with Jarnac is a standard Windows editor, very much like notepad. It supports all the usual functions, including cut/copy/paste, undo/redo to multiple levels, basic syntax highlighting and find. The editor panel allows multiple files to be edited simultaneously by way of tabbed pages. Loading and saving of files is carried out via the editor toolbar shown immediately above the editor panel. The controls in the toolbar always operate on the currently selected edit tab.

Note that many functions are accessible from a popup menu which can be accessed by right clicking over an edit page.

5.1 Drag and Drop

The editor panel is also drag and drop aware, that is, you can drag a file from, say Windows Explorer and drop it on to the editor panel, this will cause the file to be loaded automatically in to a new edit page.

It is also possible to configure Jarnac so that double clicking on a file in, say Windows Explorer, will cause Jarnac to be started and the double-clicked file to be loaded automatically into a new edit page. You can do this by adding a new file type (.jan files) to the Windows file type list and associating Jarnac with the new file type. Add the operation 'open' and set the action to the Jarnac path followed by the string `"%1"`, note that the double quotation marks are part of the string.

6 Data Types

Jarnac supports a variety of different data types, these can be split into two groups, primitive and object types. Primitive types are generally faster to manipulate compared to object types; however object types posses a much richer array of abilities and representation.

Primitive Types

Integer	signed 32-bit integer eg. 5, 34586
Float	double precision, eg. 1.23454, 0.45, 3.4e-4
Complex	double precision complex number, eg 2 + 4i, 7i 1 - 3i
String	"a string"
Boolean	True or False
Vectors	{1,2,3} Elements may be constants or expressions
Matrices	{{1,2}, {4,5}} Elements may be constants or expressions
Code Types	<a+b>

Object Types

Lists	["XYZ", 1.2, x*y, [i,j,k, 1,2,3,4]]
Networks	defn cell [J1] S1 -> S2; v1; [J2] S2 -> S3; S2*k1; end
i/o Objects	File, Console or Printer streams
User Functions	eg function test (x) return pi*x; end;
Graphs	Data visualisation objects

Examples of different data type assignments:

```
a = 1.2345;
i = 99; j = 88; k = 77;
c = 4 + 7.4i;
d = 40i;
m = {{i,2,3}, {j,5,6}, {k,8,9}};
v = vector (1000};
Name = "Jim Smith";
Spock = True;
f = <sin (x) + cos (1.2*y)>;
ShoppingList = ["Bread", "Newspaper", "Olives"];
function MyFunc (x, y) return sin(x)*cos(y); end;
```

6.1 Integers

Integers are stored as 32 bit signed values, that is a double word. Integers have the numeric range `-2147483648` to `2147483647`. A number of functions exist to convert integers to and from string format, `StrToInt` and `IntToStr`. If you wish to control the conversion of an integer to it's string representation, use the `Format` function.

6.2 Floats

Floats represent the usual scientific floating point number. In Jarnac they are implemented using IEEE standard double format. That is each float requires eight bytes of storage. The range which a float covers is 5.0×10^{-324} to 1.7×10^{308} and each number has 15 to 16 digits of significance.

Jarnac also has built-in two floating point constants, `NAN` and `Inf`, these represent 'Not a Number' and 'Infinity' respectively, as defined in the IEEE standard.

6.3 Complex Numbers

The native float type for Jarnac is actually the complex type. A complex number is defined as pair of floating point numbers, a real part and an imaginary part. Both numbers have the same restrictions on their range as a normal floating point number. The imaginary part of a complex number is indicated by a 'i' or 'j' placed immediately after the number, for example 0.1i, or 3.1415j.

To create a complex number with a nonzero real part, add a floating point number to it, e.g., 3+4i

6.4 Boolean

Boolean types occupy one byte and can have only one of two states representing true or false. Two built-in constants, `False` and `True` are provided to allow boolean types to be set and tested.

6.5 Strings

Literal strings are indicated using the double quote character, `"`, thus, `"Hello World"` is a literal string. If you wish to include the double-quote itself as part of the literal string, use two double-quote characters in succession. For example the string `"""Hello"""` yields `"Hello"`.

The escape character, '\', may be used to insert a limited number of special characters into a literal strings. These are listed below:

`\\`	Yields the character '\'
`\n`	Carriage Return/Line Feed (newline)
`\r`	Carriage Return
`\f`	Line Feed
`\t`	Tab character

6.6 Code Types

Code types are a way of converting simple algebraic expressions into Jarnac's internal code representation. Code types start with the symbol `<` followed by the expression, finishing with a closing `>`.

For example, the following are code type expressions:

```
f = <a*b + c>
TotalFlux = <p.J1 + p.J2 + p.J3>
FuncToPlot = <sin(x)*2.3 + cos(y)*5.6>
```

When a code type is defined, the `<...>` construct returns a reference to the code, that is a reference to the internal code that Jarnac generates when it parses the expression. This is in contrast to a normal expression which is evaluated immediately and the result returned to the left hand side of the assignment.

Thus the expression, `2+3`, returns the value `5`, while the expression, `<2+3>` returns a reference to the internal Jarnac code which represents the operation `2+3`.

If you wish to evaluate a code type, use the `eval` function. The `eval` function takes a single argument which is the code type expression. The evaluation of the code occurs within the current symbol context. If Jarnac finds that it cannot locate a particular symbol in the code type expression, a runtime error is generated. When evaluating code types within user functions, Jarnac uses the user function symbol context to evaluate the code type.

Like any other data type it is possible to pass code type expressions or variables which reference code types, to user defined functions. Code types are an important data type in a number of numerical analysis situations, in particular they are important during simulation runs and when a user requests the computation of metabolic control analysis coefficients.

```
f = <x*x>;
x = 3.4;
println eval (f);
```

Side Note: The `eval` method will also accept string arguments in addition to code types. This means one can evaluate arbitrary expressions at runtime, for example:

```
function PlotMe (g, f)
   x = 0;
   for i = 1 to 50 do
       begin
       g.Markat (x, eval (f), 0);
       x = x + 0.1;
       end;
end;

g = newgraph;
PlotMe (g, "sin (x)");
ReadKey;
g.close;
```

7 Comments

Jarnac script files may be sprinkled with comments, that is text which is ignored by the Jarnac interpreter. Comments are very useful for annotating scripts and help a reader determine the purpose of the script or assist in describing the particular algorithm used. There are three ways to indicate comments, C++ style using '//', C style using '/* ... */' and the Scamp comment, '# ...;'. These three notations are illustrated in the following examples:

```
// Assign values to some variables
a = 5;
b = 7;

# Print out one variable divided by another;
println a/b;

/* None of the following will be executed:
a = 2;
b = 2.5;
```

```
println a/b;
*/

println "But you'll see this message";
```

Comments beginning with '//' end at the end of line. Comments beginning with the hash symbol ('#') end at the first semicolon and comments beginning with the symbol '/*' end at the first occurrence of '*/'.

8 Jarnac Identifiers

Identifiers, that is symbolic names that refer to data, must always start with a letter or an underscore character. Letters, underscore and digits may be mixed freely within the body of the identifier.

```
Identifier ::= Letter | UnderScore { letter | underscore | digit }
letter     ::= 'a'..'z', 'A'..'Z'
underscore ::= '_'
digit      ::= '0'..'9'
```

Punctuation characters are not allowed in an identifier. Identifiers can be of any length and case is **not** significant.

Examples of legal identifiers:

```
ATP
Glucose
Glucose_6_Phosphate
_3PG
```

Examples of illegal identifiers:

```
3PG   // Must not start with a digit
Glucose-6-Phosphate // No punctuation characters allowed
$$ATP
```

9 Assignments

Assignments are values ot variable names is achieved using the '=' symbol, thus

```
a = 2.3;
```

10 Arithmetic Operations

Jarnac supports the usual basic arithmetic operations, including the power operator represented by the symbol '^'. Many of these operations will work on a variety of data types. Jarnac implements a system call dynamic typing which means that the decision as to whether a particular operation can be performed on some data type is made at runtime rather than at compile time. This feature allows one to write generic user functions which can accept any type.

Operation	Applicable to
'+', '-' (unary ops)	integers, floats, complex, vectors, matrices
'+' (addition)	integers, floats, complex, strings, vectors, matrices
'-' (subtraction)	integers, floats, complex, vectors, matrices
'*' (multiplication)	integers, floats, complex, strings, vectors, matrices
'/' (division)	integers, floats, complex, vectors, matrices
'^' (power)	integers, floats, complex
'div' (int divide)	integers, floats

Operator Priority Levels

Arithmetic operators have the expected precedence, that is (highest first), `not, power, (*,/,and), (+,-,or,xor), (<, >, <>, ...)`. Parentheses can be used to override this order.

Examples:

```
a = 2; b = 7;

c = -(a + b)/7;

p = c^2.4;

str1 = "Hello"; str2 = " Everyone";
greeting = str1 + str2;

str3 = 2 * str1  // Yields: "HelloHello"
v = {1,2,3}; u = {5,6,7};

w = v*u;  // Dot product

c = a / 2.3;

m = {{1,2}, {6.7, 9.0}};
minv = 1/m;  // Compute the inverse
```

11 Comparison and Boolean Operators

Jarnac includes the usual range of relational operators. One special point to make is that the equivalence operator is the '==' symbol.

For example, to test whether the variable `a` equals the value four we would use:

```
if a == 4 then ....
```

The reason why the '==' symbol was chosen rather than say the more obvious use of '=', is that the interpreter is unable to distinguish in an expression such as `a = 4` whether you mean an assignment or an equivalence test. In languages such as Pascal, the assignment operator is ':=' so that equivalence can be tested using '='. Jaranc follows the same notation as in other languages such as C, Java or Python which also use '==' to indicate an equivalence test.

Jarnac also includes the built-in boolean constants, `True` and `False`.

Operation	
'x and y'	Boolean and
'x or y'	Boolean or
'x xor y'	Boolean xor
'not x'	Boolean not
'<'	Less than
'<='	Less than or equal
'>'	Greater than
'>='	Greater than or equal
'<>'	No equal to
'=='	Equality test

Examples:

```
Option1 = True;
Option2 = False;

Choice = Option1 or Option2;

if Choice = True then .....

is equivalent to the shorter version:

if Choice then .....
```

The boolean operators, `and, or, not, xor` have a higher priority than relational operations such as `<, > etc`. This means it is important when mixing these operations in a single expression, that the relational tests are bracketed away from the boolean operator. For example:

```
if (5 > 1) and (5 < 10) then .....
```

is the correct way to write such an expression, where as '`if 5 > 1 and 5 < 10 then..`' will result in a runtime error. The reason for this is that in the unbracketed version the term `1 and 5` will be computed before the relational tests resulting in the intermediate expression, `if 5 > False < 10` which clearly makes no sense because it is not possible to test whether a boolean such as `False` is greater or less than an integer!

