Introduction to PySCeS 0.4.x

http://pysces.sourceforge.net

Brett G. Olivier, Johann M. Rohwer and Jan-Hendrik S. Hofmeyr

Triple-J Group for Molecular Cell Physiology
Biochemistry Department, Stellenbosch University

Contents

1 Introducing PySCeS

> PySCeS: the Python Simulator for Cellular Systems is an extendable toolkit for the simulation, analysis and investigation of cellular systems. It is available for download from: http://pysces.sourceforge.net/

Welcome! This quick reference will guide you through the basics of modelling cellular systems with PySCeS. It is meant to be used together with the *input file guide*. If you already have PySCeS installed continue directly to Section 2, if not Section 7 contains instructions on building and installing PySCeS. This section also contains detailed licensing information.

> PySCeS is distributed under the GNU General Public Licence (GPL) and is made freely available as Open Source software.

We hope that you will enjoy using our software. If, however, you find any unexpected features (i.e. bugs) or have any suggestions on how we can improve PySCeS please let us know.

Brett G. Olivier
Johann M. Rohwer
Jan-Hendrik S. Hofmeyr
Stellenbosch (September 21, 2007)
http://www.jjj.sun.ac.za

2 Running PySCeS

In this section we assume you have PySCeS installed (see Section 7 for details) and a correctly formatted PySCeS input file that describes a cellular system in terms of its reactions, species and parameters. For a detailed description of the PySCeS model file format see the *PySCeS Input File Guide*. Note that on all platforms PySCeS model files have the extension `.psc`.

2.1 Starting a PySCeS session

To begin modelling we need to start up an interactive Python shell (we suggest iPython[1] and import PySCeS:

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08)
Type "copyright","credits" or "license" for more information.

>>> import pysces

PySCeS environment
******************
pysces.model_dir = C:\mypysces\pscmodels
pysces.output_dir = C:\mypysces

***********************************************************************
* Welcome to PySCeS (0.4.0) - Python Simulator for Cellular Systems   *
*                http://pysces.sourceforge.net                        *
* Copyright(C) B.G. Olivier, J.M. Rohwer, J.-H.S. Hofmeyr, 2004-2007  *
* Triple-J Group for Molecular Cell Physiology                        *
* Stellenbosch University, South Africa                               *
* PySCeS is distributed under the GNU General Public Licence          *
* See README.txt for licence details                                  *
***********************************************************************

>>>
```

PySCeS is now ready to use. The first time you run PySCeS it will create the file `pys_usercfg.ini` which is used to store local user configuration data. On POSIX type systems this file will be created in `<HOME>/pysces`. Under Windows this is created in the `<PYTHON>\lib\site-packages\pysces` directory. Currently, this configuration file allows you to customize the two PySCeS working directories:

[1] http://ipython.scipy.org

- `model_dir` - where PySCeS looks for your input (.psc) files.
- `output_dir` - where any temporary files or other output is generated.

If you would like to change the these defaults exit PySCeS and the interactive shell, edit the configuration files, restart an interactive shell and import PySCeS. Your new paths should be displayed on startup.

2.2 Instantiating a model object

> This guide uses the test models, supplied with PySCeS, as examples. To copy them to the redefined input file directory use the `pysces.CopyModels()` command.

Before modelling, a PySCeS model object needs to be instantiated. As a convention we use `mod` as the instantiated model instance. The following code creates such an instance using the test input file, `pysces_test_linear1.psc`:

```
>>> mod = pysces.model('pysces_test_linear1')
Assuming extension is .psc
Using model directory: /home/bgoli/pscmodels
/home/bgoli/pysces/linear1.psc loading ..... Done.
```

When instantiating a new model object, PySCeS input files are assumed to have a `.psc` extension (if not supplied). If the specified input file does not exist in the input file directory (e.g. misspelled filename) a list of existing input files is shown and the user is given an opportunity to enter the correct name.

2.3 Loading a model

Once a new model object has been initialised is needs to be loaded. During the load process the input file is parsed, the model desciption is translated into Python data structures and a structural analysis is performed. To get a usable model object use the doLoad() command:

```
>>> mod.doLoad()

Parsing file: C:\mypysces\pscmodels\pysces_test_linear1.psc
Calculating L matrix . . . . . . .  done.
Calculating K matrix . . . . . . .  done.
```

Once loaded, all the model elements contained in the input file are made available as a model attribute[2]. For example, in the input file you might find initialisations such as `s1 = 1.0 and k1 = 10.0`. For example `s1` can be accessed as the species variable, `mod.s1` as well as an attribute which holds its initial value, `mod.s1_init`.

