Summary of Some Common Jarnac Language Constructs

Sample model definition:

```
p = defn cell
     var S1,S2;    // floating species
     ext X0,X1;    // boundary species (remain constant)

     // 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;
```

// Initialize all the parameters 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>]);
```

// The above statement returns a matrix which we can graph

```
graph(m);
```

Multiple Plots

To overlay an existing graph with another use the command:

```
setghold (true);
```

For example, carry out one simulation, change a parameter and resimulate, displaying both simulations on the same graph:

```
// Carry out first simulation
m = p.sim.eval(TimeStart, TimeEnd, NumberOfDataPoints,
       [<p.Time>, <p.S1>, <p.S2>, <p.J1>]);
// Plot the results
graph (m);
// Hold the current graph
setghold (true);
// Change a parameter value
p.k1 = 35.5;
// Repeat the simulation
m = p.sim.eval(TimeStart, TimeEnd, NumberOfDataPoints,
       [<p.Time>, <p.S1>, <p.S2>, <p.J1>]);
// Plot the new results
graph (m);
// Turn off the graph hold
setghold (false);
```

Stochastic Simulations

Jarnac has a wide variety of commands for carrying out stochastic simulations. These include:

```
gillLoadSBML (sbmlString);
```

This loads an SBML model into the Gillespie simulator, eg.

```
gillLoadSBML (p.xml);
```

where p is a Jarnac model variable.

Using the loaded model, carry out a Gillespie simulation from time start to time end. The samplePeriod is indicates how often to take a measurement. A samplePeriod = 1 means take every data point that the Gillespie algorithm generates, whereas samplePeriod = 10 means take every 10^{th} point. Note that time intervals in a Gillespie simulation are not regular, to get regular time points use the grid based calls.

```
gillSimulate (startTime, endTime, samplePeriod);
```

Using the loaded model, carry out a Gillespie simulation from time start time end. The gridSize indicates when to record data from the simulation. A gridSize = 1 means record data at intervals of time one. The units of time will depend on the units given to the rate constants in the model.

```
gillSimulateOnGrid (startTime, endTime, gridSize);
```

The following call is the same as the previous one except it will do multiple simulations indicated by the populationSize argument. From the population of runs, the method will return columns corresponding to the mean changes. For example, the following call will run 10,000 simulations on a grid of width 0.1 and returns the mean change in species levels.:

```
gillSimulateMeanOnGrid (0.0, 10.0 0.1, 10000);
```

```
gillSimulateMeanOnGrid (startTime, endTime, gridSize,
populationSize);
```

The following call is the same as the previous one except it also returns the standard deviation as well as the mean.

```
gillSimulateMeanAndSDOnGrid (startTime, endTime, gridSize,
populationSize);
```

There are times when simulations runs need to be repeated exactly, this can be accomplished by setting the random number generator seed. This ensures that runs with a given random number seed will be identical.

```
gillSetSeed (value);
```

This call enables a parameter in the model to be changed without having to reload the SBML model into the Gillespie solver.

For example: `gillSetParameter ("k1", 0.1234);`

```
gillSetParameter (parameterName, value);
```

The following call will return parameters names in the model:

```
gillGetNamesOfParameters();
```

The following two calls return the names of the floating and boundary species respectively.

```
gillGetNamesOfFloatingSpecies();
gillGetNamesOfBoundarySpecies();
```

Example:

```
// This will run multiple Gillespie simulations
// at different parameter values and plot the
// resulting runs on one graph.
P = defn myModel
      $Xo -> S1;  k1*Xo;
       S1 -> $X1; k2*S1;
end;

p.Xo = 10;
p.S1 = 0;
p.X1 = 0;
p.k1 = 0.23; p.k2 = 0.56;

gillLoadSBML (p.xml);
```

```
m = gillSimulate (0, 50, 1);
graph (m);
setghold (true);
k1 = 0.1;
for i = 1 to 5 do
    begin
    gillSetParameter ("k1", k1);
    m = gillSimulate (0, 50, 1);
    graph (m);
    k1 = k1 + 2;
    end;
setghold (false);
```

Other Useful Methods:

```
m = timeSlice (matrix, lowerTime, upperTime;
```

This call will take a matrix and returns all rows between lower and upper time bounds. It is assumed that the first column of the matrix contains the time variable with ascending values.

```
v = getColumn (matrix, index);
```

This function returns a single column from a matrix m at column index and returns the column as a vector.

```
m = getColumns (matrix, [3,4,1,2]);
```

This function extracts a set of columns from a matrix argument and the returns the columns in the form of a new matrix. The columns are selected from the list argument where elements of the list indicate column indices.

```
m = pdf (v);
```

This function takes a vector as an argument and returns a matrix corresponding to the probability density function.

```
pdfPlot (v);
```

Same as the function pdf() but plots the data as a histogram.

```
exportCSV (matrix)
```

or

```
exportCSV (matrix, separator)
```

Exports the matrix, m as CSV data. Depending on your computer setup, an appropriate application will be launched (eg. Excel, OpenOffice) to load the data. This is useful for quickly transfers a matrix of data to another application such as Excel or OpenOffice.

The second version of exportCSV() allows one to specify the separator between data values. The default is to separate data values using ‘,’ but this may not always be appropriate.

For-Loop :

```
for i=1 to 100 do
    begin
      <code>
    end;
```

If-statement:

```
if a == 4 then
   // begin-end only needed for multiple statements
   begin
     <code>
   end
else
   begin
     <code>
   end;
```

printing:

```
println "statement”, x, y, z;
```

Matrix operations:

```
m1 = matrix (10,5);  // Construct a 10 x 5 matrix
v1 = matrix (10,1);  // Construct a column vector
v2 = matrix (1,6);   //  a row vector

m2 = aug (m1,v1);    // Augment the columns of m1 and
                        v2 together
m3 = augr (m1,v2);   // Augment the columns of m1 and
                        v2 together
```