

---

# **PySCeS User Guide**

***Release 0.7.0***

**Brett Olivier, Johann Rohwer, Jannie Hofmeyr**

December 12, 2008



# CONTENTS

<b>I</b>	<b>Introduction</b>	<b>1</b>
<b>II</b>	<b>Getting started</b>	<b>5</b>
<b>1</b>	<b>Loading PySCeS</b>	<b>7</b>
<b>2</b>	<b>Creating a PySCeS model object</b>	<b>9</b>
2.1	Advanced . . . . .	9
<b>3</b>	<b>Creating a PySCeS model object</b>	<b>11</b>
<b>4</b>	<b>Basic model attributes</b>	<b>13</b>
4.1	Advanced . . . . .	14
<b>III</b>	<b>Modelling</b>	<b>15</b>
<b>5</b>	<b>Structural Analysis</b>	<b>17</b>
5.1	Structural Analysis - new objects . . . . .	17
5.2	Structural Analysis - legacy . . . . .	18
<b>6</b>	<b>Time simulation</b>	<b>19</b>
6.1	Simulation results . . . . .	20
6.2	Advanced . . . . .	21
<b>7</b>	<b>Steady-state analysis</b>	<b>23</b>
7.1	New: mod.data_sstate . . . . .	24
7.2	Stability . . . . .	24
<b>8</b>	<b>Metabolic Control Analysis</b>	<b>25</b>
8.1	Elasticities . . . . .	25
8.2	Control coefficients . . . . .	25
8.3	Response coefficients . . . . .	26

<b>IV</b>	<b>Parameter scanning</b>	<b>27</b>
9	Single dimension parameter scans	29
10	Two dimension parameter scans	31
11	Multi-dimension parameter scans	33
<b>V</b>	<b>Plotting</b>	<b>35</b>
<b>VI</b>	<b>Displaying data</b>	<b>41</b>
12	Displaying/saving model attributes	43
13	Writing formatted arrays	45
13.1	Write_array() . . . . .	45
13.2	Write_array_latex() . . . . .	46
<b>VII</b>	<b>Installing and configuring</b>	<b>47</b>
14	Configuring	51
<b>VIII</b>	<b>References</b>	<b>53</b>
<b>IX</b>	<b>The PySCeS Model Description Language</b>	<b>57</b>
15	Defining a PySCeS model	61
15.1	A kinetic model . . . . .	61
15.2	Model keywords . . . . .	61
15.3	Global unit definition . . . . .	62
15.4	Symbol names and comments . . . . .	62
15.5	Compartment definition . . . . .	63
15.6	Function definitions . . . . .	63
15.7	Defining fixed species . . . . .	64
15.8	Reaction stoichiometry and rate equations . . . . .	64
15.9	Species and parameter initialisation . . . . .	66
16	Advanced model construction	69
16.1	Assignment rules . . . . .	69
16.2	Rate rules . . . . .	69
16.3	Events . . . . .	70
16.4	Piecewise . . . . .	70
17	Example PySCeS input files	73
17.1	Basic model definition . . . . .	73

17.2 Advanced example . . . . .	74
<b>X PySCeS Module documentation</b>	<b>77</b>
18 PyscesPlot2	79
19 PyscesModel	89
20 PyscesScan	107
21 PyscesInterfaces	111
22 PyscesStoich	113
23 PyscesLink	119
<b>XI Indices and tables</b>	<b>121</b>



# **Part I**

## **Introduction**



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

Welcome! This users guide will get you started with 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 straight on if not, *Installing and configuring* contains instructions on building and installing PySCeS.

PySCeS is distributed under the PySCeS (BSD style) license and is made freely available as Open Source software. See LICENCE.txt for details.

PySCeS continued development depends, to a large degree on support and feedback from Systems Biology community, if you use PySCeS in your work please cite it using the following reference:

Brett G. Olivier, Johann M. Rohwer and Jan-Hendrik S. Hofmeyr *Modelling cellular systems with PySCeS*, Bioinformatics, 21, 560-561, DOI 10.1093/bioinformatics/bti046.

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.

**The PySCeS development team.**



## **Part II**

### **Getting started**



## Loading PySCeS

In this section we assume you have PySCeS installed and configured (see Installing and configuring 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 Description Language see the *PySCeS Input File Guide*. Note that on all platforms PySCeS model files have the extension `.psc`.

To begin modelling we need to start up an interactive Python shell (we suggest `iPython`) and load PySCeS with `import pysces`:

```
Python 2.5.2 (r252:60911, Feb 21 2008, 13:11:45) [MSC v.1310 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.
```

```
>>> import pysces
```

```
Matplotlib interface loaded (pysces.plt.m)
GnuPlot interface loaded (pysces.plt.g)
Continuation routines available
NLEQ2 routines available
You are using NumPy (1.2.1) with SciPy (0.7.0b1)
PySundials available
```

```
PySCeS environment
```

```
*****
pysces.model_dir = C:\mypysces\pscmmodels
pysces.output_dir = C:\mypysces
```

```
*****
* Welcome to PySCeS (0.7.0) - Python Simulator for Cellular Systems *
*           http://pysces.sourceforge.net                       *
*           Somewhere In Time                                  *
* Copyright(C) B.G. Olivier, J.M. Rohwer, J.-H.S. Hofmeyr, 2004-2009 *
* Triple-J Group for Molecular Cell Physiology                 *
* Stellenbosch University, South Africa                       *
* PySCeS is distributed under the PySCeS (BSD style) licence, see *
* LICENCE.txt (supplied with this release) for details        *
*****
```

PySCeS is now ready to use. If you would like to test your installation try running the test suite:

```
pysces.test()
```

this also copies the test models supplied with PySCeS into your model directory.

## Creating a PySCeS model object

This guide uses the test models supplied with PySCeS as examples, if you would like to use them and have not already done so run the PySCeS tests (described in the previous section).

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: C:\mypysces\pscmodels
C:\mypysces\pscmodels\pysces_test_linear1.psc loading .....
```

When instantiating a new model object, PySCeS input files are assumed to have a `.psc` extension. 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 filename.

### 2.1 Advanced

The model constructor can also be used to specify a model directory other than the default model path:

```
mod = pysces.model('pysces_test_linear1', dir='c:\\Pysces\\psc')
```

alternatively input files can also be loaded from a string:

```
>>> F = file('c:\\Pysces\\psc\\pysces_test_linear1.psc', 'r')
>>> pscS = F.read()
>>> F.close()
>>> mod = pysces.model('test_lin1s', loader='string', fString=pscS)
```

```
Assuming extension is .psc
Using model directory: C:\mypysces\pscmodels
```

```
Using file: test_lin1s.psc  
C:\mypysces\pscmmodels\orca\test_lin1s.psc loading .....
```

note that now the input file is saved and loaded as `model_dir\orca\test_lin1s.psc`.

## Creating a PySCeS model object

Once a new model object has been created it needs to be loaded. During the load process the input file is parsed, the model description is translated into Python data structures and a stoichiometric structural analysis is performed. All this is done by simply calling `mod.doLoad()`:

```
>>> mod.doLoad()
```

```
Parsing file: C:\mypysces\pscmmodels\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 model (`mod`) attributes so that in the input file where you might find initialisations such as `s1 = 1.0` and `k1 = 10.0` these are now available as `mod.s1` and `mod.k1`. For variable species and compartments an additional attribute is created which contains the elements *initial* (as opposed to current) value. These are constructed as `<name>*_init*`:

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

Any errors generated during the loading process (almost always) occur as a result of syntax errors in the input file. These error messages may not be intuitive for example, ‘list out of range’ exception usually indicates a missing multiplication operator “3(” instead of “3\*(” or unbalanced parentheses.



## Basic model attributes

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

- `mod.ModelFile`, the name of the model file that was used.
- `mod.ModelDir`, the input file directory.
- `mod.ModelOutput`, the PySCeS work/output directory.
- Parameters are available as attributes directly as specified in the input file e.g. `k1` is `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.
- Compartments are also assigned an initial value.
- Rate equations are translated into objects that return their current value when called `mod.R1()`.

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

Groups of model properties (either tuples, lists or dictionaries)

- `mod.species` the model's variable species names (ordered relative to the stoichiometric matrix rows).
- `mod.reactions` reaction names and ordered to the stoichiometric matrices columns.
- `mod.parameters` all parameters (including fixed species)
- `mod.fixed_species` only the fixed species names
- `mod.__rate_rules__` a list of rate rules defined in the model

## 4.1 Advanced

These attributes are used by PySCeS to store additional information about the basic model components, generally they are supplied by the parser and should almost never be changed directly.

- `mod.__events__` a list of event object references which can be interrogated for event information. For example if you want a list of event names try `[ev.name for ev in mod.__events__]`
- `mod.__rules__` a dictionary containing information about all rules defined for this model
- `mod.__sDict__` a dictionary of species information
- `mod.__compartments__` a dictionary containing compartment information

# **Part III**

## **Modelling**



---

# Structural Analysis

As part of the model loading procedure, `doLoad()` automatically performs a stoichiometric (structural) analysis of the model. The structural properties of the model are captured in stoichiometric matrix (**N**), kernel matrix (**K**) and link matrix (**L**). These matrices can either be displayed with a `mod.showX()` method or used in further calculations as numeric arrays. The formal definition of these matrices, as they are used in PySCeS, is described in <sup>1</sup>.

The structural properties of a model are available in two forms, as new-style objects which have all the array properties neatly encapsulated or as legacy attributes. Although both exist it is highly recommended to use the new objects.

## 5.1 Structural Analysis - new objects

For alternate descriptions of these model properties see the next (legacy) section.

- `mod.Nmatrix` view with `mod.showN()`
- `mod.Nrmatrix` view with `mod.showNr()`
- `mod.Lmatrix` view with `mod.showK()`
- `mod.L0matrix`
- `mod.Kmatrix` view with `mod.showL()`
- `mod.K0matrix`
- `mod.showConserved()` displays any moiety conserved relationships (if present).
- `mod.showFluxRelationships()` shows the relationships between dependent and independent fluxes at steady state.