```
>>> mod.s1
1.0
>>> mod.s1_init
1.0
>>> mod.k1
10.0
```

Any errors generated during the loading process typically occur as a result of problems (e.g. syntax errors) in the input file. A typical example of this is the 'list out of range' exception which usually indicates a missing multiplication operator (`3(` instead of `3*()` or unbalanced parentheses in a rate equation expression.

2.4 Basic model attributes

Some basic model properties are accessible once the model is loaded.

- `mod.ModelFile`, the name of the loaded model input file.
- `mod.ModelDir`, the input file directory .
- `mod.ModelOutput`, PySCeS work/output directory
- The model parameters are made available using their original name as specified in the input file e.g. the value of parameter `k1` is now accessible as `mod.k1`
- External (fixed) species are made available in the same way.
- Internal (variable) species are treated in a similar way except that an additional attribute (parameter) is created to hold the species initial value (as specified in the input file) e.g. from `s1`, `mod.s1` and `mod.s1_init` are instantiated as model object attributes.
- Rate equations are translated into class methods that can be evaluated to give their current value. For example, calling `mod.R1()` returns the `R1`'s current rate.
- `mod.species` contains the model's variable species names (and order relative to the stoichiometric matrix).
- `mod.fixed_species` contains the fixed species names.

[2]to display the models attributes see the showX methods described later

- Reaction names and order can be displayed using `mod.reactions`
- Parameter names (including fixed species) can be displayed using `mod.parameters`

All basic model attributes that are described in the input file can be changed interactively. However if the model rate equations need to be changed, this can only be done in the input file after which the model should be re-instantiated and reloaded.

3 PySCeS: core analysis functions

3.1 Structural Analysis

As part of the model loading procedure, `doLoad()` automatically perform a stoichiometric (structural) analysis of the model. The structural properties[3] of the model are captured in stoichiometric matrix (**N**), kernel matrix (**K**) and link matrix (**L**). The structural matrices can either be displayed with a `mod.showX()` method or used in further calculations as a numeric array.

Structural Analysis - legacy

- `mod.nmatrix`, **N**: displayed with `mod.showN()`
- `mod.kmatrix`, **K**: displayed with `mod.showK()`
- `mod.lmatrix`, **L**: displayed with `mod.showL()` (an identity matrix means **L** does not exist i.e. no linear dependence).
- If there are linear dependencies in the differential equations then the reduced stoichiometric matrix of linearly independent, differential equations **Nr** is available as `mod.nrmatrix` and is displayed with `mod.showNr()`. If there is no dependence $\mathbf{Nr} = \mathbf{N}$.
- In the case where there is linear dependence the moiety conservation sums can be displayed by using `mod.showConserved()`. The conservation totals are calculated from the initial values of the variable species as defined in the model file.
- When the **K** and **L** matrices exist, their dependent parts ($\mathbf{K}_0$, $\mathbf{L}_0$) are available as `mod.kzeromatrix` and `mod.lzeromatrix`
- `mod.showConserved()` prints any moiety conserved relationships (if present).

[3]The formal definition of these matrices, as they are used in PySCeS, is described in:
J.-H.S. Hofmeyr (2001) *Metabolic control analysis in a nutshell*, in T.-M. Yi, M. Hucka, M. Morohashi, and H. Kitano, eds, 'Proceedings of the 2^{nd} International Conference on Systems Biology', pp. 291–300

- `mod.showFluxRelationships()` shows the relationships between dependent and independent fluxes at steady state.

If the `mod.showX()` methods are used the various matrices' row and column titles are displayed with the matrix. Additionally, all of the `mod.showX()` methods accept an open file object as an argument. If this file argument is present the method's results are output to a file and not printed to the screen. Alternatively, the order of each matrix dimension, relative to the stoichiometric matrix, is available as either a row or column array (e.g. `mod.krow, mod.lrow, mod.kzerocol`).

3.2 Structural Analysis - new objects

PySCeS now includes object oriented access to a models structural properties using the following attributes:

- mod.Nmatrix, mod.Nrmatrix
- mod.Lmatrix, mod.L0matrix
- mod.Kmatrix, mod.K0matrix

All new structural objects have an `array` attribute which holds the reference matrix as well as ridx and cidx which hold the row and column indexes (relative to the stoichiometric matrix) as well as the following methods:

- `getLabels()` return the matrix labels as tuple([rows], [columns])
- `getColsByName()` extract column(s) with label
- `getRowsByName()` extract row(s) with label
- `getIndexes()` return the matrix indices (relative to the Stoichiometric matrix) as tuple((rows), (columns))
- `getColsByIdx()` extract column(s) referenced by index
- `getRowsByIdx()` extract row(s) referenced by index

In future releases this will be the preferred way of accessing a model's structural properties.

3.3 Time simulation

PySCeS can perform time simulations by using the ODEPACK integration routine LSODA.

There are three ways of running a simulation. (i) by defining the simulation's end time and the number of required points and using the `Simulate()` method:

```
>>> mod.Simulate()
Integration successful.
```

(ii) the end time and number of points can be specified using the `doSim()` method. In this example a 20 step simulation is run from time point zero up to 10:

```
>>> mod.doSim(end=10.0, points=20.0)
Integration successful.
```

(iii) the `doSimPlot()` command runs the time simulation and displays the results. It takes the same arguments as `doSim()` as well as a `plot` string argument (this argument is described as part of the `SimPlot()` command):

```
>>> mod.doSimPlot(end=10.0, points=20.0, plot='met')
Integration successful.
```

Although easy to use, a more flexible way to visualize the results of a time simulation is to use the `mod.SimPlot()` method:

```
mod.SimPlot(plot='met',filename='',title='title',logx='',logy='')
```

Called without arguments, `mod.SimPlot()` plots all the species concentrations against time. However, it may be customized in a number of ways:

- `plot=` can be: `'met'`, all species or `'rate'`, all reaction rates or a user supplied list of model species and rates, e.g. `['s1','s2','R3','R5']`.
- If a filename argument is provided, `mod.SimPlot()` will, in addition to displaying the result on the screen, try to write a PNG image of the plot to a PNG file named *filename.png*
- By default `mod.SimPlot()` generates a title for the plot using the model file name plus the time that the plot is generated. Custom titles can be set with `title = 'mytitle'` argument.
- If `logx` and `logx` are given a string value (e.g. `logx = 'on'`) the respective axis is displayed using a logarithmic scale.

Simulation results

In PySCeS 0.4.x the simulation results have been consolidated into a new `mod.data_sim` object. By default in any simulation only the species concentrations are calculated, however, the reaction rates can easily be generated as well by setting:
`mod.data_sim_add_rates = True` *before* a simulation is executed.

Once a simulation is executed the results are stored in the `mod.data_sim` object which has the following attributes and methods:

- `data_sim.getTime()` return a vector of time points
- `data_sim.getSpecies()` return an array([[time], [species]])
- `data_sim.getRates()` return an array([[time], [rates]])
- `data_sim.getOutput(*args)` return an array consisting of time plus any specified species or rate e.g. `getOutput('s1', 'R1')`
- `data_sim.getDataAtTime(time)` return the results of the simulation at time point `time`.
- `data_sim.getDataInTimeInterval(time, bound)` return the simulation data in the interval $[time - bounds, time + bounds]$, if bounds is not specified it is assumed to be the step size.
- `data_sim.l_species` the species array labels
- `data_sim.l_rates` the rate array labels

Simulation results - legacy

> This sections is included for backwards compatibility and it is highly recommended that any new code use the `data_sim` object described in the previous section.

After any simulation has been completed the variable species concentration changes over time are stored in the array `mod.sim_res`. If the change in reaction rates over time is required, these can easily be generated using the `mod.Fix_Sim()` method:

```
sim_rates = mod.Fix_Sim(mod.sim_res,flux=1)
```

The `mod.Fix_Sim()` method outputs an array with time as the first column followed by either the species concentrations (no second argument or `flux=0`) or reaction rates (second argument `flux=1`) whose order is given by `mod.species` or `mod.reactions` respectively. If called without any argument `mod.Fix_Sim()` returns `mod.sim_res` with `time` as the first column.

Low level simulation parameters include:

- Simulation end time: `mod.sim_end`
- Number of points in the simulation: `mod.sim_points`
- LSODA maximum steps: `mod.lsoda_mxstep`. The default value, zero, causes LSODA to auto-adjust this parameter as necessary (to a maximum of 500). In systems that require more steps, the `mod.Simulate()` method automatically tries to adjust this parameter to a larger value.
- `mod.mode_sim_init` defaults to zero and the variable species are initialized with their initial values. A value of one initializes the variables with a small value.
- `mod.sim_time`: the time array used by `mod.Simulate()`.
- `mod.data_sim_add_rates`: automatically calculate and output reaction rates.

For the advanced user many more of LSODA's options are available for tweaking and are fully described in the PySCeS *Reference Manual*.

3.4 Steady-state analysis

PySCeS solves for a steady state using either the non-linear solver HYBRD[4] or NLEQ2[5] or FINTSLV (a forward integration algorithm). By default PySCeS has 'solver fallback' enabled which means that if a solver fails or returns an invalid result (i.e. contains negative concentrations) it switches to the next available solver. The solver chain is as follows:

1. HYBRD (can handle 'rough' initial conditions, converges quickly)
2. NLEQ2 (can deal with extremely non-linear systems, sensitive to bad conditioning, slightly slower)
3. FINTSLV (finds a result when max([species]) change is less than 0.1%, slow)

Solver fallback can be disabled by setting `mod.mode_solver_fallback = 0`. Each of the three solvers is highly configurable and although the default settings should work for most models a full description of all solver options can be found in the PySCeS *Reference Manual*.

To calculate a steady state use the `mod.doState()` method:

```
>>> mod.doState()
(hybrd) The solution converged.

INFO: Steady State evaluation complete.
```

[4]http://www.netlib.org/
[5]http://www.zib.de/SciSoft/ANT/nleq2.en.html

The results of a steady-state evaluation are attached as arrays as well as individual attributes and can be easily displayed using the `mod.showState()` method:

- `mod.showState()` displays the current steady-state values of both the species and fluxes.
- For each reaction (e.g. `R2`) a new attribute `mod.J_R2`, which represents it's steady-state value, is created.
- Similarly, each species (e.g. `mod.s2`) has a steady-state attribute `mod.s2_ss`
- `mod.state_species` in `mod.species` order.
- `mod.state_flux` in `mod.reactions` order.

There are various ways of initialising the steady-state solvers although, in general, the default values can be used.

- `mod.mode_state_init` initialises the solver using either the initial values (0), a value close to zero (1) or a quick simulation (2). Default behaviour is to use the initial values.
- `mod.zero_val` is the value used that is close to zero, the default is 1×10^{-8}

3.5 Reaction rates and fluxes

Most algorithms and methods will return species concentrations as their output. If, however, the reaction rates evolution over time or steady-state fluxes are required the system of rate equations can be solved using the relevant set of concentrations. In the simulation section the `mod.Fix_Sim()` was introduced which when given the results of a simulation (concentrations), could return the array of rates (including a time column)[6]. PySCeS also includes a more generic function, `mod.FluxGen()`, which when supplied with a concentration vector (or array) returns a vector (or array) of rates. Of course, given an array of steady-state concentrations `mod.FluxGen()` returns the steady-state fluxes.