12 Basic Built-In Mathematical Operations

Jarnac includes a variety of built-in mathematical operations, such as the common trigonometric and logarithmic functions. Less common functions such as Cosh, ArcSinh etc and also some mathematical constants such as `e`, are included in a math module called math. This module can be made available with the instruction, `import math`. See `math.jan` for details.

All trigonometric functions expect radians.

sin (x)	returns the sine of x
cos (x)	returns the cosine of x
tan (x)	returns the tangent of x
sqrt (x)	returns the square root of x
sqr (x)	returns the square of x
log10 (x)	returns the logarithm to the base 10 of x
ln (x)	returns the natural logarithm of x
log2 (x)	returns the logarithm to the base 2 of x
exp (x)	returns e raised to the power of x
abs (x)	return the absolute value of x
fact (x)	return the factorial of x
gammaln (x)	return the natural log of the gamma function at x
comb (n, k)	return the number of combinations of n things taken k at a time
perm (n, k)	return the number of permutations of n things taken k at a time
min (v|m|x,y)	return the minimum number in a vector, matrix or number pair
max (v|m|x,y)	return the maximum number in a vector, matrix or number pair
mean (v|m)	return the mean of the numbers in a vector or matrix
sum (v|m)	return the sum of numbers in a vector or matrix
prod (v|m)	return the product of numbers in a vector or matrix
sumofsquares (v|m)	return the sum of squares of numbers in a vector or matrix
mod (x,y)	return the modulus of two numbers
tau (x)	return the value of 1/x ln (1/rnd)
float (x)	Promote an integer to a real
trunc (x)	demote a real number to an integer
real (c)	returns the real part of a complex number
imag (c)	returns the imaginary part of a complex number

Examples

```
x = sin (pi);

H = 1e-3;
pH = -log10 (H);
```

13 Builtin Constants

pi	Pi, ratio of circumference to diameter
degrees	180/Pi, use to convert degrees to radians, eg pi*Degrees
nl	newline string
true, false	Boolean values
inf	The value infinity as defined by IEEE-754
nan	The value Not A Number as defined bt IEEE-754
tol	Global tolerance for numerical routines
displayLabels	Global boolean flag to control display of matrix labels

Type constants used by IsType() function: tInteger, tDouble, tBoolean, tString, tVector, tMatrix, tList, tNetwork, tOde

14 Control Statements

Jarnac supports the following control statements, compound, conditionals(if/then/else), while/do, for/loop and repeat/until. The repeat, while and for loops also support a break statement. Unlike some languages,

indentation is *not* important and scripting style structure is format free.

14.1 Compound Statements

```
BEGIN
  <StatementList>
END
```

14.2 If Statement

```
IF <Expression> THEN
   <CompoundStatement> | <Statement>
[ELSE
   <CompoundStatement> | <Statement> ]
```

For example:

```
if a > 3 then
   println "A is greater than 3"
else
   println "No it is not";


if S1 > 100 then
   begin
   ...
   end
else
   ...
```

14.3 While Statement

```
WHILE <Expression> DO
      <CompoundStatement> | <Statement>
```

For example:

```
i = 1;
while i < 10 do
   begin
   i = i + 1;;
   println "Do something...";
   end;
```

While loops can optionally use a break statement to terminate the loop. For example:

```
i = 1;
while i < 10 do
   begin
   i = i + 1;;
   println "Do something...";
   // The break causes the while loop to exit to the fist
   // statement after the loop
   if i > 6 then break;
   end;
```

14.4 For Loops

There two forms of for loop, the traditional iteration over a range and an alternative `for x in` loop where the iteration is over a list or vector.

The traditional form uses the syntax:

```
FOR <VariableName> = <SimpleExpression> TO <SimpleExpression> DO
      <CompoundStatement> | <Statement>
```

For example:

```
  for i = 1 to 10 do
     println "Do something...";
```

The control variable must be an integer.

For loops can optionally use a break statement to terminate the loop, for example:

```
  for i = 1 to 10 do
      begin
      println "Do something...";
      if i = 6 then break;
      end;
```

The alternative form will iterate a control variable from a list or vector. For example:

```
for x in ["one", "two", "three", "four"] do
    print x;
one  two  three four

for x in vrange () do
    process x;
```

The syntax is:

```
FOR <VariableName> IN ( <List> | <Vector> ) DO
      <CompoundStatement> | <Statement>
```

14.5 Repeat/Until

```
REPEAT
  <StatementList>
UNTIL <Expression>;
```

For example:

```
  i = 10;
  repeat
     println "Do something...";
     i = i - 1;
  until i = 1;
```

Repeat loops can optionally use a break statement to terminate the loop. See while loop example.

14.6 Break Statement

The `break` statement may be used to forcibly exit loops. A break will terminate the enclosing loop.

```
i = 10;
repeat
   println "Do something...";
   if i < 2 then break;
   i = i - 1;
until i = 1;
// Break exits to here
```

Break may be used in `repeat`, `while` and `for` loops.

15 User Defined Functions

All user defined functions have the following structure:

```
FUNCTION Name [ ( ArgList ) ] ;
  <StatementList>
END
```

The argument list is optional. User defined functions can optionally return values to the caller using the keyword `return` followed by an expression. More than one return statement may be present in a user function. Once the program reaches a return statement the function exits to the caller. For example:

```
function Add (a, b);
  return a + b;
end;
```

or

```
function Test (a, b)
  if a > then 10 then
     return True
  else
     return False;
  // The following line will never get executed
  println "You'll never see me!";
end;
```

User functions that don't return any result may also be written, for example:

```
function WriteMessage (msg)
  println msg;
end;
```

Internally a user function that does not explicitly return something to the caller, returns the null object which is ignored by the runtime system.

15.1 User Function Arguments

When a user function is compiled by Jarnac, the arguments in a user defined function are considered typeless. This means, that during runtime, any type can be passed to a user defined function. In the simple example above, 'add' can be called with a variety of argument types. The following examples are all legal calls to 'add', and will yield the expected result:

```
Add (4, 5)
Add ({1, 2, 3}, {7, 8, 9});
Add (Ones (2), Ones (2)*2);
Add ("Live ", "Long");
Add ({1,2,3}, 6)
```

Integers, float, booleans and strings are called by **value**, this means alterations to these types of argument within the body of a function will **not** change their value outside the function.

On the other hand, arguments such as vectors, matrices and any object type (eg metabolic models) are called by **reference**, this means that alterations to these types of data within the body of a user function **will** change their value outside the function. This also means that passing large matrices will **not** incur a time penalty as the function is passed a reference to the matrix. If you wish to manipulate a matrix within a function but do not wish these changes to affect the matrix outside the function then you should first make a clone of the matrix (or vector or object) before making any changes. For example:

```
function MyFunc (m)
   // Cloning will protect m from alteration
   tmp = Clone (m);
   for i = 1 to rows (tmp) do
       for j = 1 to columns (tmp) do
           tmp[i,j] = 999;
end;

m = urndm (5,5); // Create a matrix with random values
MyFunc (m);
println m;      // We should see the original m here
```

15.2 Calling User Functions

Since user functions are internally treated as object references, the Jarnac compiler has to be able to distinguish, during compilation, between a reference to the function or an invocation (call) of the function. The rule is simple, to invoke a user function one must always use brackets after the function name, even if the user function has no arguments. In the following example all the statements will cause the particular function to be called:

```
myfunc();  // Note the empty argument list
CallMe (a, b, c);
AnotherFunc (4, 5, [1,2,3]);
```

To reference a function, that is refer to the function without causing the function to be called, simply omit the bracketing. Thus the following statements do not cause the user function to be called, but instead represent a reference to the user function:

```
myfunc;
CallMe;
AnotherFunc;
```

One particular advantage of function references is that they can be passed to other user functions in the argument list and then the function can be invoked within the user function.

A common error is to call a user function or built-in without supplying a set of closing brackets. For example the built-in function urnd() returns a random number between zero and one. It is an easy mistake to type instead urnd, however this will not return a random number but a reference to the built-in function.

16 Modules

Modules are a useful way of building libraries of functionality. Modules can contain definitions of user functions and/or lines of Jarnac code. A simple module would be:

```
Module Simple
```

```
    println "Loading Simple....";

    // Initialise a few things
    ....

    function test (x)
       return x^2;
    end;

  end
```

When a module is loaded, it is first compiled and any statements which are not part of user functions are executed. This allows initialisation computations to be performed before any module user functions are called by the user. User functions themselves will be added to the module's symbol table.

A Jarnac module starts with the keyword `MODULE` followed by the name of the module. The remainder of the module should consist of statements and/or user defined functions. Modules are usually stored as text files and when required to be used, they are imported using the import statement. Note that modules may not be nested and a file may contain only one module declaration. Thus each module must reside in its own text file.

A more elaborate example of a module is given by:

```
  Module Cycle;

  p = defn CyclePathway
        S1 -> S2; k1*S1;
        S2 -> S1; k2*S2;
      end;

  p.k1 = 1.2;
  p.k2 = 4.5;
  p.S1 = 1.0;
  p.S2 = 0.0;

  Jacobian = p.Jac;

  function ComputeValue (a);
    x = sin (a);
    y = x / 2.0;
    return y + 10;
  end;

  end
```

Modules are loaded using the import keyword. Thus to import the above module use the statement, `import Cycle` where the import name is the name of the file where the module is stored.

To access objects within a module, use the module name followed by the indirection operator, '.', followed by the object you wish to access. Thus to call the user function `ComputeValue`, use the statement:

```
x = Cycle.ComputeValue (2.3)
```

To access the Jacobian, use the statement

```
J = Cycle.Jacobian
```

17 Random Number Generation

All random number generators use the Marsaglia-Bray algorithm. This algorithm has a very long cycle period, of the order of 2^{250} or more.