All new structural objects have an *array* attribute which holds the actual NumPy array data 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

## 5.2 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 **N<sub>r</sub>** is available as `mod.nrmatrix` and is displayed with `mod.showNr()`. If there is no dependence **N<sub>r</sub> = 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 (**K<sub>0</sub>**, **L<sub>0</sub>**) are available as `mod.kzeromatrix` and `mod.lzeromatrix`.
- `mod.showConserved()` prints any moiety conserved relationships (if present).
- `mod.showFluxRelationships()` shows the relationships between dependent and independent fluxes at steady state.

If the `mod.showX()` methods are used the row and column titles of the various matrices 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`).

## Time simulation

PySCeS has interfaces to two ODE solvers either LSODA from ODEPACK (part of SciPy) or SUNDIALS CVODE (using PySundials). If PySundials is installed it will automatically select CVODE if compartments, events or rate rules are detected during model load as LSODA is not able capable of event handling or changing compartment sizes. If, however, you would like to select the solver manually this is also possible:

```
mod.mode_integrator = 'LSODA'  
mod.mode_integrator = 'CVODE'
```

There are three ways of running a simulation:

1. Defining the *start*, *end* time and number of *points* and using the `mod.Simulate()` method directly:

```
mod.sim_start = 0.0  
mod.sim_end = 20  
mod.sim_points = 50  
mod.Simulate()
```

2. Using the `mod.doSim()` method where only the *end* time and *points* need to be specified. For example running a 20 point simulation from time 0 to 10:

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

3. Or using `mod.doSimPlot()` which runs the simulation and displays the results. In addition to `doSim`'s arguments the following arguments may be used:

```
mod.doSimPlot(end=10.0, points=21, plot='species', fmt='lines', filename=None)
```

where:

- *plot* can be one of species, rates or all.
- *fmt* plot format, UPI backend dependent (default='') or the *CommonStyle* 'lines' or 'points'.
- *filename* if not None (default) then the plot is exported as *filename.png*

Another way of quickly visualising the results of a simulation is to use the `mod.SimPlot` method:

```
mod.SimPlot(plot='species', filename=None, title=None, log=None, format='lines')
```

where:

- *plot*: output to plot (default='species')
- 'all' rates and species
- 'species' species
- 'rates' reaction rates
- [*SI*, *RI*, ] a list of model attributes (species, rates)
- *filename* (optional) if not None file is exported to filename (default=None)
- *title* the plot title (default=None)
- *log* use log axis for 'x', 'y', 'xy' (default=None)
- *fmt* plot format, UPI backend dependent (default='') or the *CommonStyle* 'lines' or 'points'.

Called without arguments, `mod.SimPlot()` plots all the species concentrations against time.

## 6.1 Simulation results

In PySCeS 0.7.x the simulation results have been consolidated into a new `mod.data_sim` object. By default species concentrations/amounts, reaction rates and rate rules are automatically added to the `data_sim` object. If extra information (parameters, compartments, assignment rules) is required this can easily be added using `mod.CVODE_extra_output`, a list containing any model attribute which is not added by default.

The `mod.data_sim` object which has many methods for extracting simulation data including:

- `data_sim.getTime()` return a vector of time points
- `data_sim.getSpecies()` returns array([[time], [species]])
- `data_sim.getRates()` returns array([[time], [rates]])
- `data_sim.getRules()` returns array([[time], [rate rules]])
- `data_sim.getXData` returns array([[time], [CVODE\_extra\_output]])
- `data_sim.getSimData(*args)` return an array consisting of *time* plus any available data series:

```
mod.data_sim.getSimdata('s1', 'R1', 'Rule1', 'xData2')
```

- `data_sim.getAllSimData(*args)` return an array of all simulation data
- `data_sim.getDataAtTime(time)` return the results of the simulation at *time*.
- `data_sim.getDataInTimeInterval(time, bound)` return the simulation data in the interval [time-bound, time+bound], if *bound* is not specified it is assumed to be the step size.

All the `data_sim.get*` methods by default only return a NumPy array containing the requested data, however if the argument `lbls` is set to `True` then both the array as well as a list of column labels is returned:

```
Sdata, Slabels = mod.data_sim.getSpecies(lbls=True)
```

This is very useful when using the PySCeS plotting interface (described later in this guide) to plot simulation results.

## 6.2 Advanced

PySCeS sets integrator options that attempt to configure the integration algorithms to suit a particular model. However, almost every integrator option can be overridden by the user. Simulator settings are stored in PySCeS `mod.__settings__` dictionary. For LSODA some useful keys are (`mod.__settings__[key]`):

```
'lsoda_atol': 1.0e-012
'lsoda_rtol': 1.0e-007
'lsoda_mxordn': 12
'lsoda_mxords': 5
'lsoda_mxstep': 0
```

`atol` and `rtol` are the absolute and relative tolerances, while `mxstep=0` means that LSODA chooses the number of steps (up to 500). If this is still not enough, PySCeS automatically increases the number of steps necessary to find a solution.

Additionally, CVODE allows per step step-size optimisation and automatic tolerance scaling:

```
'cvsode_abstol': 1.0e-15
'cvsode_abstol_factor': 1.0e-8
'cvsode_auto_tol_adjust': True
'cvsode_mxstep': 1000
'cvsode_reltol': 1.0e-9
'cvsode_stats': False
```

`cvsode_abstol` is considered to be the minimum absolute tolerance, PySCeS first uses the initial species values multiplied by `cvsode_abstol_factor` (so that  $[s][factor] \geq [abstol]$ ) to calculate

its absolute tolerance. Once the simulation is underway PySCeS periodically readjusts the absolute tolerance on a per species basis based on the current species value.

If CVODE cannot find a solution in the given number of steps it automatically increases *cvode\_mxstep* and tries again, however, it also keeps track of the number of times that this adjustment is required and if a specific threshold is passed it will begin to increase *cvode\_reltol* by 1.0e3 (to a maximal value of 1.0e-3). Finally, if *cvode\_stats* is enabled CVODE will display a report of its internal parameters after the simulation is complete.

## Steady-state analysis

PySCeS solves for a steady state using either the non-linear solvers **HYBRD**, **NLEQ2** or forward integration. 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** (highly optimised for extremely non-linear systems, more sensitive to bad conditioning and slightly slower convergence).
3. **FINTSLV** (finds a result when the change in  $\max([\text{species}])$  is less than 0.1%; slow convergence).

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 configurable options can be set in by way of the `mod.__settings__` dictionary.

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

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

The results of a steady-state evaluation are stored 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 its 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). The default behaviour is to use the initial values.

## 7.1 New: `mod.data_sstate`

New to PySCeS 0.7 is the `mod.data_sstate` object that by default stores steady-state data (species, fluxes, rate rules) in a manner similar to `mod.data_sim`. One notable exception is that the current steady-state values are also made available as attributes to this object (e.g. species S1's steady-state value is stored as `mod.data_sstate.S1`). Using the `mod.STATE_extra_output` list it is possible to store user defined data in the `data_sstate` object. Steady-state data can be easily retrieved using the by now familiar `.get*` methods.

- `data_sstate.getSpecies()` returns a species array
- `data_sstate.getFluxes()` returns a flux array
- `data_sstate.getRules()` returns a rate rule array
- `data_sstate.getXData()` returns an array defined in `STATE_extra_output`
- `data_sstate.getStateData(*args)` return user defined array of data ('S1','R2')
- `data_sstate.getAllStateData()` return all state data as an array

All these methods also accept the `lbls=True` argument in which case they return both array data and a label list:

```
ssdat, sslbl = mod.data_sstate.getSpecies(lbls=True)
```

## 7.2 Stability

PySCeS can analyse the stability of systems that can attain a steady state. It does this by calculating the Eigen values of the Jacobian matrix for the reduced system of independent ODE's:

- `mod.doEigen()` calculates a steady-state and performs the stability analysis
- `mod.showEigen` prints out a stability report
- `mod.doEigenShow()` combines both of the above

The Eigen values are also available as attributes `mod.lambda1` etc. By default the Eigen values are stored as `mod.eigen_values` but if `mod.__settings__['mode_eigen_output'] = 1` is set both the Eigen values as well as the left and right Eigen vectors are stored as `mod.eigen_vecleft` and `mod.eigen_vecright` respectively. Please note that there is currently no guarantee that the order of the Eigen value array corresponds to the species order.

# 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 (MCA) <sup>2, 3</sup> evaluation methods.

## 8.1 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.

As a prototype we also store the elasticities in an object, `mod.ec.*` this may become the default way of accessing elasticity data in future releases but has not been stabilised yet.

## 8.2 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.__settings__['mode_mca_scaled'] = 0` in which case the model attributes are attached as `mod.uec` and `mod.ucc` respectively.

As a prototype we also store the elasticities in an object, `mod.cc.*` this may become the default way of accessing control coefficient data in future releases but has not been stabilised yet.

### 8.3 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

## **Part IV**

### **Parameter scanning**



## 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 re-calculating 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 has a useful function `scipy.linspace(start, end, points)` (SciPy can be accessed by typing `import scipy` in your Python shell). 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 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, 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 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.} . 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=[], title=None, log=None, format='lines', filename=None)
```

- *plot* if empty `mod.scan_out` is used, otherwise any subset of `mod.scan_out` (default=[])
- *filename* the filename of the PNG file (default=None, no export)
- *title* the plot title (default=None)
- *log* if None a linear axis is assumed otherwise one of ['x','xy','xyz'] (default=None)
- *format* the backend dependent line format (default='lines') or the *CommonStyle* 'lines' or 'points'.

Called without any arguments `Scan1Plot` plots all of `mod.sim_out` against `mod.sim_in`.

## Two dimension parameter scans

Two dimension parameter scans can also easily be generated using the `mod.Scan2D` method:

```
>>> mod.Scan2D(p1, p2, output, log=False)
```

- *p1* is a list of [model parameter 1, start value, end value, points]
- *p2* is a list of [model parameter 2, start value, end value, points]
- *output* the steady-state variable e.g. 'J\_R1' or 'A\_ss'
- *log* if True scan using log ranges for both axes

To plot the results of two dimensional scan use the `mod.Scan2DPlot` method. Note that as Matplotlib cannot produce 3D plots the GnuPlot interface must be active (see the section on plotting later on in this guide):

