Python Guide for the Systems Biology Workbench

Herbert Sauro, Mike Hucka, Andrew Finney, Hamid Bolouri

{hsauro,mhucka,afinney,hbolouri}@cds.caltech.edu
Systems Biology Workbench Development Group
ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA 91125, USA

http://www.cds.caltech.edu/erato

Principal Investigators: John Doyle and Hiroaki Kitano

April 12, 2002

Contents

1 Introduction

This document describes the Python interface to the Systems Biology Workbench (Hucka et al., 2001b).

The Python interface has been designed so that it is very simple to call remote methods in other SBW aware modules. The intention in developing the Python module was that if should enable users to easily control SBW applications using Python scripts.

In addition to providing simple access to SBW modules, the Python interface also exposes many of the functions in the standard SBW API. The reader is recommended to consult the SBW API documentation for more details. In the majority of cases, users will not find it necessary to access the SBW API directly.

When a SBW module starts, Python is notified and a Python interface is automatically constructed to represent the module, this makes it very easy to access a module's functionality. When the module closes, Python will automatically remove the module's interface.

As an example consider a module which has a number of mathematical functions, including sin, cos and tan. Also assume that these functions are under the service heading 'trig', with the module name being 'Trig'.

When the Trig module starts up, the following appears on the Python console.

```
>>> ****** Trig is started up... creating proxy....done
>>>
>>> Trig

<__main__.TrigClass instance at 0087D574>
>>>
```

When Python creates a proxy object to the module, it uses the name of the modules as the basis for the variable name which will be created to represent the module. For example, the variable 'Trig' is automatically created to represent the 'Trig' module. This variable has one variable field called 'services'. This field holds the names of the services currently available. Thus the following scripts yields the output:

```
>>> Trig.services
['trig', 'log']
>>>
```

In other words, Trig contains two services, called 'trig' and 'log'. To find out the methods associated with a service use the 'methods' field variable on a particular service. For example:

```
>>> Trig.trig.methods
['sin', 'cos', 'tan']
>>>
```

Invoking a particular method is very straight forward, to invoke the sin method we for example simply use the statement:

```
>>> Trig.trig.sin (30.0)
-0.98803162409286183
>>>
```

2 Getting Help

Since Python has been designed to be used interactively, a user will often need to obtain help n specific methods in SBW modules.

To get help on a particular method one can use the SBW Python call `getHelp ()`. This call takes three arguments, the module name, the service name and the method name. For example to get help on the sin method use the following code:

```
>>> psbw.getHelp ('Trig', 'trig', 'sin')
'Returns the sine of a radian angle'
>>>
```

Note that the symbol 'psbw' is the object instance of SBW.

3 SBW Lists

SBW Lists map directly on to Python Lists. Thus if a method call requires a list as an argument one can use the following code:

```
>>> x = module.service.method ([1, 2, 'hello', 4.5677])
>>>
```

See the example application at the end of this document for an example of how Lists are employed.

4 SBW Arrays

Python arrays are defined as fixed record data blocks, that is each element in a Python array has a fixed size. However, SBW arrays permit elements to have variable sized records. In order to make Python and SBW arrays compliant, the SBW interface will only accept SBW arrays which contain integers or doubles since these are a special case when the elements of and SWB Array are fixed. However, since Python arrays are primarily used for data analysis, this limitation is not too restrictive. Python uses Lists to handle non-fixed record arrays.

To enable numeric arrays to be used to their fullest, the SBW Python implementation uses the Python Numeric extension as the means for handling arrays.

See the example application at the end of this document for an example of how arrays are employed.

5 Example

This example will employ the ODE evaluation module, called Pasadena. Pasadena allows a user to specify a set of ODEs which might represent a biological model. Users may choose to either solve the ODEs using Pasadena's built-in GUI interface or they may wish to have greater control through Pasadena's SBW interface by controlling Pasadena via Python scripts. This section describes how one might go about this.

The Pasadena module exports the following methods:

int getNumVariables ()

Returns the number of ODE variables

getVariableNames ()

Returns a list containing the names of the variables

double[] getVariables ()

Returns a double array containing the values of the variables. Note that there is a one to one correspondence between the name in the variable list and a value in the array returned by this method.

void setVariables (double[])

Sets the variables of the model to the double array.

int getNumParameters ()

Returns the number of parameters in the model

getParameterNames ()

Returns a list containing names of parameters in the model

double[] getParameters ()

Returns a double array containing the values of the parameters. Note that there is a one to one correspondence between the name in the parameter name list and a value in the parameter array returned by this method.

void setParameters (double[])

Sets the parameters of the model to the double array.

double[] getODEArray ()

Returns a double array containing the rates of change of the variables at the current variable and parameter values in the model.

double[][] getJacobian()

Returns a double matrix containing the current Jacobian

double[] getODEArrayArg (double[] s, double[] p)

Returns a double array containing the values of the ODEs given the two arguments in the call. The first argument is a double array containing the variable values, the second array is a double array containing the parameter values.

double[][] getJacobianArg(double[] s, double[] p)

Returns a matrix containing the Jacobian of the system given the two arguments in the call. The first argument is a double array containing the variable values, the second array is a double array containing the parameter values.

Note that the notation {} indicates a list, and [] indicates an array. Note that Python employs the reverse notation.

The following script illustrates a typical session between Python and the Pasadena Module:

```
>>> p = PasadenaTwain.pt
>>> p.getNumVariables()
2
>>> p.getVariableNames()
['S1', 'S2']
>>> p.getVariables()
array([0.1, 0.03], 'd')
>>> p.getODEArray()
array([0.10000000000000001, 0.029999999999999999], 'd')
>>> s = p.getVariables()
>>> s[0] = 0.6
>>> p.setVariables (s)
>>> p.getODEArray()
array([-0.049999999999999989, 0.17999999999999999], 'd')
>>> p.getJacobian()
array([[-0.5, 0.0], [0.5, -0.70000000000000551]], 'd')
>>>
```

6 Caveats

There are two issues that users should be aware of when using Python under SBW.

1. The current implementation does not recognise multiple instances of the same module, in fact if a second module of the same name is started, then the Python interface to first module is overwritten. A future version will correct this deficiency.

2. If an SBW application returns an array containing, for example strings, as a result of a module method call, Python will not recognize the data. In a future version, this problem will be partly solved by returning such arrays as Python Lists. In addition, if an argument to a method call requires an array of strings the current version will not be able to call this method.

Hucka, M., Finney, A., Sauro, H. M., and Bolouri, H. (2001b). Systems Biology Markup Language (SBML) Level 1: Structures and facilities for basic model definitions. Available via the World Wide Web at http://www.cds.caltech.edu/erato.