```
rates = mod.FluxGen(concentrations)
```

3.6 Metabolic Control Analysis

For practical purposes the following methods are collected into a set of meta-routines that all first solve for a steady state and then the required Metabolic Control Analysis[7] (MCA) evaluation methods.

[6]This has been made obsolete by the data object described in Section 3.3

[7]Kacser, H. and Burns, J. A. (1973), *'The control of flux'*, Symp. Soc. Exp. Biol. **32**, 65–104 Heinrich and Rappoport (1974), *A linear steady-state treatment of enzymatic chains: General properties, control and effector strength*, Eur. J. Biochem. **42**, 89–95

Elasticities

The elasticities towards both the variable species and parameters can be calculated using `mod.doElas()` which generates as output:

- Scaled elasticities generated as `mod.ecRate_Species`, e.g. `mod.ecR4_s2`
- `mod.showEvar()` displays the non-zero elasticities calculated with respect to the variable species.
- `mod.showEpar()` displays the non-zero parameter elasticities.

Control coefficients

Both control coefficients and elasticities can be calculated using a single method, `mod.doMca()`.

- `mod.showCC()` displays the complete set of flux and concentration control coefficients.
- Individual control coefficients are generated as either `mod.ccSpecies_Rate` for a concentration control coefficient, e.g. `mod.ccs1_R4`.
- Similarly, `mod.ccJFlux_Rate` is a flux control coefficient e.g. `mod.ccJR1_R4`.

As it is generally common practice to use scaled elasticities and control coefficients PySCeS calculated these by default. However, it is possible to generate unscaled elasticities and control coefficients by setting the attribute `mod.mode_mca_scaled=0` in which case the model attributes are attached as `mod.uec` and `mod.ucc` respectively.

Response coefficients

A new PySCeS feature is the ability to calculate the parameter response coefficients for a model with the `mod.doMcaRC()` method. Unlike the elasticities and control coefficients the response coefficients are made available as a single attribute `mod.rc`. This attribute is a data object, containing the response coefficients as attributes and has the following methods:

- `rc.var_par` individual response coefficients can be accessed as attributes made up of `variable_parameter` e.g. `mod.rc.R1_k1`
- `rc.get('var', 'par')` return a response coefficient
- `rc.list()` returns all response coefficients as a dictionary of key:value pairs
- `rc.select('attr', search='a')` select all response coefficients that refer to `'attr'` e.g. `select('R1')` or `select('k2')`

- `rc.matrix`: the matrix of response coefficients
- `rc.row`: row labels
- `rc.col`: column labels

4 PySCeS: analysis functions

In this section we look at the higher level functions that can be used to perform different types of analyses on a model.

4.1 Single dimension parameter scans

PySCeS has the ability to quickly generate and plot single dimension parameter scans. Scanning a parameter typically involves changing a parameter through a range of values and recalculating the steady state at each step. Two methods are provided which simplify this task, `mod.Scan1()` is provided to generate the scan data while `mod.Scan1Plot()` is used to visualise the results. The first step is to define the scan parameters:

- `mod.scan_in` is a string defining the parameter to be scanned e.g. `'x0'`
- `mod.scan_out` is a list of strings representing the attribute names you would like to track in the output eg. `['J_R1','J_R2','s1_ss','s2_ss']`
- You also need to define the range of points that you would like to scan over. For a linear range SciPy[8] has a useful function `scipy.linspace(start, end, points)`. If you need to generate a log range use `scipy.logspace(start, end, points)`.

Both `scipy.linspace` and `scipy.logspace` use the number of points (including the start and end points) in the interval as an input. Additionally, the start and end values of `scipy.logspace` must be entered as indices, e.g. to start the range at 0.1 and end it at 100 you would write `scipy.logspace(-1, 2, steps)`. Setting up a PySCeS scan session might look something like:

```
import scipy
mod.scan_in = 'x0'
mod.scan_out = ['J_R1','J_R6','s2_ss','s7_ss']
scan_range = scipy.linspace(0,100,11)
```

Before starting the parameter scan, it is important to check that all the model attributes involved in the scan do actually exist. For example, `mod.J_R1` is created when `mod.doState()` is executed, likewise all the elasticities (`mod.ecR_S`) and control coefficients (`mod.ccJ_R`) are only created when the `mod.doMca()` method is called. If all the

[8]SciPy can be accessed by typing `import scipy` in your Python shell.

attributes exist you can perform a parameter scan using the `mod.Scan1(scan_range)` method which takes your predefined scan range as an argument:

```
mod.Scan1(scan_range)

Scanning ...
11 (hybrd) The solution converged.
(hybrd) The solution converged ...

done.
```

When the scan has been successfully completed[9], the results are stored in the array (`mod.scan_res`) that has `mod.scan_in` as its first column followed by columns that represent the data defined in `mod.scan_out`. If one or more of your input or output parameters is not a valid model attribute, it will be ignored. Once the parameter scan data has been generated, the next step is to visualise it using the `mod.Scan1Plot()` method:

```
mod.Scan1Plot()(plot=[], filename='', title='title', logx='' , logy='')
```

`mod.Scan1Plot()` behaves in a similar manner to `mod.SimPlot()` and when called with no arguments simply plots all the elements of `mod.scan_out` against `mod.scan_in`. You can, however, use the following arguments to modify the default behaviour:

- `plot=[]`, a list argument that if empty (default) is assumed to be `mod.scan_out`. However, any combination of `mod.scan_out` elements can be entered as a list and individually plotted.
- If a filename is set, `Scan1Plot()` will in addition to displaying the results on the screen, try to write a PNG image of the plot to a file: *filename.png*
- By default `Scan1Plot()` generates a title for the plot using the model file name plus the time the plot is generated. If required, custom graph titles can be set with `title='custom title'`
- If `logx` and `logx` are given a string value (`logx = 'on'`) the corresponding graph axis will be displayed using a logarithmic scale.

4.2 Multi-dimension parameter scans

This new PySCeS feature allows multi-dimensional parameter scanning. Any combination of parameters is possible and can be added as 'master' parameters that change

[9]If invalid steady states are generated during the scan they are replaced by NaN. Scan1 also reports the scan parameter values which generated the invalid states.

independently or 'slave' parameters whose change is coordinated with the previously defined parameter. Unlike `mod.Scan1()` this function is accessible as `pysces.Scanner` a custom class that can be instantiated with a loaded PySCeS model object:

```
>>> sc1 = pysces.Scanner(mod)
>>> sc1.addScanParameter('x3', 1, 10, 11)
>>> sc1.addScanParameter('k2', 0.1, 1000, 5, log=True)
>>> sc1.addScanParameter('k4', 0.1, 1000, 5, log=True, slave=True)
>>> sc1.addUserOutput('J_R1', 's1_ss')
>>> sc1.Run()

...
scan: 55 states analysed

>>> sc1_res = sc1.getResultMatrix()
>>> print sc1_res[0]
array([1., 0.1, 0.1, 97.94286647, 49.1380999])

>>> print sc1_res[-1]
array([1.0e+01, 1.0e+03, 1.0e+03, -3.32564878e+00, 3.84227702e-03])
```

In this scan we define two independent (`x3, k2`) and one dependent (`k3`) scan parameters and track the changes in the steady state variables `J_R1` and `s1_ss`. Note that `k2` and `k4` use a logarithmic scale. Once run the input parameters cannot be altered, however, the output can be changed and the scan rerun.

- `sc1.addScanParameter(name, start, end, points, log, slave)` where `name` is the input parameter (as a string), `start` and `end` define the range with the required number of `points`. While `log` and `slave` are boolean arguments indicating the point distribution and whether the axis is independent or not.
- `sc1.addUserOutput(*args)` an arbitrary number of model attributes to be output can be added (this method automatically tries to determine the level of analysis needed) e.g. `addUserOutput('J_R1', 'ecR1_k2')`
- `sc1.Run()` run the scan, if subsequent runs are required after changing output parameters use `sc1.RunAgain()`
- `sc1.getResultMatrix()` return the scan results as an array containing both input and output.
- `sc1.UserOutputList` the list of output names
- `sc1.UserOutputResults` an array containing only the output
- `sc1.ScanSpace` the generated list of input parameters.

5 Model attribute display functions

Besides the `showX()` methods already described in this guide, PySCeS has display methods which allow you to print various sets of information either o the screen or to a file. The most comprehensive of these functions `mod.showModel()` allows you to save the model object as a PySCeS input file.

5.1 Displaying/saving model attributes

All of the `showX()` methods, with the exception of `mod.showModel()` operate in exactly the same way. If called without an argument, they display the relevant information to the screen. Alternatively, if given an open, writable (ASCII mode) file object as an argument, they write the requested information to the open file. This allows the generation of customized reports containing only information relevant to the model. `mod.showModel()` acts as a 'File → save' function and simply needs to be given a filename to save the current model to disk as a PySCeS input file.

- `mod.showSpecies()` prints the current value of the model species (mod.M).
- `mod.showSpeciesI()` prints the initial, parsed in, value of the model species (mod.Mi).
- `mod.showPar()` prints the current value of the model parameters.
- `mod.showState()` prints the current steady-state fluxes and species.
- `mod.showConserved()` prints any moiety conserved relationships (if present).
- `mod.showFluxRelationships()` shows the relationships between dependent and independent fluxes at steady state
- `mod.showRateEq()` prints the reaction stoichiometry and rate equations.
- `mod.showODE()` prints the differential equations.
- `mod.showModel(filename=None)` prints the entire model as an input file format to the screen, or to a file names *filename.psc* if the filename argument is supplied.

Assuming you have loaded a model and run `mod.doState()` the following code opens a Python file object (`rFile`), writes the steady-state results to the file associated with the file object (`results.txt`) and then closes it again.