Marsaglia, George and Arif Zaman. "A New Class of Random Number Generators", Annals of Applied Probability, vol. 1 No. 3 (1991), pp. 462-480.

```
urnd()          returns a random number 0 <= x <= 1.0
urndm (n, m)    Create an n by m matrix and fill with urnd numbers
urndv (n)       Create an n sized vector and fill with urnd numbers
rnd (x)         returns a random number 0 <= x (exclusive)
```

18 System Calls

```
eval (string arg | Code Type)   evaluate a string or code type argument
size (arg)                      Returns the size (length) of the argument
sizeof (arg)                    Returns the number of bytes occupied by argument
mem                             Returns the amount of memory allocated on the heap
timer                           Returns the current system timer in milliseconds
                                Resolution approximately to 1 ms
Symbols (SymbolPath)            Returns list of symbols, eg Symbols ("main")
del (SymbolPath)                Clear a symbol, eg del ("main.defaultmodel")
delall                          Clears all symbols from the workspace
vars                            Display a list of user introduced variables
fun                             Assuming the startup file has loaded, this will display
                                a list of the available built-in symbols, essentially it
                                executes, symbols ("builtinfuncs")
```

19 Console Commands

cmds	Display list of console commands
help	Display brief help message
?name	Quick help, eg ?sqrt
cd path	Change directory to path
cls	Clear the console screen
delete FileName	Delete a file
dir	Generate a file directory listing
edit name	Edit a file
exe FileName	Execute the external process, FileName, eg exe Notepad
list name	List a file to the console
mkdir path	Create a subdirectory
hardcopy name	print a file to the current printer
quit	Quit Jarnac
pwd	display the current directory
rename OldName NewName	Rename a file
rmdir path	Remove a directory, eg rmdir MyOldModels
run ScriptFile	Execute the script file ScriptFile (.jan is optional)
startup	Run the startup script
status	Retrieve module and machine stack status

20 Miscellaneous

Miscellaneous built-in functions:

AddHelp String	Adds a help string to a user defined function or module
IsType (Arg)	Returns the data type of the argument
Pause (n)	Pauses execution for n milliseconds
Beep	Issue audible beep
FloatToStr (x)	Convert float argument to string form
IntToStr (x)	Convert integer argument to string form
vrange (n|[])	Generate a range of values and return as a vector vrange (10) or vrange ([start, stop, step]) eg vrange ([5, 15, 2])
TimeNow	Returns the current time as a string
DateNow	Returns the current date as a string
Clone (p)	Clone object p and return clone, eg q = clone (p)

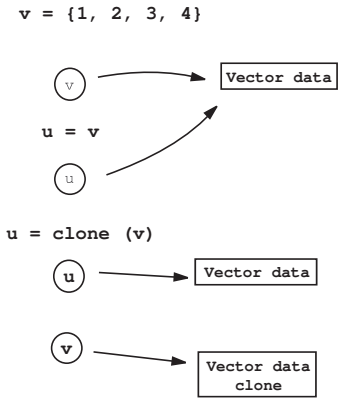
21 Stats Module

gauss (Mean, StdDev)	Generate a random number from a normal distribution
exprandom()	Generate a random number from an exponential distribution with unit mean
LinReg (x, y)	Perform linear regresion on vectors x, y, returns [slope, y-intercept]
Corr (x, y)	Compute correlation coefficient between vector x and y
PolyFit (x, y, pa, yfit)	fit polynomial with initial parameters, pa to data x, y, returns fitted y vector in yfit

22 Memory Model

22.1 Shared references and the Clone Method

All data types types except, integer, float, boolean and string, are termed reference types. To understand what this means, consider a variable which stores a vector, say, `v = {1,2,3,4}`. Let us make the assignment, `u = v`. What actually happens here is that because vectors are reference types, only a reference to `v` is copied to `u`, in other words the data held in `v` is not copied. See figure below:



Copying references is the default mode of operation and helps to speed up the execution of script files since it avoids the potentially lengthy act of copying data from one storage area to another. However, there is a pitfall to be aware of, since `u` and `v` reference the same data, any changes made to the data via `v` will automatically be reflected in the data referenced by `u` since `u` and `v` reference the same piece of data. For example:

```
v = {1,2,3};
u = v;  // Doesn't copy data but copies the reference

v[1] = 999
println v
  {999, 2, 3}
println u
  {999, 2, 3}
```

This is an important principle which users must be aware of. If, during the assignment, `u = v` the intention is to make an exact copy of `v` with a separate identity, then one should should use the clone method. This performs a so-called deep copy. The following lines show the clone method in operation:

```
v = {1,2,3};
u = clone (v);  // Makes a deep copy of v

v[1] = 999
println v
  {999, 2, 3}
println u
  {1, 2, 3}
```

The `clone` method makes a deep copy of the object. This means that all the data in an object is copied over to the left-hand side. `Clone` can take any object type as an argument and in the case of nested lists, it will clone recursively until all nested levels are copied.

23 Matrix and Vector Data Types

A **vector** is a simple list of numbers, for example:

```
{1, 2, 3}
{1.23, -6.7, 999}
```

Curly brackets are used delimit the vector and commas to delimit individual vector elements.

A **matrix** is a two dimensional list of numbers, with n rows and m columns – where n need not necessarily equal m. The syntax for specifying matrices or vectors is illustrated in the examples below:

```
{{1, 2, 3},
 {4, 5, 6},
 {7, 8, 9}}

{{1.23, 6.7}, {876, -321}}

{1, 5.6, 9.8}
```

Note the use of curly brackets to delineate the rows and columns. A ',' is used to separate individual rows.

Matrices and vectors can be empty, as in:

```
v = {}
m = {{}}
```

When defining a matrix or vector, the elements can be constants, variables or simple expressions, in the case of expressions, the expression must resolve to an integer or float type. Examples:

```
v = {pi, pi+1, pi+2}
m = {{a+b/2, pi/2}, {a-b, 6.7}}
```

To create space for a vector or a matrix one can use the functions `vector()` and `matrix()`; both these functions take integer arguments to indicate the dimensions of the vector or matrix, for example:

```
v = vector (100);  // create a hundred element vector

m = matrix (5,6);  // create a matrix with 5 rows, 6 columns
```

Vectors and matrices created in this way have all their elements set to the value zero.

24 Matrix/Vector Operations

The individual elements of a vector or matrix can be examined or set using simple indexing, For example, if `n = {1, 2, 3}`, then:

```
n[3] = 4;
result = n[3] + Log10 (1000);
```

Indexing of matrices is by row/column, thus `m[3,4]` indicates the element at row three and column four. Example:

```
m = {{1, 2, 3},
     {4, 5, 6},
     {7, 8, 9}};

m[1,2] = 2.5;
result = m[2,3] + 5.6;
```

Indices can be constants or expressions, for example:

```
m = {{1, 2},
     {4, 5},
     {7, 8}};

for i = 1 to 2 do
    m[1,i] = m[2,i];
```

24.1 Extracting and copying rows to and from a Matrix

Individual rows of a matrix can be accessed by omitting the column index during indexing, thus if m is a matrix then $m[2]$ refers to the second row in the matrix. The assignment $v = m[2]$ means extract the second row and copy it into a vector. Copying rows into a matrix is equally simple, to copy the vector, v, into matrix m at the fifth row, use the assignment, $m[5] = v$. Note that the original contents of the fifth row are lost in this operation.

Three functions, InsertRow, DeleteRow and AppendRow can be used to add, remove or append new rows to a matrix, thus

```
m = {{1,2,3}};
AppendRow (m, {4,5,6});  // Add a second row to the matrix
InsertRow (m, {9,8,7}, 1); // Insert a new row before row one in matrix m
DeleteRow (m, 1); // Delete row one
```

These functions can also be used to manipulate vectors, thus the statement `appendrow (v, {1,2,3})` will append the vector `{1,2,3}` to the vector v.

24.2 Standard Matrix Operations

`Ident (n)`	Returns an identity matrix of size n
`Ones (n)`	Returns a matrix of size n filled with ones
`Zeros (n)`	Returns a matrix of size n filled with zeros
`Matrix (n,m)`	Returns an empty matrix of size n by m
`Diag (n, k)`	Returns a diagonal matrix if size n with diagonal elements, k
`Sum (m)`	Returns the sum of the elements of a matrix or vector
`Prod (m)`	Returns the product of the elements of a matrix or vector
`Mean (m)`	Returns the mean of the elements of a matrix or vector
`SumOfSquares (m)`	Returns the sum of squares of the elements of a matrix or vector
`Rows (m)`	Returns the number of rows in matrix m
`Columns (m)`	Returns the number of columns in matrix m
`tr (m)`	Returns the transpose of matrix m
`vm (m)`	Display matrix graphically
`getRowName (m, r)`	Return the row label of matrix m at row r
`getColumnName (m, c)`	Return the column label of matrix m at column c
`setRowName (m, r, str)`	Sets the row label at r of m to str
`setColumnName (m, c, str)`	Sets the column label at c of m to str
`aug (m1, m2)`	Returns the augmented matrix from arguments m1 and m2

24.3 Matrix Manipulation

AppendRow (m, v)	Appends the vector v to the bottom of the matrix m
InsertRow (m, rr, v)	Inserts a vector v immediately before position rr
DeleteRow (m, rr)	Deletes row rr from matrix m
Matrix (n,m)	Create and return a matrix of given size
Vector (n)	Create and return a vector of given size
Dim (v|m)	Returns the dimension of a matrix or vector as a vector

24.4 Advanced Matrix Operations

det (m)	Returns the determinant of matrix m
rank (m)	Returns the rank of the matrix m
echelon (m)	Returns the fully reduced echelon of matrix m
reorder (m)	Returns reordered matrix m, independent rows moved to the top
lu (m)	Compute LU decomposition, returns list in the form [L, U, P]
ns (m)	Returns the null space of matrix m
svd (m)	Returns the SVD of matrix m in the form of a list [u,w,v]
eigenvalues (m)	Returns the eigenvalues of m as an array
eigenvectors (m)	Returns the eigenvectors of m as an array of vectors

Many of the advanced matrix routines are computed using the industry strength IMSL library.

25 Numerical Analysis

df (List, Symbol)	Numerically differentiates the argument list with respect to Symbol eg df (<x^2>, x)
sdf (List, Symbol)	Same as df() but scaled by sym/List
solve (UserFunction, y)	Solve a set of non-linear functions Solve returns the euclidian norm of the left-hand sides

The function solve can be used to solve a set of non-linear equations. Solve takes two arguments, the name of the user function which defines the set of non-linear equations, and a vector y of initial values for the dependent variables. An example should make this clear:

```
// This is the user function containing the non-linear equations
function MyFunc (y, yd)
   yd[1] = 4.5 - y[1]*1.2;
   yd[2] = y[1]*1.2 - y[2]*3.4;
end;

// y is the solution vector
y = vector (2);
// Initialise the solution vector with some suitable values
y[1] = 5.0; y[2] = 10.0;

// ...and solve
norm = solve (MyFunc, y);
```

```
// y now contains the solution, the return value norm will equal
// the euclidian norm of the left-hand side of the equations,
// the closer to zero the better the solution.
println "norm=", norm, "solution = ", y;
```

The user function must accept two arguments, the first argument is a vector containing the values of the dependent variables. This is an input argument and must not be altered in the user function but should be used to compute the current value of the non-linear equations. The second argument is a predefined vector which the user function must fill with the left-hand sides of the non-linear equations.