```
>>> mod.Scan2DPlot(title=None, log=None, format='lines', filename=None)
```

- *filename* the filename of the PNG file (default=None, no export)
- *title* the plot title (default=None)
- *log* if None a linear axis is assumed otherwise one of ['x','xy','xyz'] (default=None)
- *format* the backend dependent line format (default='lines') or the *CommonStyle* 'lines' or 'points'.



## 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 independently or *slave* parameters whose change is coordinated with the previously defined parameter. Unlike `mod.Scan1()` this function is accessed via the `pysces.Scanner` class that is 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 necessary) e.g. `addUserOutput('J_R1', 'ecR1_k2')`
- `sc1.Run()` run the scan, if subsequent runs are required after changing output parameters use `sc1.RunAgain()`. Note that it is not possible to change the input parameters

once a scan has been run, if this is required a new Scanner object should be created.

- `sc1.getResultMatrix(stst=False)` return the scan results as an array containing both input and output if `stst = True` append the steady-state fluxes and concentrations to the user output so that output has dimensions `[scan_parameters]+[state_species+state_flux]+[Useroutput]` otherwise return the default `[scan_parameters]+[Useroutput]`.
- `sc1.UserOutputList` the list of output names
- `sc1.UserOutputResults` an array containing only the output
- `sc1.ScanSpace` the generated list of input parameters.

# **Part V**

## **Plotting**



The PySCeS plotting interface has been completely rewritten to facilitate the use of multiple plotting back-ends via a Unified Plotting Interface (UPI). Using the UPI we ensure that a specified subset of plotting methods is back-end independent (although the UPI can be extended with back-end specific methods). So far Matplotlib (default) and GnuPlot back-ends have been implemented.

The common UPI functionality is accessible as `pysces.plt.*` while back-end specific functionality is available as `pysces.plt.m` (Matplotlib) and `pysces.plt.g` (GnuPlot).

While the Matplotlib is activated by default GnuPlot needs to be enabled (see Configuring PySCeS section) and then activated using `pysces.plt.p_activateInterface('gnuplot')`. All installed interfaces can be activated or deactivated as required:

```
>>> pysces.plt.p_activateInterface(interface)
>>> pysces.plt.p_deactivateInterface(interface)
```

where *interface* is either 'matplotlib' or 'gnuplot'. The PySCeS UPI defines currently has the following methods:

`plot(data, x, y, title="", format=")` plot a single line data[y] vs data[x]

- *data* the data array
- *x* x column index
- *y* y column index
- *title* is the line key
- *format* is the backend format string (default='')

`plotLines(data, x, y=[], titles=[], formats=["])` plot multiple lines i.e. data[y1, y2, ] vs data[x]

- *data* the data array
- *x* x column index
- *y* is a list of line indexes, if empty all of y not including x is plotted
- *titles* a list of line keys, if empty Line1, Line2, etc is used
- *formats* a list (per line) of format strings, if formats only contains a single item, this format is used for all lines.

`splot(data, x, y, z, title="", format=")` plot a surface i.e. data[z] vs data[y] vs data[x]

- *data* the data array
- *x* x column index

- *y* y column index
- *z* z column index
- *title* the surface key
- *format* a format string (default='')

`plotSurfaces(data, x, y, z=[], titles=[], formats=[""])` plot multiple surfaces i.e. `data[z1, z2, ]` vs `data[y]` vs `data[x]`

- *data* the data array
- *x* x column index
- *y* y column index
- *z* a list of z column indexes, if empty all data not including x, y are plotted
- *titles* a list of surface keys, if empty Surf1, Surf2 etc. is used
- *formats* is a list (per line) of format strings (default='')

If *formats* only contains a single item, this format is used for all surfaces.

`replot()` replot the current figure using all active interfaces (useful with GnuPlot type interfaces)

`save(name, directory=None, dfmt='%.8e')` save the plot data and (if possible) the back-end specific format file

- *filename* the filename
- *directory* optional (default = current working directory)
- *dfmt* the data format string (default='%.8e')

`export(name, directory=None, type='png')` export the current plot as a <format> file (currently only PNG is guaranteed to be available on all back-ends).

- *filename* the filename
- *directory* optional (default = current working directory)
- *type* the file format (default='png').

`setGraphTitle(title='PySCeS Plot')` set the graph title, unset if *title=None*

- *title* (string, default='PySCeS Plot') the graph title

`setAxisLabel(axis, label=")` sets one or more axis label

- *axis* x, y, z, xy, xz, yz, zyx

- *label* label string (default=None)

Called with only the *axis* argument clears that axis' label.

`setKey(value=False)` enable or disable the current plot key, no arguments removes key.

- *value* boolean (default = False)

`setLogScale(axis)` set *axis* to log scale

- *axis* is one of x, y, z, xy, xz, yz, zyx

`setNoLogScale(axis)` set axis to a linear scale

- *axis* is one of x, y, z, xy, xz, yz, zyx

`setRange(axis, min=None, max=None)` set one or more axis range

- *axis* is one of x, y, z, xy, xz, yz, zyx
- *min* is the range(s) lower bound (default=None, back-end auto-scales)
- *max* range(s) upper bound (default=None, back-end auto-scales)

`setGrid(value)` enable or disable the graph grid

- *value* (boolean) True (on) or False (off)

`plt.closeAll()` Close all active Matplotlib figures



# **Part VI**

## **Displaying data**



## 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 customised reports containing only information relevant to the model.

- `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.

Please note that the `mod.showModel()` method is not recommended for saving models as a PySCeS input file instead use the Core2 based `pysces.interface.writeMod2PSC` method instead:

```
>>> pysces.interface.writeMod2PSC(mod, filename, directory, iValues=True, getstr
```

- *filename*: writes `<filename>.psc` or `<model_name>.psc` if None
- *directory*: (optional) an output directory
- *iValues*: if True (default) then the models initial values are used (or the current values if False).
- *getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

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

## 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 the output of these methods, all the following sections use the same example array as input.

### 13.1 `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`

## 13.2 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

# **Part VII**

## **Installing and configuring**



Before installing or building PySCeS the following software is required:

- Python 2.5 (or 2.4 plus the Elementree/cElementree packages)
- Numpy 1.2+
- SciPy 0.7.0 (0.6.x will work with NumPy > 1.0.5)
- GCC 4.2+ on Linux or MinGW GCC 3.4.5 on Windows is required for building PySCeS from source only
- Matplotlib 0.98.3 with the TkAgg backend (this is the default, but optional, plotting package but can be replaced with GnuPlot)

optional, but highly recommended, packages:

- libSBML 3.x install with the Python bindings for SBML support
- GnuPlot alternative plotting back-end
- iPython highly recommended for interactive modelling sessions
- SciTE editor for editing and running PySCeS based modelling programs

This software stack provides a powerful scientific programming platform which is used by PySCeS to provide a flexible Systems Biology Modelling environment.

PySCeS 0.7.0 itself has been modularised into a main package and a (growing) number of support modules which extends its core functionality. The most important of these is the advanced simulation support added by installing PySundials (<http://pysundials.sf.net>). Linux users should build and install the SUNDIALS library and PySundials (build instructions on the PySundials web site). Windows users can simply download and install the *pysces\_pysundials* module.

- *pysces\_pysundials* a binary port of SUNDIALS+PySundials for Windows
- *pysces\_metatool* adds elementary mode support to PySCeS using MetaTool
- *pysces\_mariner* SOAP based web services gateway, including a PySCeS server and remote client
- *pysces\_kraken* (coming soon) PySCeS distributed processing module (currently distributed with PySCeS)

PySCeS and its extension modules use either the Python distutils or the Numpy distutils extensions. Assuming you have working versions of NumPy and SciPy on a Linux type operating systems building PySCeS is as easy as:

```
python setup.py install
```

On Windows (with MinGW) depending on your system configuration this becomes:

```
python setup.py config --compiler=mingw32 build --compiler=mingw32 install
```

In this release we have started prototyping Python egg support (currently only for windows) which is implemented via the *setupegg.py* build scripts.

By default PySCeS installs with a version of ZIB's NLEQ2 non-linear solver. This software is distributed under its own non-commercial licence. Please see the README.txt document provided with this PySCeS installation for details.

## Configuring

PySCeS has two configuration (\*.ini) files that allows one to specify global (per installation) and local (per user options). Currently the multiuser options are only fully realised on Linux based systems. Global options are stored in the *pyscfg.ini* file which is created in your PySCeS installation directory (this is a Windows version with the Linux defaults indicated with in *\*value\**):

```
[Pysces]
install_dir = c:\python25\lib\site-packages\pysces
gnuplot_dir = c:\model\gnuplot\binaries
model_dir = os.path.join(os.getenv('HOMEDRIVE')+os.path.sep,'Pysces','psc')
output_dir = os.path.join(os.getenv('HOMEDRIVE')+os.path.sep,'Pysces')
*model_dir = os.path.join(os.path.expanduser('~'),'Pysces','psc')*
*output_dir = os.path.join(os.path.expanduser('~'),'Pysces')*
```

The *[Pysces]* section contains information on the installation directory, the directory where the GnuPlot executable(s) can be found and the default model file and output directories. As we shall see some of these defaults can be overruled by the local configuration options:

```
[ExternalModules]
nleq2 = True

[PyscesModules]
pitcon = True
```

These sections define whether 3rd party algorithms (e.g. NLEQ2) are available for use, while the last section allows the alternate plotting backends to be enabled or disabled:

```
[PyscesConfig]
gnuplot = True
matplotlib = True
```

The user configuration files (*pys\_usercfg.ini*) are created when PySCeS is imported/run for the *first time*. On Windows this is in *<HOMEDRIVE>\Pysces* while on Linux this is in *\$HOME\Pysces*. Once created the user configuration files can be edited and will be used for every subsequent PySCeS session:

```
[Pysces]
output_dir = C:\mypysces
model_dir = C:\mypysces\psmodels
gnuplot = False
```

Here I have customised my default model and output directories and disabled GnuPlot (enabled above). If required *gnuplot\_dir* can also be set to point to an alternate location on a per user basis. Once you have PySCeS configured to your personal requirements you are ready to begin modelling.

**Part VIII**

**References**



---

<sup>1</sup>Hofmeyr, J.-H.S. (2001) *Metabolic control analysis in a nutshell*, in T.-M. Yi, M. Hucka, M. Morohashi, and H. Kitano, eds, Proceedings of the 2nd International Conference on Systems Biology, pp. 291-300.

<sup>2</sup>Kacser, H. and Burns, J. A. (1973), *The control of flux*, Symp. Soc. Exp. Biol. **32**, 65-104.

<sup>3</sup>Heinrich and Rappoport (1974), *A linear steady-state treatment of enzymatic chains: General properties, control and effector strength*, Eur. J. Biochem. **42**, 89-95.



## **Part IX**

# **The PySCeS Model Description Language**



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

PySCeS uses an ASCII text based *input file* to describe a cellular system in terms of its stoichiometry, kinetics, compartments and parameters. Input files may have any filename with the single restriction that, for cross platform compatibility, they must end with the extension *.psc*. In this document we describe the PySCeS Model Description Language (MDL) which has been updated and extended for the PySCeS 0.7.x release.

PySCeS is distributed under the PySCeS (BSD style) license and is made freely available as Open Source software. See LICENCE.txt for details.

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 and specifically the PySCeS MDL please let us know.



# Defining a PySCeS model

## 15.1 A kinetic model

The basic description of a kinetic model in the PySCeS MDL contains the following information:

- whether any fixed (boundary) species are present
- the reaction network stoichiometry
- rate equations for each reaction step
- parameter and boundary species initial values
- the initial values of the variable species

Although it is in principle possible to define an ODE based model without reactions or free species, for practical purposes PySCeS requires a minimum of a single reaction. Once this information is obtained it can be organised and written as a PySCeS input file. While this list is the minimum information required for a PySCeS input file the MDL allows the definition of advanced models that contain compartments, global units, functions, rate and assignment rules.

## 15.2 Model keywords

In PySCeS 0.7.x it is now possible to define keywords that specify model information. Keywords have the general form

```
<keyword>: <value>
```

The *Modelname* (optional) keyword, containing only alphanumeric characters (or `_`), describes the model filename (typically used when the model is exported via the PySCeS interface module) while the *Description* keyword is a (short) single line model description.

```
Modelname: rohwer_sucrose1  
Description: Sucrose metabolism in sugar cane (Johann M. Rohwer)
```

Two keywords are available for use (optional) with models that have one or more compartments defined. Both take a boolean (True/False) as their value:

- *Species\_In\_Conc* specifies whether the species symbols used in the rate equations represent a concentration (True, default) or an amount (False).
- *Output\_In\_Conc* tells PySCeS to output the results of numerical operations in concentrations (True, default) or in amounts (False).

```
Species_In_Conc: True
Output_In_Conc: False
```

More information on the effect these keywords have on the analysis of a model can be found in the PySCeS Reference Manual.

### 15.3 Global unit definition

PySCeS 0.7 supports the (optional) definition of a set of global units. In doing so we have chosen to follow the general approach used in the Systems Biology Modelling Language (SBML L2V3) specification. The general definition of a PySCeS unit is: ``<UnitType>: <kind>, <multiplier>, <scale>, <exponent>`` where *kind* is a string describing the base unit (for SBML compatibility this should be an SI unit) e.g. mole, litre, second or metre. The base unit is modified by the multiplier, scale and index using the following relationship:  $\langle multiplier \rangle * (\langle kind \rangle * 10^{*\langle scale \rangle})^{*\langle index \rangle}$ . The default unit definitions are:

```
UnitSubstance: mole, 1, 0, 1
UnitVolume: litre, 1, 0, 1
UnitTime: second, 1, 0, 1
UnitLength: metre, 1, 0, 1
UnitArea: metre, 1, 0, 2
```

Please note that defining these values does not affect the numerical analysis of the model in any way.

### 15.4 Symbol names and comments

Symbol names (i.e. reaction, species, compartment, function, rule and parameter names etc.) must start with either an underscore or letter and be followed by any combination of alphanumeric characters or an underscore. Like all other elements of the input file names are case sensitive:

```
R1
_subA
par1b
ext_1
```

Explicit access to the “current” time in a time simulation is provided by the special symbol `_TIME_`. This is useful in the definition of events and rules (see chapter on advanced model construction for more details).

Comments can be placed anywhere in the input file in one of two ways, as single line comment starting with a `#` or as a multi-line triple quoted comment `"""<comment>"""`:

```
# everything after this is ignored
```

```
"""
```

```
This is a comment
spread over a
few lines.
```

```
"""
```

## 15.5 Compartment definition

By default (as is the case in all PySCeS versions < 0.7) PySCeS assumes that the model exists in a single unit volume compartment. In this case it is **not** necessary to define a compartment and the ODE’s therefore describe changes in concentration per time. However, if a compartment is defined, PySCeS assumes that the ODE’s describe changes in substance amount per time. Doing this affects how the model is defined in the input file (especially with respect to the definitions of rate equations and species) and the user is **strongly** advised to read the Users Guide before building models in this way. The compartment definition is as follows  
 Compartment: <name>, <size>, <dimensions>, where <name> is the unique compartment id, <size> is the size of the compartment (i.e. length, volume or area) defined by the number of <dimensions> (e.g. 1,2,3):

```
Compartment: Cell, 2.0, 3
Compartment: Memb, 1.0, 2
```

## 15.6 Function definitions

A new addition to the PySCeS MDL is the ability to define SBML styled functions. Simply put these are code substitutions that can be used in rate equation definitions to, for example, simplify the kinetic law. The general syntax for a function is Function: <name>, <args> {<formula>} where <name> is the unique function id, <arglist> is one or more comma separated function arguments. The <formula> field, enclosed in curly brackets, may only make use of arguments listed in the <arglist> and therefore **cannot** reference model attributes directly. If this functionality is required a forcing function (assignment rule) may be what you are looking for.

```
Function: rmm_num, Vf, s, p, Keq {
Vf*(s - p/Keq)
}
```

```
Function: rmm_den, s, p, Ks, Kp {  
s + Ks*(1.0 + p/Kp)  
}
```

The syntax for function definitions has been adapted from Frank Bergmann and Herbert Sauro's "Human Readable Model Definition Language" (Draft 1).

## 15.7 Defining fixed species

Boundary species, also known as fixed or external species, are a special class of parameter used when modelling biological systems. The PySCeS MDL fixed species are declared on a single line as `FIX: <fixedlist>`. The `<fixedlist>` is a space separated list of symbol names which should be initialised like any other species or parameter:

```
FIX: Fru_ex Glc_ex ATP ADP UDP phos glycolysis Suc_vac
```

If no fixed species are present in the model then this declaration should be omitted entirely.

## 15.8 Reaction stoichiometry and rate equations

The reaction stoichiometry and rate equation are defined together as a single reaction step. Each step in the system is defined as having a name (identifier), a stoichiometry (substrates are converted to products) and rate equation (the catalytic activity, described in terms of species and parameters). All reaction definitions should be separated by an empty line. The general format of a reaction in a model with no compartments is:

```
<name>:  
    <stoichiometry>  
    <rate equation>
```

The `<name>` argument follows the syntax as discussed in a previous section, however, when more than one compartment has been defined it is important to locate the reaction in its specific compartment. This is done using the `@` operator:

```
<name>@<compartment>:  
    <stoichiometry>  
    <rate equation>
```

Where `<compartment>` is a valid compartment name. In either case this then followed either directly (or on the next line) by the reaction stoichiometry.

Each `<stoichiometry>` argument is defined in terms of reaction substrates, appearing on the left hand side and products on the right hand side of an identifier which labels the reaction as either reversible (=) or irreversible (>). If required each reagent's stoichiometric coefficient (PySCeS

accepts both integer and floating point) should be included in curly braces `{}` immediately preceding the reagent name. If these are omitted a coefficient of one is assumed:

```
{2.0}Hex_P = Suc6P + UDP # reversible reaction
Fru_ex > Fru           # irreversible reaction
species_5 > $pool      # a reaction to a sink
```

The PySCeS MDL also allows the use of the `$pool` token that represents a placeholder reagent for reactions that have no net substrate or product. Reversibility of a reaction is only used when exporting the model to other formats (such as SBML) and in the calculation of elementary modes. It does not affect the numerical evaluation of the rate equations in any way.

Central to any reaction definition is the *<rate equation>* (SBML kinetic law). This should be written as valid Python expression and may fall across more than one line. Standard Python operators `+` `-` `*` `/` `**` are supported (note the Python power e.g.  $2^4$  is written as `2**4`). There is no shorthand for multiplication with a bracket so  $-2(a+b)^h$  would be written as `-2*(a+b)**h` and normal operator precedence applies:

<code>+, -</code>	addition, subtraction
<code>*, /</code>	multiplication, division
<code>+x, -x</code>	positive, negative
<code>**</code>	exponentiation

Operator precedence increase from top to bottom and left to right (adapted from the Python Reference Manual).

The PySCeS MDL parser has been developed to parse and translate different styles of infix into Python/NumPy based expressions, the following functions are supported in any mathematical expression:

- `log`, `log10`, `ln`, `abs`
- `pow`, `exp`, `root`, `sqrt`
- `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`
- `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctanh`
- `floor`, `ceil`, `ceiling`, `piecewise`
- `notanumber`, `pi`, `infinity`, `exponentiale`

Logical operators are supported in rules, events etc but *not* in rate equation definitions. The PySCeS parser understands Python infix as well as libSBML and NumPy prefix notation.

- `and` or `xor` or `not`
- `>` `gt(x,y)` `greater(x,y)`
- `<` `lt(x,y)` `less(x,y)`
- `>=` `ge(x,y)` `geq(x,y)` `greater_equal(x,y)`

- `<= le(x,y) leq(x,y) less_equal(x,y)`
- `== eq(x,y) equal(x,y)`
- `!= neq(x,y) not_equal(x,y)`

Note that currently the MathML *delay* and *factorial* functions are not supported. Delay is handled by simply removing it from any expression, e.g. *delay(f(x), delay)* would be parsed as *f(x)*. Support for *piecewise* has been recently added to PySCeS and will be discussed in the *advanced features* section.

A reaction definition when no compartments are defined:

```
R5: Fru + ATP = Hex_P + ADP
    Fru/Ki5_Fru) * (Fru/Km5_Fru) * (ATP/Km5_ATP) / (1 +
    Vmax5 / (1 + Fru/Ki5_Fru) * (Fru/Km5_Fru) * (ATP/Km5_ATP) / (1 +
    Fru/Km5_Fru + ATP/Km5_ATP + Fru*ATP / (Km5_Fru*Km5_ATP) +
    ADP/Ki5_ADP)
```

and using the previously defined functions:

```
R6:
    A = B
    rmm_num(V2, A, B, Keq2) / rmm_den(A, B, K2A, K2B)
```

When compartments are defined note how now the reaction is now given a location and that because the ODE's formed from these reactions must be in changes in substance per time the rate equation is multiplied by its compartment size. In this particular example the species symbols represent concentrations (*Species\_In\_Conc: True*):

```
R1@Cell:
    s1 = s2
    Cell * (Vf1 * (s1 - s2/Keq1) / (s1 + KS1 * (1 + s2/KP1)))
```

If *Species\_In\_Conc: True* the location of the species is defined when it is initialised and will be explained later in this manual. The following example shows the species symbols explicitly defined as amounts (*Species\_In\_Conc: False*):

```
R4@Memb: s3 = s4
    Memb * (Vf4 * ((s3/Memb) - (s4/Cell) / Keq4) / ((s3/Memb)
    + KS4 * (1 + (s4/Cell) / KP4)))
```

Please note that at this time we are not certain if this form of rate equation is translatable into valid SBML in a way that is interoperable with other software.

## 15.9 Species and parameter initialisation

The general form of any species (fixed, free) and parameter is simply:

```
property = value
```

Initialisations can be written in any order anywhere in the input file but for human readability purposes these are usually placed after the reaction that uses them or grouped at the end of the input file. Both decimal and scientific notation is allowed with the following provisions that neither floating point (*1.* ) nor scientific shorthand (*1.e-3*) syntax should be used, instead use the full form (*1.0e-3*), (*0.001*) or (*1.0*).

Variable or free species are initialised differently depending on whether compartments are present in the model. While in essence the variables are set by the system parameters the

Although the variable species concentrations are determined by the parameters of the system, their initial values are used in various places, calculating total moiety concentrations (if present), time simulation initial values (e.g. time=zero) and as initial guesses for the steady-state algorithms. If an empty initial species pool is required it is not recommended to initialise these values to zero (in order to prevent potential divide-by-zero errors) but rather to a small value (e.g.  $10^{-8}$ ).

For a model with no compartments these initial values assumed to be concentrations:

```
NADH = 0.001
ATP   = 2.3e-3
sucrose = 1
```

In a model with compartments it is expected that the species are located in a compartment (even if *Species\_In\_Conc: False*) this is done using the @ symbol:

```
s1@Memb = 0.01
s2@Cell = 2.0e-4
```

A word of warning, the user is responsible for making sure that the units of the initialised species match those of the model. Please keep in mind that **all** species (and anything that depends on them) is defined in terms of the *Species\_In\_Conc* keyword. For example, if the preceding initialisations were for *R1* (see Reaction section) then they would be concentrations (as *Species\_In\_Conc: True*). However, in the next example, we are initialising species for *R4* and they are therefore in amounts (*Species\_In\_Conc: False*):

```
s3@Memb = 1.0
s4@Cell = 2.0
```

Fixed species are defined in a similar way and although technically a parameter, they should be given a location in compartmental models:

```
# InitExt
X0 = 10.0
X4@Cell = 1.0
```

However, fixed species are true parameters in the sense that their associated compartment size does not affect their value when it changes size. If compartment size dependent behaviour is required an assignment or rate rule should be considered.

Finally, the parameters should be initialised. PySCeS checks if a parameter is defined that is not present in the rate equations and if such parameter initialisations are detected a harmless warning is generated. If, on the other hand, an uninitialised parameter is detected a warning is generated and a value of 1.0 assigned:

```
# InitPar
Vf2 = 10.0
Ks4 = 1.0
```

# Advanced model construction

## 16.1 Assignment rules

Assignment rules or forcing functions are used to set the value of a model attribute before the ODE's are evaluated. This model attribute can either be a parameter used in the rate equations (this is traditionally used to describe an equilibrium block) a compartment or an arbitrary parameter (commonly used to define some sort of tracking function). Assignment rules can access other model attributes directly and have the generic form `!F <par> = <formula>`. Where `<par>` is the parameter assigned the result of `<formula>`. Assignment rules can be defined anywhere in the input file:

```
!F S_V_Ratio = Mem_Area/Vcyt
!F sigma_test = sigma_P*Pmem + sigma_L*Lmem
```

These rules would set the value of `<par>` which whose value can be followed with using the simulation and steady state `extra_data` functionality.

## 16.2 Rate rules

PySCeS now includes support for rate rules which are essentially directly encoded ODE's which are evaluated after the ODE's defined by the model stoichiometry and rate equations. Unlike the SBML rate rule, PySCeS allows one to access a reaction symbol in the rate rules (this is automatically expanded when the model is exported to SBML). The general form of a rate rule is `RateRule: <par> {<function>}`. Where `<name>` is the model attribute (e.g. compartment or parameter) whose rate of change is described by the `<formula>`. It may also be defined anywhere in the input file:

```
RateRule: Mem_Area {
  (sigma_P)*(Mem_Area*k4*(P)) + (sigma_L)*(Mem_Area*k5*(L))
}

RateRule: Vcyt { (1.0/Co)*(R1()+ (1-m1)*R2()+ (1-m2)*R3()-R4()-R5()) }
```

Remember to initialise any new parameters used in the rate rules.

## 16.3 Events

Time dependant events may now be defined whose definition follows the event framework described in the SBML L2V1 specification. The general form of an event is *Event*: `<name>`, `<trigger>`, `<delay>` { `<assignments>` }. As can be seen an event consists of essentially three parts, a conditional `<trigger>`, a set of one or more `<assignments>` and a `<delay>` between when the trigger is fired (and the assignments are evaluated) and the eventual assignment to the model. Assignments have the general form `<par> = <formula>`. Events have access to the “current” simulation time using the `_TIME_` symbol:

```
Event: event1, _TIME_ > 10 and A > 150.0, 0 {
V1 = V1*vfact
V2 = V2*vfact
}
```

The following event illustrates the use of a delay of ten time units as well as the prefix notation (used by libSBML) for the trigger (PySCeS understands both notations):

```
Event: event2, geq(_TIME_, 15.0), 10 {
V3 = V3*vfact2
}
```

*Note:* in order for PySCeS to handle events it is necessary to have the PySundials installed

## 16.4 Piecewise

Although technically an operator piecewise functions are sufficiently complicated to warrant their own section. A piecewise operator is essentially an *if, elif, ..., else* logical operator that can be used to conditionally “set” the value of some model attribute. Currently piecewise is supported in rule constructs and has not been tested directly in rate equation definitions. The piecewise function’s most basic incarnation is *piecewise*(`<val1>`, `<cond>`, `<val2>`) which is evaluated as:

```
if <cond>:
    return <val1>
else:
    return <val2>
```

alternatively, *piecewise*(`<val1>`, `<cond1>`, `<val2>`, `<cond2>`, `<val3>`, `<cond3>`):

```
if <cond1>:
    return <val1>
elif <cond2>:
    return <val1>
elif <cond3>:
    return <val3>
```

or `piecewise(<val1>, <cond1>, <val2>, <cond2>, <val3>, <cond3>, <val4>):`

```
if <cond1>:  
    return <val1>  
elif <cond2>:  
    return <val2>  
elif <cond3>:  
    return <val3>  
else:  
    return <val4>
```

can also be used. A “real-life” example of an assignment rule with a piecewise function:

```
!F Ca2plus=piecewise(0.1, lt(_TIME_, 60), 0.1, gt(_TIME_, 66.0115), 1)
```

In principle there is no limit on the amount of conditional statements present in a piecewise function, the condition can be a compound statements *a or b and c* and may include the `_TIME_` symbol.



## Example PySCeS input files

### 17.1 Basic model definition

PySCeS test model *pysces\_test\_linear1.psc*:

```
FIX: x0 x3

R1: x0 = s0
    k1*x0 - k2*s0

R2: s0 = s1
    k3*s0 - k4*s1

R3: s1 = s2
    k5*s1 - k6*s2

R4: s2 = x3
    k7*s2 - k8*x3

# InitExt
x0 = 10.0
x3 = 1.0
# InitPar
k1 = 10.0
k2 = 1.0
k3 = 5.0
k4 = 1.0
k5 = 3.0
k6 = 1.0
k7 = 2.0
k8 = 1.0
# InitVar
s0 = 1.0
s1 = 1.0
s2 = 1.0
```

## 17.2 Advanced example

This model includes the use of *Compartments*, *KeyWords*, *Units* and *Rules*:

```
Modelname: MWC_wholecell2c
Description: Surovtsev whole cell model using J-HS Hofmeyr's framework

Species_In_Conc: True
Output_In_Conc: True

# Global unit definition
UnitVolume: litre, 1.0, -3, 1
UnitSubstance: mole, 1.0, -6, 1
UnitTime: second, 60, 0, 1

# Compartment definition
Compartment: Vcyt, 1.0, 3
Compartment: Vout, 1.0, 3
Compartment: Mem_Area, 5.15898, 2

FIX: N

R1@Mem_Area: N = M
    Mem_Area*k1*(Pmem)*(N/Vout)

R2@Vcyt: {244}M = P # m1
    Vcyt*k2*(M)

R3@Vcyt: {42}M = L # m2
    Vcyt*k3*(M)*(P)**2

R4@Mem_Area: P = Pmem
    Mem_Area*k4*(P)

R5@Mem_Area: L = Lmem
    Mem_Area*k5*(L)

# Rate rule definition
RateRule: Vcyt { (1.0/Co)*(R1()+ (1-m1)*R2()+ (1-m2)*R3()-R4()-R5()) }
RateRule: Mem_Area { (sigma_P)*R4() + (sigma_L)*R5() }

# Rate rule initialisation
Co = 3.07e5 # uM p_env/(R*T)
m1 = 244
m2 = 42
sigma_P = 0.00069714285714285711
sigma_L = 0.00012

# Assignment rule definition
!F S_V_Ratio = Mem_Area/Vcyt
```

```
!F Mconc = (M)/M_init
!F Lconc = (L)/L_init
!F Pconc = (P)/P_init

# Assignment rule initialisations
M_init = 199693.0
L_init = 102004
P_init = 5303
Mconc = 1.0
Lconc = 1.0
Pconc = 1.0

# Species initialisations
N@Vout = 3.07e5
Pmem@Mem_Area = 37.38415
Lmem@Mem_Area = 8291.2350678770199
M@Vcyt = 199693.0
L@Vcyt = 102004
P@Vcyt = 5303