```
rFile = open('results.txt','w')
mod.showState()      # print the results to screen
mod.showState(rFile) # write the results to the file results.txt
rFile.close()
```

5.2 Writing formatted arrays

The `showX()` methods described in the previous sections allow the user a convenient way to write the predefined matrices either to screen or file. However, for maximum flexibility, PySCeS includes a suite of array writers that enable one to easily write, in a variety of formats any array to a file. Unlike the `showX()` methods, the `Write_array` methods are specifically designed to write to data to a file.

In most modelling situations it is rare that an array needs to be stored or displayed that does not have specific labels for its rows or columns. Therefore, all the `Write_array` methods take list arguments that can contain either the row or column labels. Obviously, these lists should be equal in length to the matrix dimension they describe and in the correct order.

There are currently three custom array writing methods that work either with a 1D (vector) or 2D arrays (matrices). To allow an easy comparison of these methods' output, all the following sections use the same example array as input.

`Write_array()`

The basic array writer is the `Write_array()` method. Using the default settings this method writes a 'tab delimited' array to a file. It is trivial to change this to a 'comma delimited' format by using the `separator = ' '` argument. Numbers in the array are formatted using the global number format.

If column headings are supplied using the `Col = []` argument they are written above the relevant column and if necessary truncated to fit the column width. If a column name is truncated it is marked with a * and the full length name is written as a comment after the array data. Similarly row data can be supplied using the `Row = []` argument in which case the row names are displayed as a comment which is written after the array data.

Finally, if the `close_file` argument is enabled the supplied file object is automatically closed after writing the array. The full call to the method is:

```
 mod.Write_array(input, File=None, Row=None, Col=None, separator='  ')
```

which generates the array

```
## Write_array_linear1_11:12:23
#s0             s1             s2
-3.0043e-001   0.0000e+000   0.0000e+000
 1.5022e+000  -5.0217e-001   0.0000e+000
 0.0000e+000   1.5065e+000  -5.0650e-001
 0.0000e+000   0.0000e+000   1.0130e+000
# Row: R1  R2  R3  R4
```

By default, each time an array is written, PySCeS includes an array header consisting of the model name and the time the array was written. This behaviour can be disabled by setting: `mod.write_array_header = 0`

`Write_array_latex()`

The `Write_array_latex()` method functions similarly to the generic `Write_array()` method except that it generates a formatted array that can be included directly in a LaTeX document. Additionally, there is no separator argument, column headings are not truncated and row labels appear to the left of the matrix.

```
 mod.Write_array_latex(input, File=None, Row=None, Col=None)
```

which generates

```
%% Write_array_latex_linear1_11:45:03
\[
\begin{array}{r|rrr}
  & $\small{s0}$ & $\small{s1}$ & $\small{s2}$ \\ \hline
 $\small{R1}$ &-0.3004 & 0.0000 & 0.0000 \\
 $\small{R2}$ & 1.5022 &-0.5022 & 0.0000 \\
 $\small{R3}$ & 0.0000 & 1.5065 &-0.5065 \\
 $\small{R4}$ & 0.0000 & 0.0000 & 1.0130 \\
 \end{array}
\]
```

and in a typeset document appears as

	s0	s1	s2
R1	−0.3004	0.0000	0.0000
R2	1.5022	−0.5022	0.0000
R3	0.0000	1.5065	−0.5065
R4	0.0000	0.0000	1.0130

6 PySCeS module functions

So far this guide has mostly focused on methods and attributes of the model class (i.e. `mod.X()`). In this section we examine generic methods that are made available as the modules `pysces.*`

6.1 Graph plotting and formatting using pysces.plt.*

The simulation and scanning sections described two built-in functions used to plot specific types of data. Currently, all plotting in Matplotlib [10] using the PyLab interface. This

[10] http://matplotlib.sourceforge.net

interface to focusses heavily on cross-platform interactive plotting and has been designed to use Matplotlib's TKagg back-end[11]. PySCeS makes the Pylab interface available as `pysces.plt.P.*`

Plotting 2D arrays

The main plotting method for 2D data[12] is the `pysces.plt.plot2D()`:

- `data` the array containing the data to be plotted
- `x` the (Python style) numeric index of the x-axis data range
- `ylist`: a list containing the indexes of the y-axis data ranges.
- `labels`: (optional) a list equal in length and in the same order as `data` that contains custom data labels that should be used in the graph key. If this argument is not present the column index is used as the key.
- `style` this is the line style to be used for all data series. It should be a Matplotlib[13] style format string.

The following code sample illustrates how to plot simulation results using the `pysces.plt` functionality:

```
>>> sim_s = mod.data_sim.getSpecies()
>>> sim_s_l = ['time'] + mod.data_sim.l_species
>>> pysces.plt.plot2D(sim_s, 0, [1,2,3], labels=sim_s_l, style='o-')
```

The most basic of the plotting methods is `plotX()`. This method, when given a matrix, uses the first column for the x-axis data range and the rest of the columns as y-axis data. This method is meant as a 'quick and dirty' way of visualising array data.