26 Jarnac Objects

Jarnac supports a limited concept of object types. In the current release all object types are built-in and it is not possible to introduce user defined object types. Moreover the current object model does not support inheritance. However Jarnac objects do provide a convenient way to organize the functionality of particular data types. In particular, networks, lists, files, user functions, graphs and ode models data items[1] have associated with them a set of fields, namely, methods, functions and properties which allows a user to manipulate the data contained within the object. Specifying a particular field of an object is achieved through the indirection operator, '.'. For example, the statement, `p.xtitle = "X Axis"`, attempts to invoke the property xtitle on the object p. If object p does not include the property xtitle then an error is generated. In this case xtitle refers to a property of the graphing object, thus p should be a reference to a graph object (eg as generated by the statement, `p = NewGraph`). The three different types of fields associated with objects, namely Methods, Functions and Properties have the following features:

Method Field - A method field may take arguments but returns nothing to the caller, e.g p.method (a, b), or g.method

Function Field - A function field may take arguments but also returns a result to the caller, e.g x = p.func (a,b), or x = h.func

Property Field - A property field takes no arguments but has a value which can be assigned or examined. Properties act very much like variables. Properties may have different types, thus one property may be a float type while another a matrix type. For example:

```
->p.prop = 1.2
->println p.prop + 10.0
11.2
->p.propm = matrix (2,2)
->p.propm[1,1] = 99
->prinln p.propm
[[99, 0]
 [0,  0]]
->
```

The names of all methods, functions and properties available to a particular object can be obtained using the methods, functions and properties methods, which are standard methods available to all objects. See next section for details.

26.1 Standard Methods

There are three standard methods which are available to all object types. These methods are:

[1] This list may be expanded in future releases to include matrices and vectors

`methods`	Returns a list of the names of the object methods
`functions`	Returns a list of the names of the object functions
`properties`	Returns a list of the names of the object properties

Example:

```
->a = [1,2,3]
->a.methods
 ["clear"]
```

26.2 Build-in Object Types

`Lists`	`Stores any data type (including lists) in sequential order`
`File`	`File Objects for streaming data to and from files, console and printers`
`Sys`	`System Object`
`User Functions`	`User defined functions`
`Cellular Networks`	`Models of biological cells`
`Graphs`	`Data visualisation objects`

26.3 User Functions as Objects

User Functions are implemented as Jarnac objects. The most important aspect of this feature is that user functions can be passed around just like ordinary variables. For example it is possible to pass user functions as arguments to other user functions. Consider for example the following code fragment:

```
function MyFunc (x)
  return pi*x;
end;

function CallMe (f, x)
  return f (x);
end;

CallMe (MyFunc, 9);
```

The code defines two functions, one called `MyFunc` which performs a very simple operation on its single argument and a second called `CallMe` which appears to treat one of its arguments as a function call, `f(x)`. The example then illustrates a call to `CallMe`, passing the argument `MyFunc`. `MyFunc` without an argument list is treated as a reference to the user function object. However, inside the function `CallMe` it has arguments applied to it (or just `()` if there were no arguments) which results in the user function being called.

Since user functions are Jarnac objects it is also possible to assign user functions to other variables. The following example should make this clear:

```
function MyFunc (x)
  return pi*x;
end;

z = MyFunc;

z (10);
31.415926
```

The only check that is made when a function object is passed to another function is that the number of expected arguments in the internal function call match.

This functionality enables one to do tricks like this:

```
function MyFunc (x)
  return x*x;
end;

a = [];
for i = 1 to 5 do a.append (MyFunc);

for i = 1 to 5 do println a[i](i);
1
4
9
16
25
```

26.4 List Objects

Lists are ordered collections of data where the data can be any valid Jarnac data type, including other lists. Examples include:

```
[1,2,3,4]
[1, 5.6, "str", {1,2,3}, [1,2,3]]
[]
```

Lists can be assigned to variables, for example

```
a = [1,2,3,4]
b = [] // Empty list
c = ["abc", [1,2,3, ["str", "xyx"]]]  // Nested lists
```

Once a list is stored in a variable name a variety of operations become available. For example, individual items in the list can be set or examined by simple indexing, eg if `a = [8, "str", True]` then `a[2]` will equal `"str"`. To set a specific item simply use expressions like, `a[2] = vector (100)` which in this case replaces the second item with a vector of 100 elements. Note that indexing begins at one. The index may be a constant or variable that resolves to an integer, eg `a[i]`

A summary of all operations include:

`a = [1,2,3]`	Creation
`a[1]`	Indexing
`b = [1,[2,[3]]]`	Nesting
`b[2][2]`	Nested indexing
`[1,2] + [5]`	Concatenation
`2 * [1,2,3]`	Repeated concatenation eg [1,2,3,1,2,3]
`size (a)`	Number of elements
`a.count`	Number of elements, same as size (a)
`a.append (x)`	Append a with item x, eg a.append ("str")
`a.head`	Returns the first element of the list
`a.tail`	Returns the last element of the list
`a.delete (i)`	Delete item at position i
`a.remove (item)`	Delete the specific item from the list, eg a.remove ("str")
`a.reverse`	Reverse the order of items
`i = a.find (item)`	Search for item and return index, -1 if not found
`a.clear`	Clear the list of all items
`a.load ("xyz.dat")`	Load list object from a file
`a.save ("xyz.dat")`	Save list object to a file

Save/Load may require some explanation. Save allows one to save a list object to file storage and load will load a list object from file storage. If the specified file does not contain a valid list object, an error is reported. For example

```
a = [vector (1000), vector (100), vector (10)]
a.save ("vec.dat");
a.clear
a.load ("vec.dat")

b = []
b.load ("vec.dat")
```

These operations are useful for storing any form of Jarnac data to disk for permanent storage. This includes things like user functions and networks.

For temporary storage of objects, one can use the user stack, thus:

```
a = [pi, pi*2, pi*3, pi*4]
push (a)
a = 1
// calculations involving a.....

a = pop;  // Retrieve original value of a
```

Note that data saved using the user stack does not persist between Jarnac sessions. If you wish to save objects etc between different Jarnac sessions then it is important that the save/load methods be used instead.

26.5 Sys Object

The Sys object is a built-in object which provides certain system dependent information. There is no need to create the sys object, it is created automatically at startup and is always available. Be careful if you wish to access the sys object from within a user function. Since sys is defined outside the user function access to sys can only be made if sys is made an external reference in the user function. Thus to access sys within a user function, make sure that one of the first lines in the user function is the statement, `extern sys`.

`sys.path`	Current directory path
`sys.ModulePath`	Current module search path
`sys.version`	Returns a string indicating the current version number
`sys.prompt`	A property for setting or examining the current console prompt
`sys.copyright`	Returns the copyright notice
`sys.dformat`	Default format code when outputting floating point values
`sys.DefaultGraphWidth`	When a graph is created, this is it's default width
`sys.DefaultGraphHeight`	When a graph is created, this is it's default height
`sys.ScreenWidth`	The current screen width in pixels
`sys.ScreenHeight`	The current screen height in pixels

26.6 File Management

`DeleteFile (FileName)`	Deletes the file given by the file name argument
`CopyFile (SrcFileName, DestFilename)`	Copies the source file to the destination file

26.7 Stream Object

The stream object is a data stream which can be attached to files, printers and the console device. Five stream constructors are provided:

`f = CreateFile ("xyz.dat")`	Create a stream to a new ASCII file for write only operation
`f = OpenFile ("xyz.dat")`	Open a stream onto an existing file for read/write ASCII operations
`f = OpenObjectFile ("xyz.dat")`	Open a stream onto an existing file for read/write object operations
`f = OpenPrinter`	Open a write only stream on to the current default printer
`f = OpenConsole`	Open a stream to the console device

The Stream object is still under development but currently supports the following operations.

`f.read (a, b, ...)`	Read items from the stream f, items must be in ASCII format
`f.write (a, b, ...)`	Write items to the stream f in ascii format
`f.writebinary (a,b, ...)`	Write items to the stream f in binary format
`f.close`	Close the stream
`f.length`	Returns the length of the stream
`f.filename`	If the stream is attached to a file, this return the file name
`f.eof`	Returns true if at end of the file stream
`f.seek (p)`	Sets the position of the read cursor of the file stream
`f.position`	Returns the position of the stream pointer
`f.landscape`	If the stream is attached to the default printer, this sets landscape mode
`f.portrait`	If the stream is attached to the default printer, this sets portrait mode
`f.readchar`	Reads and returns a string containing a single character from the file stream
`f.writechar (ch)`	Writes a string containing a single character to the file stream
`f.readline`	Reads and returns a line (up to NL) from the file stream
`f.readfile`	Reads and returns the entire contents of a file stream as a string

In the current implementation, all read/write operations exchange data in ASCII format. For the moment use the List Save/Load methods to store and retrieve data in binary format.

Examples:

```
f = CreateFile ("data.dat")
f.write (vector (1000))
f.close;

// Open an existing file and read in the data
f2 = OpenFile ("data.dat");
f2.read (v);
f2.close;
```

26.8 Graphing Objects

The graphing objects are used to visualise data, either as 2D graphs or using a limited degree of 3D graphing. The following graph object constructors are provided:

`g = NewGraph`	Create a 2D graphing panel in the centre of the screen
`g = NewGraph (x, y)`	Same as NewGraph except x,y specify the panel position in normalised screen coordinates
`g = NerwGraph (x, y, w, h)`	Same as NewGraph except x,y,w,h specify the panel position and size in normalised screen coordinates
`g = NewGraph (True|False)`	Same as NewGraph but with a boolean argument indicating whether to make the panel visible or not (see show)
`g = NewGraph (x, y, True|False)`	Same as NewGraph (x, y) but with visibility flag
`g = NewGraph (x, y, w, h, True|False)`	Same as NewGraph (x, y, w, h) but with visibility flag

Examples:

```
// Open a graph panel at the top left of the screen
g = newgraph (0, 0);
// Open a graph panel and position it 1/4 down, 1/3 across the screen
g = newgraph (0.25, 1/3);
```

The 2D Graphing object (NewGraph) supports the following operations. All coordinates are expressed in graph coordinates.