# Parameter initialisations
k1 = 0.00089709
k2 = 0.000182027
k3 = 1.7539e-010
k4 = 5.0072346e-005
k5 = 0.000574507164

"""
Simulate this model to 200 for maximum happiness and
watch the surface to volume ratio and scaled concentrations.
"""
```

This example illustrates almost all the new features included in the PySCeS MDL. Although it may be slightly more complicated than the basic model described above it is still, by our definition, certainly human readable.



## **Part X**

### **PySCeS Module documentation**



## PyscesPlot2

PyscesPlot2 is a new graphics subsystem for PySCeS which will include a Unified Plotting Interface which can take advantage of different plotting backends via a common user interface.

**class FIFOBuffer** (*size*)

Simple fixed size FIFO buffer.

**add** (*x*)

**get** ()

**class GnuPlotUPI** (*work\_dir=None, gnuplot\_dir=None*)

PySCeS/GnuPlot is reborn, leaner and meaner than ever before. This class enables plotting with GnuPlot via a subprocess link:

- *work\_dir* optional argument setting directory for dat file(s)
- *gnuplot\_dir* optional argument specifying the location of pgnuplot.exe (win32) or gnuplot

GnuPlot backend to the Unified Plotting Interface.

**export** (*name, directory=None, type='png'*)

Export the current plot as a <format> file.

- *filename* the filename
- *directory* optional (default = current working directory)
- *type* the file format (default='png').

Currently only PNG is guaranteed to be available in all interfaces.

**g\_file\_write\_array** (*arr, dfmt=None*)

Write a normal (2D) dataset to temp file. Dumps the array to file using the format:

- *arr* the array (*r*>0, *c*>1)
- *fmt* default '%.8e'

**g\_file\_write\_array3D** (*arr, yaxis=1, dfmt=None*)

Write a GnuPlot format 3D dataset. The *yaxis* argument specifies the column that should be used to split the dataset into GnuPlot slices.

- *arr* the array (*r*>1, *c*>2)
- *fmt* default '%.8e'
- *yaxis* default 1

**g\_pause** ()

A small pause defined by *self.PAUSE\_TIME* (multiplied by 2 when in multiplot).

**g\_write** (*cmd*)

Write a command to the GnuPlot interpreter

- cmd* the GnuPlot command

**plot** (*data, x, y, title=""*, *format='w l'*)

Plot a single line data[*y*] vs data[*x*] where:

- data* the data array
- x* x column index
- y* y column index
- title* is the line key
- format* is the GnuPlot format string (default='w l')

Format can also be the *CommonStyle* 'lines' or 'points'.

**plotLines** (*data, x, y=, [], titles=, [], formats=, ['w l']*)

Plot a multiple lines data[*y1, y2, ...*] vs data[*x*] where:

- data* the data array
- x* x column index
- y* is a list of line indexes, if empty all of *y* not including *x* is plotted
- titles* is a list of line keys if empty Line1, Line2, Line3 is used
- formats* is a list (per line) of GnuPlot format strings (default='w l').

If *formats* only contains a single item, this format is used for all lines and can also be the *CommonStyle* 'lines' or 'points'.

**replot** ()

Replot the current GnuPlot plot

**replotAndWait** (*seconds=0.5*)

Replot the current GnuPlot plot and wait default (*seconds = 0.5*) or until enter is pressed (*seconds = -1*)

**save** (*name, directory=None, dfmt=None*)

Save the last plot as a GnuPlot file *name.plt* which references *name.dat*.

- name* the name of the GnuPlot plt and and datafile
- directory* (optional) the directory to use (defaults to working directory)
- dfmt* is ignored and uses the value of *self.DATF\_FORMAT*

**set** (*key, value=""*)

Send *set <key>* or optionally *set <key> <value>* to GnuPlot.

**setAxisLabel** (*axis, label=""*)

Set the axis label:

- axis* = x, y, z, xy, xz, yz, zyx
- label* = string (default='')

Called with only the axis argument clears the axis label.

**setDataFileNumberFormat** (*format='%0.8e'*)

Sets the format string for data written to file

- format* format string (default='%.8e')

**setGraphTitle** (*title='PySCeS Plot'*)

Set the graph title, unset if title argument is None

- title* (string, default='PySCeS Plot') the graph title

**setGrid** (*value*)

Display or remove graph grid.

- value* (boolean) True (on) or False (off)

**setKey** (*value=False*)

Enable or disable the current plot key, no arguments removes key.

- value* boolean (default = False)

**setLogScale** (*axis*)

Set axis to logscale where:

- axis* = x, y, z, xy, xz, yz, zyx

**setMultiplot** ()

Begin a multiplot session

**setNoLogScale** (*axis*)

Set axis to a linear scale where:

- axis* = x, y, z, xy, xz, yz, zyx

**setOrigin** (*xpos=0, ypos=0*)

Set the origin (lower left corner) of the next plot. Uses GnuPlot screen coordinates. If no arguments are supplied reset origin to 0,0.

- xpos* of next plot (default = 0)

- ypos* of next plot (default = 0)

**setRange** (*axis, min=None, max=None*)

Set axis range where:

- axis* = x, y, z, xy, xz, yz, zyx

- min* = range(s) lower bound (default=None) autoscale

- max* = range(s) upper bound (default=None) autoscale

If only the *axis* argument is provided, GnuPlot will autoscale the ranges to the data.

**setSize** (*width=1.0, height=1.0*)

Set the size of the next plot relative to the GnuPlot canvas (e.g. screen) size which is defined to be 1. For example if `width = height = 0.5` the plot is 1/4 the size of the viewable canvas. If no arguments are supplied reset size to 1,1.

- width* of next plot (default = 1.0)

- height* of next plot (default = 1.0)

**setSizeAndOrigin** (*width=1, height=1, xpos=0, ypos=0*)

Set the size and origin of the next plot. If no arguments are supplied, reset the size to 1,1 and origin to 0.0

- width* of next plot (default = 1.0)

- height* of next plot (default = 1.0)

- xpos* of next plot (default = 0)

- *ypos* of next plot (default = 0)

**setTerminal** (*name, options=""*)

Sets the terminal, gnuplot: set terminal *name options*

**splot** (*data, x, y, z, title="", format='w l'*)

Plot a surface data[z] vs data[y] vs data[x] where:

- *data* the data array
- *x* x column index
- *y* y column index
- *z* z column index
- *title* is the surface key
- *format* is the Gnuplot format string (default='w l')

Format can also be the *CommonStyle* 'lines' or 'points'.

**splotSurfaces** (*data, x, y, z=, [], titles=, [], formats=, ['w l']*)

Plot data[z1, z2, ] vs data[y] vs data[x] where:

- *data* the data array
- *x* x column index
- *y* y column index
- *z* list of z column indexes, if empty all of z not including x, y are plotted
- *titles* is a list of surface keys, if empty Surf1, Surf2, Surf3 is used
- *formats* is a list (per line) of Gnuplot format strings (default='w l').

If *formats* only contains a single item, this format is used for all surface and can also be the *CommonStyle* 'lines' or 'points'.

**unset** (*key, value=""*)

Send *unset <key>* or optionally *unset <key> <value>* to Gnuplot.

**unsetMultiplot** ()

End a multiplot session.

**class MatplotlibUPI** (*work\_dir=None*)

Refactored Matplotlib backend to the Unified Plotting Interface

- *work\_dir* (optional) working directory

**closeAll** ()

Close all open matplotlib figures.

**export** (*name, directory=None, type='png'*)

Export the current plot as a <format> file.

- *filename* the filename
- *directory* optional (default = current working directory)
- *type* the file format (default='png').

Currently only PNG is guaranteed to be available in all interfaces.

**hold** (*hold=False*)

Enable plot holding where each new graph is plotted on top of the previous one.

- *hold* boolean (default = False)

**plot** (*data*, *x*, *y*, *title*=", *format*='-')

Plot a single line *data*[*y*] vs *data*[*x*] where:

- *data* the data array
- *x* x column index
- *y* y column index
- *title* is the line key
- *format* is the Matplotlib format string (default='-')

Format can also be the *CommonStyle* 'lines' or 'points'.

**plotLines** (*data*, *x*, *y*=, [], *titles*=, [], *formats*=, ['-'])

Plot a multiple lines *data*[*y*<sub>1</sub>, *y*<sub>2</sub>, ] vs *data*[*x*] where:

- *data* the data array
- *x* x column index
- *y* is a list of line indexes
- *titles* is a list of line keys
- *formats* is a list (per line) of Matplotlib format strings.

If *formats* only contains a single item, this format is used for all lines and can also be the *CommonStyle* 'lines' or 'points'.

**save** (*name*, *directory*=None, *dfmt*='%0.8e')

Save the plot data to

- *filename* the filename
- *directory* optional (default = current working directory)
- *dfmt* the data format string (default='%0.8e')

**setAxisLabel** (*axis*, *label*=")

Set the axis label:

- *axis* = x, y, z, xy, xz, yz, zyx
- *label* = string (default='')

Called with only the axis argument clears the axis label.

**setGraphTitle** (*title*='PySCeS Plot')

Set the graph title, unset if *title*=None

- *title* (string, default='PySCeS Plot') the graph title

**setGrid** (*value*)

Display or remove graph grid.

- *value* (boolean) True (on) or False (off)

**setKey** (*value*=False)

Enable or disable the current plot key, no arguments removes key.

- *value* boolean (default = False)

**setLineWidth** (*width*=1)

Sets the line width for current axis

- *width* the line width

**setLogScale** (*axis*)

Set axis to logscale where:

- axis* = x, y, z, xy, xz, yz, zyx

**setNoLogScale** (*axis*)

Set axis to a linear scale where:

- axis* = x, y, z, xy, xz, yz, zyx

**setRange** (*axis*, *min=None*, *max=None*)

Set axis range where

- axis* = x, y, z, xy, xz, yz, zyx

- min* = range(s) lower bound (default=None) autoscale

- max* = range(s) upper bound (default=None) autoscale

**class PlotBase** ()

Abstract class defining the Unified Plotting Interface methods. These methods should be overridden and the class extended by interface specific subclasses.

**axisInputStringToList** (*input*)

Extracts axis information from a string input, returns a boolean triple representing (x=True/False, y=True/False, z=True/False).

- input* the input string

**export** (*name*, *directory=None*, *type='png'*)

Export the current plot as a <format> file.

- filename* the filename

- directory* optional (default = current working directory)

- type* the file format (default='png').

Currently only PNG is guaranteed to be available in all interfaces.

**plot** (*data*, *x*, *y*, *title=""*, *format=""*)

Plot a single line data[y] vs data[x] where:

- data* the data array

- x* x column index

- y* y column index

- title* is the line key

- format* is the XXX format string (default='')

Format can also be the *CommonStyle* 'lines' or 'points'

**plotLines** (*data*, *x*, *y=*, [], *titles=*, [], *formats=*, [])

Plot a multiple lines data[y1, y2, ] vs data[x] where:

- data* the data array

- x* x column index

- y* is a list of line indexes, if empty all of y not including x is plotted

- titles* is a list of line keys, if empty Line1,Line2,Line3 is used

- formats* is a list (per line) of XXX format strings.

If *formats* only contains a single item, this format is used for all lines and can also be the *CommonStyle* 'lines' or 'points'.

**save** (*name*, *directory=None*, *dfmt='%0.8e'*)

Save the plot data and (optionally) XXX format file

- *filename* the filename
- *directory* optional (default = current working directory)
- *dfmt* the data format string (default='%.8e')

**setAxisLabel** (*axis*, *label*=")

Set the axis label:

- *axis* = x, y, z, xy, xz, yz, zyx
- *label* = string (default='')

Called with only the axis argument clears the axis label.

**setGraphTitle** (*title*= 'PySCeS Plot')

Set the graph title, unset if title=None

- *title* (string, default='PySCeS Plot') the graph title

**setGrid** (*value*)

Display or remove graph grid.

- *value* (boolean) True (on) or False (off)

**setKey** (*value*=False)

Enable or disable the current plot key, no arguments removes key.

- *value* boolean (default = False)

**setLogScale** (*axis*)

Set axis to logscale where:

- *axis* = x, y, z, xy, xz, yz, zyx

**setNoLogScale** (*axis*)

Set axis to a linear scale where:

- *axis* = x, y, z, xy, xz, yz, zyx

**setRange** (*axis*, *min*=None, *max*=None)

Set axis range where

- *axis* = x, y, z, xy, xz, yz, zyx
- *min* = range(s) lower bound (default=None) autoscale
- *max* = range(s) upper bound (default=None) autoscale

**plot** (*data*, *x*, *y*, *z*, *title*="", *format*="")

Plot a surface data[z] vs data[y] vs data[x] where:

- *data* the data array
- *x* x column index
- *y* y column index
- *z* z column index
- *title* is the surface key
- *format* is the XXX format string (default='')

Format can also be the *CommonStyle* 'lines' or 'points'.

**plotSurfaces** (*data*, *x*, *y*, *z*=[, ], *titles*=[, ], *formats*=[, ])

Plot data[z1, z2, ] vs data[y] vs data[x] where:

- *data* the data array

- *x* x column index
- *y* y column index
- *z* list of z column indexes, if empty all of z not including x, y are plotted
- *titles* is a list of surface keys, if empty Surf1, Surf2, Surf3 is used
- *formats* is a list (per line) of XXX format strings (default='').

If *formats* only contains a single item, this format is used for all surfaces and can also be the *CommonStyle* 'lines' or 'points'.

**wait** (*seconds=3*)

Wait *seconds* (default = 3) or until enter is pressed (seconds = -1)

**class PyscesUPI** ()

This is the frontend to the PySCeS Unified Plotting Interface (pysces.plt.\*) that allows one to specify which backend should be used to plot when a UPI method is called. More than one interface can be active at the same time and so far the Matplotlib and GnuPlot backends are available for use.

This is an experiment which must be refactored into a more general way of doing things. Basically, I want an instance of the abstract plotting class which will plot to one, any or all currently available backends. If anybody has an idea how I can generate this class automatically please let me know ;-)

**closeAll** ()

Close all active Matplotlib figures

**export** (*name, directory=None, type='png'*)

Export the current plot as a <format> file.

- *filename* the filename
- *directory* optional (default = current working directory)
- *type* the file format (default='png').

Currently only PNG is guaranteed to be available in all interfaces.

**p\_activateInterface** (*interface*)

Activate an interface that has been set with **p\_setInterface()** but deactivated with **p\_deactivateInterface**

- *interface* one of ['matplotlib', 'gnuplot']

**p\_deactivateInterface** (*interface*)

Deactivate the interface. This does not delete the interface and it is possible to reactivate the deactivated interface with **p\_activateInterface**.

- *interface* one of ['matplotlib', 'gnuplot']

**p\_setInterface** (*name, instance*)

Add an interface to the backend selector

- *name* the interface name currently one of ['matplotlib', 'gnuplot']
- *instance* an instance of a PlotBase derived (UPI) interface

**plot** (*data, x, y, title="", format=""*)

Plot a single line data[y] vs data[x] where:

- *data* the data array
- *x* x column index

- *y* y column index
- *title* is the line key
- *format* is the backend format string (default='')

**plotLines** (*data*, *x*, *y*=, [], *titles*=, [], *formats*=, [])

Plot a multiple lines data[*y*1, *y*2, ] vs data[*x*] where:

- *data* the data array
- *x* x column index
- *y* is a list of line indexes, if empty all of *y* not including *x* is plotted
- *titles* is a list of line keys, if empty Line1,Line2,Line3 is used
- *formats* is a list (per line) of XXX format strings.

If *formats* only contains a single item, this format is used for all lines.

**replot** ()

Replot the current figure for all active interfaces

**save** (*name*, *directory*=None, *dfmt*='%.8e')

Save the plot data and (optionally) XXX format file

- *filename* the filename
- *directory* optional (default = current working directory)
- *dfmt* the data format string (default='%.8e')

**setAxisLabel** (*axis*, *label*="")

Set the axis label:

- *axis* = x, y, z, xy, xz, yz, zyx
- *label* = string (default=None)

Called with only the axis argument clears the axis label.

**setGraphTitle** (*title*='PySCeS Plot')

Set the graph title, unset if title=None

- *title* (string, default='PySCeS Plot') the graph title

**setGrid** (*value*)

Display or remove graph grid.

- *value* (boolean) True (on) or False (off)

**setKey** (*value*=False)

Enable or disable the current plot key, no arguments removes key.

- *value* boolean (default = False)

**setLogScale** (*axis*)

Set axis to logscale where:

- *axis* = x, y, z, xy, xz, yz, zyx

**setNoLogScale** (*axis*)

Set axis to a linear scale where:

- *axis* = x, y, z, xy, xz, yz, zyx

**setRange** (*axis*, *min*=None, *max*=None)

Set axis range where

- *axis* = x, y, z, xy, xz, yz, zyx
- *min* = range(s) lower bound (default=None) autoscale
- *max* = range(s) upper bound (default=None) autoscale

**splot** (*data*, *x*, *y*, *z*, *title*=", *format*="")

Plot a surface data[*z*] vs data[*y*] vs data[*x*] where:

- *data* the data array
- *x* x column index
- *y* y column index
- *z* z column index
- *title* is the surface key
- *format* is the XXX format string (default='')

**splotSurfaces** (*data*, *x*, *y*, *z*=, [], *titles*=, [], *formats*=, [""])

Plot data[*z*<sub>1</sub>, *z*<sub>2</sub>, ] vs data[*y*] vs data[*x*] where:

- *data* the data array
- *x* x column index
- *y* y column index
- *z* list of z column indexes, if empty all of z not including x, y are plotted
- *titles* is a list of surface keys, if empty Surf1, Surf2, Surf3 is used
- *formats* is a list (per line) of XXX format strings (default='').

If *formats* only contains a single item, this format is used for all surfaces.

## PyscesModel

This module contains the core PySCeS classes which create the model and associated data objects

**class BagOfStuff** (*matrix, row, col*)

A collection of attributes defined by row and column lists used by Response coefficients etc matrix is an array of values while row/col are lists of row column name strings

**get** (*attr1, attr2*)

Returns a single attribute “attr1\_attr2” or None

**list** ()

Return all attributes as a attr:val dictionary

**load** ()

**select** (*attr, search='a'*)

Return a dictionary of <attr>\_<name>, <name>\_<attr> : val or {} if none If attr exists as an index for both left and right attr then: search='a' : both left and right attributes (default) search='l' : left attributes only search='r' : right attributes

**class Event** (*name, mod*)

Event's have triggers and fire EventAssignments when required. Ported from Core2.

**reset** ()

**setAssignment** (*var, formula*)

**setTrigger** (*formula, delay=0.0*)

**class EventAssignment** (*name, mod*)

Event assignments are actions that are triggered by an event. Ported from Core2 to build an event handling framework fro PySCeS

**evaluateAssignment** ()

**setFormula** (*formula*)

**setVariable** (*var*)

**class Function** (*name, mod*)

Function class ported from Core2 to enable the use of functions in PySCeS.

**addFormula** (*formula*)

**setArg** (*var, value=None*)

**class IntegrationDataObj ( )**

This class is specifically designed to store the results of a time simulation It has methods for setting the Time, Labels, Species and Rate data and getting Time, Species and Rate (including time) arrays. However, of more use:

- **getOutput(\*args)** feed this method species/rate labels and it will return an array of [time, sp1, r1, ....]
- **getDataAtTime(time)** the data generated at time point “time”.
- **getDataInTimeInterval(time, bounds=None)** more intelligent version of the above returns an array of all data points where: time-bounds <= time <= time+bounds

**getAllSimData (lbls=False)**

Return all available data as time+species+rates+rules if lbls=True returns (array,labels) else just array

**getDataAtTime (time)**

Return all data generated at “time”

**getDataInTimeInterval (time, bounds=None)**

getDataInTimeInterval(time, bounds=None) returns an array of all data points where: time-bounds <= time <= time+bounds where bound defaults to stepsize

**getOutput (\*args)**

Old alias for getSimData() getOutput(\*args) feed this method species/rate labels and it will return an array of [time, sp1, r1, ....]

**getRates (lbls=False)**

return time+rate array

**getRules (lbls=False)**

Return time+rule array

**getSimData (\*args, \*\*kwargs)**

getSimData(\*args) feed this method species/rate labels and it will return an array of [time, sp1, r1, ....]

**getSpecies (lbls=False)**

return time+species array

**getTime (lbls=False)**

return the time vector

**getXData (lbls=False)**

Return time+xdata array

**setLabels (species=None, rates=None, rules=None)**

set the species, rate and rule label lists