```
pysces.plt.plotX(data)
```

Saving graphs

Any active plot can be saved in one of two ways: by clicking the save button in the interactive window or by calling `pysces.plt.save` from the command line. This function simply needs to be called with a `fname` argument and results in the active plot to be written into the current working directory as *filename.png*. The optional string argument, `path`, allows a path (e.g. `'c:\\temp'`) to be specified.

```
pysces.plt.save(fname, path=None)
```

[11]While other back-ends may work they are not guaranteed to work properly.
[12]3D plots are currently not supported
[13]please see the Matplotlib documentation for more details

Formatting graphs

The following formatting methods and can be used interactive to manipulate the current active figure The first set affects the overall appearance of the graph.

- `gridon()`: enable grid display
- `gridoff()`: disable grid display

Next, the scaling used on the X and Y axis can be set with these methods.

- `logx()`: x-axis uses a logarithmic scale
- `logy()`: y-axis uses a logarithmic scale
- `linx()`: x-axis uses a linear scale
- `liny()`: y-axis uses a linear scale

In the case of the axis range methods, the `start` and `end` arguments are the minimum and maximum values of the relevant axis.

- `xrng(start, end)`
- `yrng(start, end)`

Finally, axis titles can be set using the label methods which take a single string argument.

- `xlabel(l='')`
- `ylabel(l='')`

7 Installing PySCeS 0.4.x

PySCeS is distributed as a Python package and is installed using the Python distribution utilities[14]. Before installing PySCeS you need at least a complete, working, version of SciPy 0.5.x (or newer). The basic prerequisites for building and installing PySCeS from scratch are:

- Python 2.4 or newer (`http://www.python.org/`)
- GCC compiler suite (including Fortran and C++ compilers, (3.3 or newer))
- Numpy – Numerical Python (`http://numpy.sourceforge.net/`)
- SciPy – Scientific libraries for Python (`http://www.scipy.org/`)

[14]in conjunction with the Numpy distutils extensions

- Matplotlib – Plotting library for Python (http://matplotlib.sourceforge.net/), be sure to install/enable the TKagg backend (especially on Linux).

We strongly recommend (especially for user interactivity and graphing purposes[15]) that the following Python modules also be installed:

- iPython – an extended Python editing shell (http://ipython.scipy.org/)
- wxPython – a cross-platform GUI toolkit (http://www.wxpython.org/download.php)

On systems using Microsoft Windows^{TM} 2000 or XP PySCeS has been designed to be compatible with the MinGW (http://www.mingw.org/) GCC compilers[16]. Using a Posix environment such as Cygwin should also be possible. As it can be daunting to compile and build applications on Windows, a binary release of PySCeS and its associated support applications are available from the main PySCeS site[17].

As part of the setup process, PySCeS installs a number of Python packages that are critical for its functioning. These bundled modules include David Beazley's Python Lex–Yacc or PLY[18]. Assuming the required packages are properly install PySCeS can be built as follows:

- Linux/Cygwin: `python setup.py install`
- Windows (MinGW): `python setup.py config --compiler=mingw32 build --compiler=mingw32 install`

Additionally, two external applications are compiled and installed with PySCeS: Thomas Pfeiffer's MetaTool[19]

```
METATOOL is a C program developed from 1998 to 2000 by Thomas
Pfeiffer (Berlin) in cooperation with Stefan Schuster and
Ferdinand Moldenhauer (Berlin) and Juan Carlos Nuno (Madrid).
```

which consists of two executable binaries that are used by PySCeS to calculate elementary modes and the non-linear solver library NLEQ2. The ZIB institute (http://www.zib.de/) has kindly given us permission to use and distribute the NLEQ2 algorithm as part of PySCeS, providing such usage does not contravene their licence terms as described in `nleq2/nleq2_readme`.

```
* Licence
  You may use or modify this code for your own non commercial
```

[15]and generally a happier modelling experience!

[16]MinGW release 3.1.0-1 has been successfully used to compile PySCeS and SciPy on Windows 2000/XP

[17]http://pysces.sourceforge.net/download.html

[18]http://systems.cs.uchicago.edu/ply/

[19]http://www.biologie.hu-berlin.de/biophysics/Theory/tpfeiffer/metatool.html

```
purposes for an unlimited time.
In any case you should not deliver this code without a special
permission of ZIB.
In case you intend to use the code commercially, we oblige you
to sign an according licence agreement with ZIB.
```

If you are not using PySCeS for research or personal use you **must** disable NLEQ2 installation by setting nleq2 = 0 in `setup.py` or obtain a licence from ZIB.