Methods:

`MoveTo (x, y)`	Move drawing cursor to coords x, y
`DrawTo (x, y)`	Draw a line from the current cursor position to x, y
`DrawLine (x1,y1,x2,y2)`	Draw a line between two points
`Mark (u, v, MarkId)`	u and v are vectors, mark places a mark at u[i],v[i] eg p.Mark (u, v, 1), MarkId is a positive integer indicating type of mark symbol to use (there are 26 in all)
`Join (u, v)`	u and v are vectors. Join connects u[i],v[i] to u[i+1],v[i+1] with straight line segments eg p.Join ({1,2,3}, {4.5, 6.7, 7.8})
`Circle (x,y,r)`	Draw a circle at x,y of radius r
`Bar (v)`	Draw a bar for each element in the vector argument v
`Graph (m)`	Graph the data in matrix m to the graph panel associated with with the graph object, eg p.Graph (m), the data in the matrix is copied to the graph object
`EnableLegend`	Applicable to Graph()
`DisableLegend`	Applicable to graph()
`GraphSelect (m, [])`	Same as Graph but with additional list argument selecting the columns of m to graph, eg. p.GraphSelect (m, [1,2,3]) means graph columns 1, 2 and 3. The first column indicated in the list is always treated as the x coordinate.
`Select ([])`	Change the selection of columns in the object data eg p.select ([2,3])
`Draw`	Redraw the graph using the current data held in the the object, eg p.draw
`Hide`	Hides the graphing window, eg p.Hide
`Show`	Shows the graphing windows, i.e makes it visible eg p.show

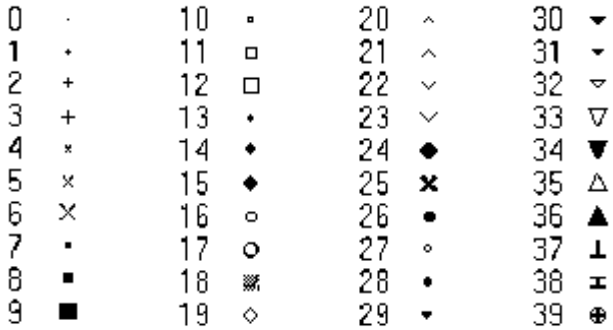
Properties:

`Title`	Set the main title of the graph, eg p.title = "Main Title"
`XTitle`	Set the X axis title, eg p.xtitle = "X Axis"
`YTitle`	Set the Y axis title, eg p.ytitle = "Y Axis"
`XGridOn`	Apply a vertical grid
`XGridOff`	Remove the vertical grid
`YGridOn/YGrifOff`	Apply or remove a horizontal grid
`LineColor`	Set the line color, eg p.linecolor = "red"
`MarkColor`	Set the marker color
`ChartColor`	Set the chart background color
`FillColor`	Set the fill color for circles and bars
`Data`	Gives access to the data stored in the object. The graph method, p.graph (m), as well as rendering the graph, copies the data to the data property. One can later use the Data property to access or set individual data points or even the entire set. eg p.Data = urndm (100, 5), to set the entire set or use p.Data = matrix (100, 5) to setup an empty array x = p.Data[1,2], to access an individual data element p.Data[1,2] = sin (30*degrees), to set an element
`xmin`	Specifies of minimum x axis value, eg p.xmin = 0.0
`xmax`	Specifies of maximum x axis value, eg p.xmax = 1.0
`ymin`	Specifies of minimum y axis value, eg p.ymin = -10.0
`ymax`	Specifies of maximum y axis value, eg p.ymax = 20
`axes`	Vector property, used to set all axes limits with one instruction, eg p.axes = 0, -10, 1.0, 20

Currently available colours:

Red	Blue	Yellow	Green
Lime	Navy	Purple	

Currently available markers:



26.9 Visual Slider Objects

The Slider method will return a visual slider object.

`value`	Current value of the slider
`max`	Max value on the slider scale
`min`	Minimum value on the slider scale
`changed`	Set true if slider has recently changed, reset when value is inspected
`OnChanged`	User function pointer, set this is a user function that will be called each time the slider changes value
`hide`	Hide the slider
`show`	Make the slider visible
`close`	Destroy the slider

26.10 Miscellaneous Input/Output Statements

`ReadKey`	`Suspends program execution until user hits a key, character is returned to caller, eg ch = ReadKey`
`ReadString`	`Suspends program execution until user enters a string followed by enter key, string returned to caller, eg try, println readstring`
`graph (m <,[..]>)`	`Graph the matrix argument m to the default graph panel This graph statement has no relation to the previosuly discussed NewGraph Object`

26.11 Miscellaneous Storage Functions

`Push (arg)`	`Push the argument onto the user stack, e.g. Push (1,2,3)`
`Pop`	`Pop the item from the user stack, e.g. v = Pop`
`save (x, filename)`	`Save item x as a binary object to the file, filename, eg save (m, "m.dat")`
`load (filename)`	`Returns the binary item loaded from the file, filename, eg m = load ("m.dat") Note: to save more than a single item, append the items to a list object and save the list`

Save/load will save and load items in binary format. Write will export data in text format. Examples:

```
m = {1, 2, 3, 4};
save (m, "data.dat");
... do something else with m...and recover the old m with...
m = load ("data.dat");

m = [1, {1,2,3}, [True, defn cell ... end]];
save (m, "list.dat");

m = load ("list.dat");
```

27 Exception Handling

Jarnac supports exception handling. Exceptions are a controlled means of breaking out of the normal flow of control in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code or by any other code that directly or indirectly invoked the code where the error occurred.

When ever a runtime errors occurs, Jarnac raises an exception. Alternatively users may raise exceptions through the use of the `raise` statement. Exception handlers are specified with the `try ... except <ExceptionCode> end` statement.

When an exception is not handled at all, Jarnac returns to its interactive main loop where a built-in outer exception handler will display the exception message to the console.

Here is an example of an exception handler:

```
try
  a = 1.4;
  b = 0.0;
  // Cause an exception, divide by zero
  c = a/b;

except eMathError
   println "Math Exception: " + Sys.ExceptionMsg;
end;
```

or one can be much more specific

```
try
  a = {{1,2},{2,4}}
  // Cause an exception by inverting a singular matrix
```

```
    b = 1/a;

  except ESingularity
     println "Matrix Inversion Exception: " + Sys.ExceptionMsg;
  end;
```

A couple of points:

- The protected code is written between a `try`, `except` block.
- The exception to be handled is specified by placing the name of the exception after the `except` keyword.
- The code to handle the exception is placed between the `except` keyword and the terminating `end`.
- Details of the most recent exception are stored in Sys.ExceptionMsg

The following list gives the current range of exceptions which can be handled, this list is likely to grow in future releases.

Exception	Description
`eAnyError`	Handles any runtime error
`eMathError`	Handles any math runtime error
`eGeneralError`	Handles any non-math runtime error
Math Exceptions	
`eDivideByZero`	Division by zero exception
`eIntDivideByZero`	Integer division by zero
`eOverFlow`	Floating point overflow Exception
`eMRangeError`	Maths range error, eg sqrt (-9), log (-100)
`eSingularity`	Matrix singularity exception
General Exceptions	
`eValueError`	Values uninitialised or incorrect during computation
`eTypeError`	Incorrect data type applied during operation
`eInternalError`	Internal error, eg access violation (contact author!)
`eArgumentError`	Incorrect argument passed to function

If you wish to raise your own exception use the statement `raise <ExceptionCode>`, for example

```
  raise EMathError;
```

28 Network Models

The main reason for developing Jarnac is to support the construction, manipulation and analysis of biological cellular models, including metabolic, signal transduction and gene pathways. This facility is enabled by the network objects (metabolic models). The following script illustrates how a metabolic model is declared, initialised and it's time course evaluated.

```
// Define the pathway model
p = defn Pathway
      // Declare the floating and boundary species
      var S1, S2; ext X0, X1;
```

```
      // Declare each reaction and its associated rate law
      J1: X1 -> S1; k1*X0 - k2*S2;
      J2: S1 -> S2; Vm*S1/(Km + S1);
      J3: S2 -> X1; k3*S2^n;
    end;


// Now initialise all the parameter and variables
p.X0 = 1.2; p.X1 = 0.0;
p.S1 = 0.001; p.S2 = 0.001;

p.k1 = 1.2; p.k2 = 5.6;
p.Vm = 10.5; p.Km = 0.4;

p.k3 = 5.6; p.n = 3.5;

// Define some convenient variables
TimeStart = 0;
TimeEnd = 10;
NumberOfDataPoints = 100;

// ...  perform the simulation
m = p.sim.eval (TimeStart, TimeEnd, NumberOfDataPoints, [<p.Time>, <p.S1>, <p.S2>, <p.J1>]);

// eval returns a matrix. Graph the time course using graph
graph (m);
```

There are two ways to declare metabolic models, named or default.

28.1 Default Models

Default declarations are recommended for the novice as it reduces the amount of typing that has to be done. An example of a default model is:

```
DefaultModel cell
  [J1] $X0 -> S1; k1*X0;
  [J2] S1 -> $X1; k2*S1;
end;
```

A default model starts with the keyword, `DefaultModel` followed by a model identifier, in the example this is given by, `cell`. Following this is the model itself which comprises of a list of reactions specifications. The list is terminated with the keyword, `end`.

28.2 Specifying Model Reactions

Within the body of a default or named model, details of the kinetic model are given. The syntax should be almost self-explanatory. Each reaction in a model is specified using three terms, the name of the reaction, the stoichiometric details of the reaction and the rate law which governs the rate of the reaction. A semicolon must terminate the stoichiometric details and a semicolon must also terminate the reaction rate law.

28.2.1 Reaction Name

Naming a reaction is optional, but if a name is not specified, Jarnac will generate an internal one for itself. Naming a reaction is important because when a model is declared, the reaction name can be used to refer to the rate through the reaction.

There are two ways to specify the name of a reaction: one can use the SCAMP syntax of square brackets; or an alternative and simpler syntax which uses a semicolon. The two approaches are illustrated below:

```
[GlucoseFlux] Glucose -> .....
 GlucoseFlux: Glucose -> .....
```

The choice of syntax is entirely up to the user.

28.2.2 Reaction Stoichiometry

The second component of a reaction is the stoichiometric specification, that is the number and types of species which take part in the reaction. This is written using a familiar chemical reaction notation, that is a left side indicating the reactants and a right side indicating the products. The reactants and products are separated by a conversion symbol of which there are two to choose from, one to indicate a reversible reaction (`=>`) and an other to signify an irreversible reaction, `->`. The reversiblity of a reaction as specified by the conversion symbol is only used for structural analyses, particularly the computation of elementary modes. It does no reflect actual dynamic reversiblity during a simulation, this is indicated instead by the particular rate law chosen for the reaction.

The examples below illustrate some reaction declarations:

```
Citrate -> IsoCitrate
Glucose + ATP -> Glucose_6_P + ADP
2 A + 3 B -> 3C + D
A -> B + 5 C
A => B
```

The examples also illustrate how one indicates stoichiometries other than unity, this is accomplished simply by proceeding the species name with an integer number indicating the stoichiometry. In the current release, the stoichiometry must be an integer constant.

Boundary species can be indicted by proceeding the name of the species with a dollar symbol, for example, `$Glucose`. Alternatively boundary species can be predeclared using the `ext` keyword. Further details of this and other declarations is given in a later section.

28.2.3 Reaction Rate Laws

The third component of a reaction is the rate law which governs the rate of the reaction. Rate laws are specified using normal algebraic equations, for example:

```
k1*S1
Vmax*S1/(S1 + Km)
Sin (S1*Time);
```

Jarnac also has built-in rate laws, see the section below for details.

28.3 Access to Model Properties

Within the body of the main program access to the various properties of the model is very straightforward. To assign a value of 5.6 to the rate constant `k1`, simply use the statement, `k1 = 5.6;`. The same applies to metabolite concentrations, thus to initialise a default model one could use:

```
k1 = 5.6;
k2 = 0.5;
X0 = 1.0;
X1 = 0.0;
S1 = 0.1;
```

28.4 Named Models

A second approach to defining metabolic models is by declaring **named** models. An example of a named model declaration is:

```
p = defn cell
  [J1] $X0 -> S1; k1*X0;
  [J2] S1 -> $X1; k2*S1;
end;
```

A named model starts with an identifier which is used to reference the model later on. In this case the identifier, `p` is used, however any suitable identifier could be used here, eg `MyModel, Cell1_2a`. Following the model reference is an equals symbol representing an assignment operation, followed by the keyword, `defn` and then a label for the model, in this case, `cell`. The model label bears no relationship to the model reference name. Whereas the reference name can change - through subsequent variable assignments - the model label is fixed thoughout the life of the model.

The rules for writing the reactions of a model are the same as for default model7, thus a reaction should have an identifier, a stoichiometric specification and a rate law. The main difference to a default model is that for a named model, when referencing model properties, each model property must be prefixed with the name of the model, thus to initialise model p one would have to be write:

```
p.k1 = 5.6;
p.k2 = 0.5;
p.X0 = 1.0;
p.X1 = 0.0;
p.S1 = 0.1;
```

One advantage of named models is that they can be passed as arguments in user defined functions since the model name is treated as a Jarnac variable. For example:

```
function Init (p)
  p.k1 = 4.5;
  p.k2 = 0.5;
  p.S1 = 0.12;
end;
```

Note that metabolic models are passed to functions by reference, therefore changes made to p in the body of the function will result in equivalent changes to the model external to the function. It is also of interest to note that the `return` statement makes a copy of whatever object it is asked to return, this means that inserting `return p` in the above function would result not only changes to the original model but also the creation of an identical copy returned to the caller. To use this new copy one would need to use a statement such as:

```
r = Init (p)
```

where r now contains an identical copy of p. This is one way to clone a model, although an easier approach is simply to use the clone method, thus:

```
r = p.clone
```

Since each named model will reside in it's own memory space, named models also permit more than one model to coexist in a single Jarnac session.

28.5 Built-in Rate Laws

Evaluating kinetic rates laws is an expensive operation for Jarnac, it is therefore advantageous to use, if possible, the built-in rates laws. Built-in rate laws evaluate approximately twice a fast as the explicit forms so if possible I would recommend that the reader use them.

There aren't many built-in kinetic rate laws in release 2.0, it is hoped that this will be improved upon in future releases.

```
uui (S1, Vm, Km)                                    Uni-Uni Irreversible
uur (S1, S2, Vm, Km1, Km2, Keq)                     Uni-Uni Reversible
bbi (S1, S2, Vm, Km1, Km2, Kia)                     Bi-Bi Irreversible
hill (S, Vm, K, h)                                  Hill Equation
unibi (A, P, Q, Vmf, Vmr, Kma, Kmp, Kmq, Kip, Keq)  Uni-Bi Reversible
biuni (A, B, P, Vmf, Vmr, Kma, Kmb, Kmp, Kia, Keq)  Bi-Uni Reversible
```

28.5.1 Single Substrate Irreversible Michaelis-Menten

The built-in function `uui` evaluates the single substrate Michaelis-Menten model (Vm S/(S + Km)). The arguments must be in the order:

```
uui (S1, Vm, Km);
```

28.5.2 Single Substrate Reversible Michaelis-Menten

The built-in function `uur` evaluates the single substrate reversible Michaelis-Menten model, given by the equation:

$$v = \frac{Vm/Km_1(S - P/Keq)}{1 + S/Km_1 + P/Km_2}$$

The arguments to `uur` must be in the order:

```
uur (S, P, Vm1, Km1, Km2, Keq);
```

28.5.3 Hill Equation

The built-in function `hill` evaluates the Hill equation:

$$v = \frac{VS^h}{K_{0.5}^h + S^h}$$

The arguments to `hill` must be in the order:

```
hill (S, V, K, h)
```

28.5.4 Irreversible Bi-Bi

The built-in function `bbi` evaluates the two substrate irreversible Michaelis-Menten model, given by the equation:

$$v = VAB/(K_{ia}K_b + K_bA + K_aB + AB)$$

The arguments to `bbi` must be in the order:

```
bbi (A, B, V, Ka, Kb, Kia)
```

28.5.5 Reversible Ordered Uni-Bi

The built-in function `UniBi` evaluates the one substrate, two products reversible Michaelis-Menten model. The products are assumed to come off the enzyme in strict order, namely P followed by Q. The law given by the equation:

$$v = \frac{\text{Vmf } (A - P\ Q/\text{Keq})}{(\text{Kma} + A(1 + P\ \text{Kip}) + (\text{Vf}/(\text{Vr Keq}))(\text{Kmq}\ P + \text{Kmp}\ Q + P\ Q))}$$

where Vmf and Vmr are the forward and reverse maximum rates respectively.

The arguments to `UniBi` must be in the order:

`uibi (A, P, Q, Vmf, Vmr, Kma, Kmp, Kmq, Kip, Keq)`

28.5.6 Reversible Ordered Bi-Uni

The built-in function `BiUni` evaluates the two substrate, one product reversible Michaelis-Menten model. The products are assumed to bind to the enzyme in strict order, namely B followed by A. The law given by the equation:

$$v = \frac{\text{Vf } (AB - P/\text{Keq})}{(AB + \text{Kma}\ B + \text{Kmb}\ A + (\text{Vf}/(\text{Vr Keq}))(\text{Kmp} + P(1 + A/\text{Kia})))}$$

The arguments to `BiUni` must be in the order:

biuni (A, B, P, Vmf, Vmr, Kma, Kmb, Kmp, Kia, Keq)

28.5.7 Example

```
DefaultModel cell
  [J1] $X0 -> S1; uui (X0, Vm1, Km1);
  [J2] S1 -> $X1; uur (S1, X1, Vm2, Km2, Km3, Keq);
end;

Vm1 = 6.7; Vm2 = 0.3; ...etc.
```

28.6 Predeclarations

Jarnac allows a number of predeclarations of symbols to be made before the model reactions are defined. Three declarations are allowed:

`var`	Declare floating molecular species eg `var ATP, ADP, AMP;`
`ext`	Declare fixed or boundary molecular species eg `ext Glucose, Lactate;`
`vol`	Declare volume compartments eg `vol Cytosol, Mitochondria;`

Example

```
p = defn cell
    var ATP, ADP, AMP;
    ext Glucose, Lactate;

    [Kinase] ATP + ADP -> 2 ADP; v;

    [J1] Glucose -> Intermediate; v1;
    [J2] Intermediate -> Lactate; v2;
end;
```

28.7 Compartmentation

Unless otherwise indicated, Jarnac will declare an internal 'universal' compartment of unit volume and all molecular species will be assumed to be located in this compartment.

Some models however require multiple volumes, and in these situations we need some way to declare the volumes and also some way to allocate the various molecular species to the different volumes.

Declaring new volume compartments is easy, simply use the `vol` keyword, for example:

```
vol Blood, Cytosol;
```

After the `vol` keyword, we just list the names of all the compartments in the model.

To indicate the location of a molecular species in a particular volume use a modification of the `var` keyword. If a species, say `Sp1`, is located is a compartment, `Vol1`, we indicate this using the `in` keyword in combination with `var` as shown below:

```
vol Vol1;
var Sp1 in Vol1;
```

Obviously, the volume compartments must be declared before declaring the molecular species. The phrase, `Sp1 in Vol1` tells Jarnac that `Sp1` is located in compartment, `Vol1`.

If you wish to employ multiple compartments then you must make sure that the rate laws return rates in terms of amount rather than concentration.

Example compartment Model.

```
// Two step linear chain - Volume Handling Tests
// The simulation compares a numerical solution to the
// known analytical solution

p = defn ThreeStep
      vol VA, VB, VC;
      var A in VA, B in VB, C in VC;

      [J1] A -> B; k1*A*VA;
      [J2] B -> C; k2*B*VB;
end;

// Volumes MUST be set before the concentrations

p.VA = 10;
p.VB = 5;
p.VC = 1;

p.A = 1.0; Ao = p.A;
p.B = 0.0;
p.C = 0.0;

p.k1 = 7.5; p.k2 = 10.5;

t = 0; hstep = 0.02; m = {{}};

// Declare these explicitly so that we can extern them in AnalyticalSolution
ASol = 0.0; BSol = 0.0; CSol = 0.0;

function AnalyticalSolution (p);
```

```
   extern Ao, t;
   extern ASol, BSol, CSol;

   k1t = -p.k1*t;
   k2t = -p.k2*t;
   k2k1 = p.k2 - p.k1;

   ASol = Ao*exp(k1t);
   BSol = Ao*p.k1*(p.VA/p.VB)*(exp(k2t)/k2k1)*(exp(k2k1*t) - 1);
   CSol = Ao*(p.VA/p.VC)*(1 - exp(k1t)*(p.k2/k2k1) + exp(k2t)*(p.k1/k2k1));
end;


print "Simulate?";
if ReadKey == "y" then
   begin
   // Calculate first point
   AnalyticalSolution (p); // Compute Analytical Solution
   v = {p.Time, p.A, ASol, p.A-ASol, p.B, BSol, p.B-BSol, p.C, CSol, p.C-CSol};

   AppendRow (m, v);
   // Set up the column labels so we know whats what
   SetColumnName (m, 1, "Time");
   SetColumnName (m, 2, "A");
   SetColumnName (m, 3, "ASol");
   SetColumnName (m, 4, "ErrA");
   SetColumnName (m, 5, "B");
   SetColumnName (m, 6, "BSol");
   SetColumnName (m, 7, "ErrA");
   SetColumnName (m, 8, "C");
   SetColumnName (m, 9, "CSol");
   SetColumnName (m, 10, "ErrA");

   repeat
      t = p.sim.OneStep (t, hstep);
      AnalyticalSolution (p); // Compute Analytical Solution

      v = {p.Time, p.A, ASol, p.A-ASol, p.B, BSol, p.B-BSol, p.C, CSol, p.C-CSol};
      appendrow (m, v);
   until t > 1;
   end;
```

28.8 Forcing Functions

Sometimes it is desirable to be able to perform additional computations each time the model is evaluated by one of Jarnac's simulators. Two examples are worth considering. The simplest use of this facility is to enable large rate laws to be split in to more manageable expressions. So rather than having one long expression after the reaction specification we can have a number of smaller terms, this may useful when trying to make something clear to the human reader. For example:

```
p = defn cell

     // This is a so-called forcing function
     Term1 = S1*k1;
```

```
    Term2 = S2*k2;
    Term3 = Sin ((k1+k2)*Time);
    LongEquation = Term1 + Term2/Term3;

    [J1] S1 -> S2; LongEquation;
end;
```

A more interesting application is where you might have a model where one of the reactions is so fast that it is effectively at equilibrium during the course of the simulation. In such a situation it is a good idea to remove the fast reaction from the integrator's responsibility and compute the equilibrium concentrations separately. Let's say we have a three step linear chain where the middle reaction is very fast compared to two end reactions, we might model the system using the following declaration:

```
p = defn FastReaction
      S1 -> S2; 0.34*S1;
      S2 -> S3; 1000*S2 - 2000*S3;  // Very fast reaction
      S3 -> S4; 0.1*S3;
end;
```

Since the second reaction is very fast compared to the surrounding reactions, the integrator will have to use very small time steps in order to accurately carry out the integration. We can ease the burden on the integrator without compromising the solution to the model is we assume that `S2` and `S3` are in fact at equilibrium throughout the simulation. If we assume this we can pre-compute the equilibrium concentrations of `S2` and `S3` before the rates of reactions are computed as showing in the following script:

```
p = defn FastReaction

     T = S1 + S2;
     S2 = 1000*T/(1000+3000);
     S3 = T - S2;

     S1 -> S2; 0.34*S1;
     S3 -> S4; 0.1*S3;
end;
```

28.9 Loading and Saving Models as SBML

Models my be saved and loaded in standard SBML format.

To save a model as SBML use the method `xml` on the model object. The xml method without an argument returns the SBML as a string. To save the SBML directly to a file, include the name of the file as an argument to the method xml. Thus

```
// Print the SBML to the screen
println p.xml;

// Save the model to a file called mymodel.xml
p.xml ("mymodel.xml");
```

To save a model in SBML Level 2, use the method `xml2`, eg

```
// Print the SBML Level 2 to the screen
println p.xml2;

// Save the model to a file called mymodel.xml
p.xml2 ("mymodel.xml");
```

The built-in method, `network` is a constructor for networks. The following command will create an empty network:

```
p = network;
```

The constructor network, also takes an optional argument. This argument can either be a SBML file name or a SBML string, for example

```
p = network ("mymodel.xml");
```

or

```
f = openfile ("mymodel.xml");
str = f.readfile ();
f.close();

p = network (str);
```

28.10 Model Properties and Methods

Models have a number of methods, properties and objects associated with them. Model objects in turn also have associated methods and properties. We have already encountered one property, the 'clone' property which can be used to make an exact copy of a network model.

The table below summarizes these.

All derivatives are evaluated using the foward/backward difference formula $f' = (1/2h)(f(x+h) - f(x-h))$, unless specified otherwise.

Method Name	Description
Jac	*Evaluates and returns the Reduced Jacobian matrix.*
fJac	*Evaluates and returns the Full Jacobian matrix.*
uElast	*Evaluates and returns the unscaled elasticity matrix*
Elast	*Evaluates and returns the scaled elasticity matrix*
Sm	*Returns the stoichiometry matrix*
Nr	*Returns the reduced stoichiometry matrix*
L	*Returns the Reder link matrix*
L0	*Returns the L0 partition of the L matrix*
Conserve	*Returns the conservation matrix*
em	*Returns a matrix of elementary mode vectors (requires MetaTool)*
rv	*Returns the vector of reaction rates*
dv	*Returns the vector of rates of change*
nReactions	*Returns the number of reactions*
nMetabolites	*Returns the total number of floating metabolites*
fs	*Returns a list containing the names of all floating species*
bs	*Returns a list containing the names of all boundary species*
ps	*Returns a list containing the names of all parameters (excluding boundary species)*
prs (r)	*Returns a list containing the names of parameters for reaction r*
rs	*Returns a list containing the names of all reactions*
vs	*Returns a list containing the names of all compartment*
cc (<CodeType Expression>, Symbol)	*Computes and returns the control coefficient*
ucc (”)	*Computes and returns the unscaled control coefficient*
ee (”)	*Computes and returns the scaled elasticity coefficient*
uee (”)	*Computes and returns the unscaled elasticity coefficient*
xml	*Returns the model as an SBML string*
xml (“filename”)	*Saves the model as SBML to the named file*

Property Name	Description
DiffMethod	*Set differentiation method (3 or 5 step)*

Properties generated from model dependent information.

Properties	Description
CSUMx	*Internally generated moiety total parameters where x is an integer value indicating the xth conservation relationship, eg CSUM1*
cv	*Vector property giving access to the conservation totals, eg p.cv[1] = p.ATP + p.ADP; p.cv = {1.2, 5.6}; println p.cv The order of the elements in cv match the order in the conservation matrix*
dx	*Internally generated rate of change properties, where x should be replaced by the name of a metabolite species, eg p.dATP would return the rate of change for ATP in model p*
sv	*Vector property which gives access to metabolite concentrations, eg p.sv = {1.2, 4.5, 6.6}; p.sv = urndv (size (p.sv)); println p.sv;*

Model Objects

Steady State Object (`ss`):

Methods	Operations
ss.eval	*Evaluate the steady state, eg ss.eval*
Properties	**Operation**
ss.tol *eg ss.tol = 1e-4*	*Set the tolerance for the steady state solver*

Simulation Object (`sim`):

Methods	Operations
sim.eval (Start, Finish, nPoints, Output List of Code Type Expressions);	*Evaluate the time course over a range and return the result in a matrix*
sim.OneStep (StartTime, TimeStep);	*Compute model over give time step and return the next time*
Properties	**Operation**
sim.rtol	*Integration Relative Tolerance*
sim.atol	*Integration Absolute Tolerance*

28.10.1 Generating Time Course Data

It is very easy to generate time course data for a simulation in Jarnac. All one needs to do is call the simulation eval function of the model. Each model that is declared in Jarnac acquires a host of functions, one of which is the `sim.eval` function. This function takes four arguments, the start time for the simulation, the end time for the simulation, the number of data points to generate during the simulation and finally a list containing the quantities that should be monitored during the simulation. The function returns a matrix data type, where each row corresponds to a data point and the columns correspond to the elements specified in the monitoring list. For example:

```
// Perform a simulation from time zero to time 100
// Generate 50 points (that is every two time points)
// Monitor the following data, Time, S1, S2, J1, J1/J2

m = p.sim.eval (0, 100, 50, [<p.Time>, <p.S1>, <p.S2>, <p.J1>, <p.J1/p.J2>]);
```

Note that the monitored quantities must be code types, that is the expressions must be enclosed in angle brackets otherwise a compiler error will be generated.

The example shown above ill generate a matrix, `m`, which has fifty rows and five columns. The first column will contain the time during the simulation – `P.Time` is an automatically generated variable which gives access to the independent time variable. The remaining four columns correspond to the indicated expressions, namely, `p.S1` and so on. The first three arguments in the example are shown as constants but could easily be substituted with any legal Jarnac expression.

OneStep An alternative approach to generating time course data is to use the sim.OneStep function. The OneStep function moves the simulation on one time step. Its syntax is very simple:

```
sim.OneStep (StartTime, TimeStep)
```

The first argument is the start time and the second argument the time step over which to carry out the simulation. Upon completion, OneStep will return the new simulation time, that is, StartTime + TimeStep. This makes is very easy to write OneStep as part of a simulation loop, for example:

```
TimeStart = 0;
TimeEnd = 100;
NumberOfDataPoints = 50;
t = TimeStart;
hstep = TimeEnd/NumberOfDataPoints;

repeat
  t = p.sim.OneStep (t, hstep);
  ...save, print do other calculations etc at current point in simulation
until t > TimeEnd;
```

The advantage of `sim.OneStep` over `sim.Eval` is that `sim.OneStep` allows much more flexibility to control the simulation. For example, between time points it is possible to change parameters etc and so alter the course of the simulation. Of course this extra flexibility requires more effort by the user to implement.

28.10.2 Metabolic Control Analysis

The modelling object has special built-in support for computing the various steady-state coefficients of metabolic control analysis. These include the control coefficients and the elasticity coefficients. Control coefficients are computed using the `cc()` function and elasticities, using the `ee()` function. Each of these functions takes two arguments, the first argument is either a reference to a single dependent variable or it could be some function of one or more dependent variables. In all cases, the first argument must be a code type, that it, the expression must be enclosed in angle brackets otherwise a compiler error will result. For example

```
C1 = p.cc (<p.J1+p.J2>, p.Vm1);
E1 = p.ee (<p.FeedFlux>, p.S1);
C2 = p.cc (<p.J1>, p.CSUM1);
C3 = p.cc (<p.J1>, p.X0);
```

The second argument is always the independent variable, in other words a parameter of the model. Legal parameters include quantities such as enzyme kinetic parameters, boundary conditions, or conservation totals. When an elasticity is being computed, the independent variable can include the floating species names.

```
e1 = p.ee (<p.J1>, p.Km);
e2 = p.ee (<p.J2>, p.S1);
```

28.11 Runtime manipulation of network models

In the current release there is experimental support for manipulating network models at runtime. The following operations are available:

`p.addfloat (name)`	Add a new floating metabolite to the network
`p.addfixed (name)`	Add a new fixed metabolite to the network
`p.deletemetabolite (name)`	Delete fixed or floating metabolite
`p.deletereaction (name)`	Delete the specified reaction from the network
`p.connect ()`	Add a new reaction to the network, see below for details

Example:

```
p = network  // Create an empty network
p.addfloat ("S1");
p.addfloat ("S2");
p.connect ("J1", ["S1"], ["S2"], "k1*S1");

p.S1 = 1.2; p.S2 = 0.1; p.k1 = 0.4;
m = p.sim.eval (0, 10, 0.5, [p.Time, p.S1]);
graph (m);
```

Connect takes four arguments, the name of the reaction to add, two lists, and a string representing the kinetic rate laws. The first list is the list of reactants and the second the list of products. Elements of the lists must contain the names of the species. If a species names starts with an integer value then this is assumed to be the species' stoichiometry, eg

```
S1 -> S2       :  ["S1"], ["S2"]
S1 -> S2 + S3  :  ["S1"], ["S2", "S3"]
2 S1 -> 3 S2 + S3 : ["2S1"], ["3 S2", "S3"]
```

29 SBW Compliant Features

THIS SECTION SUBJECT TO CHANGE

On start up, Jarnac will attempt to make a connection with a local SBW broker. Once connected, Jarnac has access to modules connected to the SBW environment. In addition, Jarnac exposed and set of SBW services which allows Jarnac to be used as a simulation server.

The interface to SBW is still under development and is somewhat experimental, however the interface is functional.

If Jarnac makes a successful connection to SBW, a new object, `sbw` is made available. This variable summarises the current SBW environment, at the present time, the sbw object is not updated as modules startup and shutdown in the SBW environment, this will be changed in a future release.

The `sbw` object has a single property called `modules`. This function returns a list of the currently operating modules.

```
->sbw.modules
[BROKER, Trig, Graph, Jarnac]
->
```

The names in the module list are functions which can be used to obtain details of the services and methods available for particular modules. Thus the statement:

```
->sbw.modules[2]()
['trig', 'logs']
->
```

will return a list of available services. The entries in this list are also functions which can in turn be used to obtain the list of methods available in a particular method, for example:

```
->sbw.modules[2]()[1]()
['sin', 'cos', 'tan']
->
```

The list of methods represent the method functions themselves. Thus to execute a particular method, for example the sin method, use the statement:

```
->sbw.modules[2]()[1]()[1](3.4)
0.059306373
->
```

30 Detailed Description of Functions - Incomplete

30.1 System Calls

Size (object) Size returns the number of items in the indicated object. Types such as integers, booleans and floats all return a value of one. If the object is a string type then size returns the number of characters in the string. In the case of vectors, matrices or lists, size returns the number of elements in the object. Size returns the number of systems equations if the object is either a metabolic model or a ode object.

```
size (4.5)
length = size ("hello")
v = {1,2,3}
println size (v)
```

SizeOf (arg) SizeOf returns the number of bytes occupied by the argument.

```
sizeof (4.5)
sizeof ("hello")
v = {1,2,3}
println sizeof (v)
```

Mem This function returns the number of bytes currently allocated on the system heap (dynamic memory).

timer This function returns the current system timer in milliseconds. Resolution is approximately to 1ms. timer is useful for timing operations.

```
Start = timer
...
do stuff
...
TimeTaken = timer - Start;
```

Status This function returns the names of all currently loaded modules and the status of the machine stack.

Symbols (arg) This function returns a list of currently active symbols. The supplied argument, which must be a string type, determines which symbol(s) are listed. The empty argument, "", returns a list of all the top level symbols. Subsequence symbol levels are accessed using the indirection operator '.' For example:

```
symbols ("")
[BuiltInConst, BuiltInVar, BuiltInFuncs, main, a]
symbols ("main.a")
[a <Integer>]
symbols ("builtinfuncs")
[cos, sin, tan, ... etc ]
```

del (arg) This function deletes the symbol specified in the argument, which must be a string type. Symbols levels are accessed using the indirection operator '.' For example:

```
a = 4
del ("main.a")
```

delall This function deletes all user introduced symbols. Essentially is performs a clear operation on the workspace.

```
a = 4
delall
```

30.2 Input/Output and Graphing

Graph (m <,[...]>) Graph is used to plot data held in a matrix, m. The data in the matrix argument, m, should be arranged in columns, where each column represents a variable. By default the first column is the independent variable and the remaining columns the dependent variables. Each dependent column will generate it's own curve on the graph. The choice of dependent and independent variable can be changed through the optional second argument. This argument must be a list of integers, where the integers represent the column indices. The first column index in the list will be considered the independent variable and the remaining indices the dependent variables.

If m is a 100 by 4 matrix then

```
graph (m);
graph (m, [1,2]);  // Plot column two against column one
graph (m, [4,1]);  // Change independent variable to column four
graph (m, [1,2,3,4]); // Same as graph (m)
```

30.3 Miscellaneous

AddHelp Str This operation is used to add help strings to user defined functions. For example

```
function MyFunc
  AddHelp "This will compute the....";
  ...do stuff ...
  return x;
end
```

Once MyFunc is compiled the help string can be accessed using the quick help feature which is question mark symbol followed by the name of the symbols, for example

```
?MyFunc
This will compute the....
```

Quick help applies to built-in as well as user defined functions.

IsType (arg) Jarnac uses runtime-typing to determine what operation to perform on data, this allows dynamic typing to occur, where an operation, such as `a + b` can represent the addition of integers, floats, strings, vectors and so on, the actual operation is selected at runtime and not at compile time. Sometimes it is desirable to know the type of a particular piece of data at runtime. For example a user function which can only work with vectors should make sure that the argument passed to the function is a vector. This can be accomplished with the following code:

```
function AddVectors (v1, v2)
  if (IsType (v1) = tVector) and (IsType (v2) = tVector) then
     begin
     do stuff....
     end
  else
     println "Error, arguments to AddVectors must be vector types";
end
```

`IsType` returns an integer which indicates the type. There are builtin type constants which can be used to determine the meaning of the value returned by `IsType`. These constants can be listed with the command: `symbols ("bsymbols")`, they include, `tInteger, tDouble, tBoolean, tString, tVector, tMatrix, tList, tNetwork, tOde`.

31 Calling Jarnac from Python

Jarnac implements a standard BSD socket interface. This enables other applications to communicate with Jarnac and access much of the functionality within Jarnac. Thus it is possible for external appliciations, either hosted locally or remotely, to carry out simulations, define models, etc.

A Jarnac socket acts as server listening to requests from remote client sockets. The socket may be turned off or on as necessary. The port number is set by default to 7777 but may be changed from the preferences dialog box.

In the current implementation the Jarnac socket will accept any valid Jarnac script. Any output to the Jarnac console (eg via the print statement) is directed at the client socket so that the client can capture the results of simulations etc. Using this technique it is very easy to run simulations remotely from say a Linux platform.

The remainder of this section illustrates how one can use Python as the client and how one can run a simple simulation remotely.

In order to make the socket connection slightly secure it is first necessary to 'login' to the Jarnac server, if you don't do this all requests to the server will be rejected. The syntax for logging in is very simple:

`login <password>`

that is the string 'login' followed by one or more space characters followed by the Jarnac login password. The password can be set in the preferences dialog box from within Jarnac but by default is set to the string 'Jarnac$'. If the login is successful, Jarnac will return the string `ready` to the client. One can also logout of a Jarnac session by issuing the string 'logout' to the server.

Note that the host url in the following Python example is a fictional one, the actual url will depend on the location of the running Jarnac server.

Python Script to remotely control Jarnac and perform a simple calculation:

```
from socket import *
HOST = 'cds.remote.com/server'  # A url for the remote host
PORT = 7777                     # The same port as used by the server
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
# Set no blocking
s.setblocking (0)
s.send ('login Jarnac$')
# Assume the login was successful and discard the ready message
s.recv (128)
s.send('print sin (30) + cos(20)')
data = s.recv(1024)
s.close()
print data
```

or to run a simulation one could use:

```
from socket import *
HOST = 'cds.remote.com/server'  # A url for the remote Jarnac host
PORT = 7777                     # The same port as used by the server
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
s.setblocking (0)
s.send ('login Jarnac$')
# Assume the login was successful and discard the ready message
s.recv (128)
s.send('p = defn cell J1: $X0 -> S1; k1*X0; J2: S1 -> $X1; k2*S1; end')
s.send('p.X0 = 1.2; p.S1 = 0.1; p.X1 = 0')
s.send('p.k1 = 4.5; p.k2 = 2.3')
s.send('p.ss.eval')
s.send('print "S1=", p.S1, " Flux=", p.J1')
data = s.recv(1024)
s.close()
print data
```

If you wish to make a connection to Jarnac locally, for example you're running Python and Jarnac on the same machine then the local host address which you should use to make a connection is given by the Python function `gethostname()`. Thus the `connect` method call becomes, `s.connect (gethostname(), PORT)` if client and server and located on the same machine.

32 Examples

This section gives some example Jarnac programs. The first example shows how one can specify a model, initialise its state and compute the steady state flux and metabolite concentration.

```
// Very Simple Steady state calculation

  DefaultModel model
    [J1] $X0 -> S1; Vmax1/Km1*(X0 - S1/Keq1)/(1 + X0/Km1 + S1/Km2);
    [J2] S1 -> $X1; Vmax2/Km3*(S1 - X1/Keq2)/(1 + S1/Km3 + X1/Km4);
  end;

  Vmax1 = 1; Vmax2 = 6.5;
  Km1 = 3.4; Km2 = 2.3;
  Km3 = 6.7; Km4 = 1.2;
  Keq1 = 7;  Keq2 = 9;

  X0 = 1; X1 = 0; S1 = 0.1;

  ss.eval;  // Compute the steady state

  println "Steady state concentration and flux";
  println;
  println "S1=", S1, ", J1=", J1, nl;
```

The second example is a much more substantial program and illustrates simple looping, evaluating some matrices of interest and computing a control coefficient over a range of steady states.

```
// Simple Steady state calculation with repeat
// loop and control coefficient calculation


  // Network Specification
  DefaultModel model
    [J1] $X0 -> S1; Vmax1/Km1*(X0 - S1/Keq1)/(1 + X0/Km1 + S1/Km2);
    [J2] S1 -> $X1; Vmax2/Km3*(S1 - X1/Keq2)/(1 + S1/Km3 + X1/Km4);
  end;


  // Initialise constants and variables
  Vmax1 = 3; Vmax2 = 1.5;
  Km1 = 3.4; Km2 = 2.3;
  Km3 = 6.7; Km4 = 8.2;
  Keq1 = 15; Keq2 = 1;

  X0 = 1; X1 = 0; S1 = 0.1;

  // Format () is a general purpose formating function
  println "Stoichiometry matrix=", nl, Format ("%g ", [Sm]);
  println "Reder's L matrix=", nl, Format ("%g", [L]);

  if rows (L0) > 0 then
     println "Reder's L0 matrix=", nl, Format ("%g", [L0])
  else println "No L0 matrix available", nl;
```

```
println "Reder's Nr matrix=", nl, Format ("%g ", [Nr]);

ss.eval;  // Compute the steady state

println "Steady state concentration and flux", nl;
println "S1=", S1, " J1=", J1, nl;

// Compute the steady state and one control coefficient over
// a range of boundary condition X0

println " X0     S1       J1       C(J1, E1)";
println "-----------------------------------";
repeat
  ss.eval;
  println FormatStr (" %2g   %8.5g   %7.5g   %7.5g",
             [X0, S1, J1, CC(<J1>, Vmax1)]);
  X0 = X0 + 3.0;
until X0 > 30;
```

The final example illustrates a time course simulation using Heinrich's oscillating model from their '77 review. If you run this from the Jarnac Windows front end you will be able to view graphical output generated by this program.

```
// 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 Pathway
     [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 = 8.0;

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

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

  // 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;
```