**setRates (rates, lbls=None)**

set the rate array

**setRules (rules, lbls=None)**

Set the results of rate rules

**setSpecies (species, lbls=None)**

Set the species array

**setTime** (*time*, *lbl=None*)

Set the time vector

**setXData** (*xdata*, *lbls=None*)

Sets extra simulation data

**class NewCoreBase** ()

Core2 base class, needed here as we use Core2 derived classes in PySCes

**get** (*attr*)

Return an attribute whose name is str(attr)

**getName** ()

**setName** (*name*)

**class NumberBase** ()

Derived Core2 number class.

**getValue** ()

**setValue** (*v*)

**class PieceWise** (*pwd*, *mod*)

Generic piecewise class adapted from Core2 that generates a compiled Python code block that allows evaluation of arbitrary length piecewise functions. Piecewise statements should be defined in assignment rules as *piecewise*(*<Piece>*, *<Conditional>*, *<Other-Value>*) where there can be an arbitrary number of *<Piece>*, *<Conditional>* pairs.

- args* a dictionary of piecewise information generated by the InfixParser as Infix-Parser.piecewises

**class PysMod** (*File=None*, *dir=None*, *loader='file'*, *fString=None*)

This is the PySCeS Core class, a.k.a. pysces.model()

**CVODE** (*initial*)

CVODE(initial)

PySCeS interface to the CVODE integration algorithm.

Arguments: initial: vector containing initial species concentrations

**CVODE\_EVENTS** (*t*, *svec*, *eout*, *f\_data*)

**CVODE\_VPYTHON** (*s*)

Future VPython hook for CVODE

**EvalCC** ()

EvalCC()

Calculate the MCA control coefficients using the current steady-state solution.

mod.\_\_settings\_\_[“mca\_ccj\_upsymb”] = 1 attach the flux control coefficients to the model instance mod.\_\_settings\_\_[“mca\_ccs\_upsymb”] = 1 attach the concentration control coefficients to the model instance

Arguments: None

**EvalEigen** ()

EvalEigen()

Calculate the eigenvalues or vectors of the unscaled Jacobian matrix and thereby analyse the stability of a system

Arguments: None

**EvalEpar** (*input=None, input2=None*)

EvalEpar(input=None,input2=None)

Calculate reaction elasticities towards the parameters.

Both inputs (input1=species,input2=rates) should be valid (steady state for MCA) solutions and given in the correct order for them to be used. If either or both are missing the last state values are used automatically. Elasticities are scaled using input 1 and 2.

mod.\_\_settings\_\_[“elas\_epar\_upsymb”] = 1 attach individual elasticity symbols to model instance

Arguments:

input [default=None]: species concentration vector input2 [default=None]: reaction rate vector

**EvalEvar** (*input=None, input2=None*)

EvalEvar(input=None,input2=None)

Calculate reaction elasticities towards the variable species.

Both inputs (input1=species,input2=rates) should be valid (steady state for MCA) solutions and given in the correct order for them to be used. If either or both are missing the last state values are used automatically. Elasticities are scaled using input 1 and 2. mod.\_\_settings\_\_[“elas\_evar\_upsymb”] = 1 attach individual elasticity symbols to model instance

Arguments:

input [default=None]: species concentration vector input2 [default=None]: reaction rate vector

**EvalRC** ()

EvalRC()

Calculate the MCA response coefficients using the current steady-state solution.

Arguments: None

**FINTSLV** (*initial*)

FINTSLV(initial)

Forward integration steady-state solver. Finds a steady state when the maximum change in species concentration falls within a specified tolerance. Returns the steady-state solution and a error flag. Algorithm controls are available as mod.fintslv\_<control>

Arguments:

initial: vector of initial concentrations

**Fix\_S\_fullinput** (*s\_vec, amounts=True*)

Fix\_S\_fullinput(s\_vec)

Using the full concentration vector evaluate the dependent species

Arguments:

s\_vec: a full length concentration vector

**Fix\_S\_indinput** (*s\_vec, amounts=True*)

Fix\_S\_indinput(s\_vec, amounts=True) whether to use self.\_\_tvec\_a\_\_ (default) or self.\_\_tvec\_c\_\_

Given a vector of independent species evaluate and return a full concentration vector.

Arguments:

s\_vec: vector of independent species

**Fix\_Sim** (*metab, flux=0, par=0*)

**Deprecated**

**FluxGenSim** (*s*)

**Deprecated**

**Forcing\_Function** ()

Forcing\_Function()

User defined forcing function either defined in the PSC input file as !F or by overwriting this method. This method is evaluated prior to every rate equation evaluation.

Arguments: None

**HYBRD** (*initial*)

HYBRD(initial)

PySCeS interface to the HYBRD solver. Returns a steady-state solution and error flag. Good general purpose solver. Algorithm controls are available as mod.hybrd\_<control>

Arguments:

initial: vector of initial species concentrations

**InitialiseCompartments** ()

**InitialiseEvents** ()

**InitialiseFunctions** ()

**InitialiseInputFile** ()

InitialiseInputFile()

Parse the input file associated with the PySCeS model instance and assign the basic model attributes

Arguments: None

**InitialiseModel** ()

InitialiseModel()

Initialise and set up dynamic model attributes and methods using the model defined in the associated PSC file

Arguments: None

**InitialiseOldFunctions** ()

InitialiseOldFunctions()

Parse and initialise user defined functions specified by !T !U in the PSC input file

Arguments: None

**InitialiseRuleChecks** ()

**InitialiseRules** ()

**LSODA** (*initial*)

LSODA(initial)

PySCeS interface to the LSODA integration algorithm. Given a set of initial conditions LSODA returns an array of species concentrations and a status flag. LSODA controls are accessible as `mod.lsoda_<control>`

Arguments:

initial: vector containing initial species concentrations

**LoadFromFile** (*File=None, dir=None*)

`__init__`(File=None,dir=None)

Initialise a PySCeS model object with PSC file that can be found in optional directory. If a filename is not supplied the `pysces.model_dir` directory contents is displayed and the model name can be entered at the prompt (<ctrl>+C exits the loading process).

Arguments:

File [default=None]: the name of the PySCeS input file dir [default=`pysces.model_dir`]: the optional directory where the PSC file can be found

**LoadFromString** (*File=None, fString=None*)

Docstring required

**NLEQ2** (*initial*)

NLEQ2(initial)

PySCeS interface to the (optional) NLEQ2 algorithm. This is a powerful steady-state solver that can usually find a solution for when `HYBRD()` fails. Algorithm controls are available as: `mod.nleq2_<control>` Returns as steady-state solution and error flag.

Arguments:

initial: vector of initial species concentrations

**PITCON** (*scanpar, scanpar3d=None*)

PITCON(scanpar,scanpar3d=None)

PySCeS interface to the PITCON continuation algorithm. Single parameter continuation has been implemented as a “scan” with the continuation being initialised in `mod.pitcon_par_space`. The second argument does not affect the continuation but can be used to insert a third axis parameter into the results. Returns an array containing the results. Algorithm controls are available as `mod.pitcon_<control>`

Arguments:

scanpar: the model parameter to scan (x5) scanpar3d [default=None]: additional output parameter for 3D plots

**ParGenSim** ()

Deprecated

**ReloadInitFunc** ()

ReloadInitFunc()

Recompile and execute the user initialisations (!I) as defined in the PSC input file. and in `mod.__InitFuncs__`

Arguments: None

**ReloadUserFunc ()**

ReloadUserFunc()

Recompile and execute the user function (!U) from the input file.

Arguments: None

**ResetNumberFormat ()**

ResetNumberFormat()

Reset PySCeS default number format stored as mod.mode\_number format to %2.4e

Arguments: None

**ScaleKL (input, input2)**

ScaleKL(input,input2)

Scale the K and L matrices with current steady state (if either input1 or 2 == None) or user input.

Arguments:

input: vector of species concentrations input2: vector of reaction rates

**Scan1 (range1=, [], runUF=0)**

Scan1(range1=[],runUF=0)

Perform a single dimension parameter scan using the steady-state solvers. The parameter to be scanned is defined (as a model attribute "P") in mod.scan\_in while the required output is entered into the list mod.scan\_out. Results of a parameter scan can be easilly viewed with Scan1Plot().

mod.scan\_in - a model attribute written as in the input file (eg. P, Vmax1 etc)

mod.scan\_out - a list of required output ['A','T2', ...] mod.scan\_res - the results of

a parameter scan mod.\_\_settings\_\_[ "scan1\_mca\_mode" ] - force the scan algorithm to evaluate the elasticities (1) and control coefficients (2) (this should also be auto-detected by the Scan1 method).

Arguments:

range1 [default=[]]: a predefined range over which to scan. runUF [default=0]: run (1) the user defined function mod.User\_Function (!U) before evaluating the steady state.

**Scan1Plot (plot=, [], title=None, log=None, format='lines', filename=None)**Plot the results of a parameter scan generated with **Scan1()**

- *plot* if empty mod.scan\_out is used, otherwise any subset of mod.scan\_out (default=[])
- *filename* the filename of the PNG file (default=None, no export)
- *title* the plot title (default=None)
- *log* if None a linear axis is assumed otherwise one of ['x','xy','xyz'] (default=None)
- *format* the backend dependent line format (default='lines') or the *Common-Style* 'lines' or 'points'.

**Scan2D (p1, p2, output, log=False)**

Generate a 2 dimensional parameter scan using the steady-state solvers.

- *p1* is a list of [parameter1, start, end, points]
- *p2* is a list of [parameter2, start, end, points]

- output* the steady-state variable e.g. 'J\_R1' or 'A\_ss'
- log* scan using log ranges for both axes

**Scan2DPlot** (*title=None, log=None, format='lines', filename=None*)

Plot the results of a 2D scan generated with Scan2D

- filename* the filename of the PNG file (default=None, no export)
- title* the plot title (default=None)
- log* if None a linear axis is assumed otherwise one of ['x','xy','xyz'] (default=None)
- format* the backend dependent line format (default='lines') or the *Common-Style* 'lines' or 'points'.

**SerialDecode** (*filename*)

SerialDecode(filename)

Decode and return a serialised object saved with SerialEncode.

Arguments:

filename: the filename (.pscdat is assumed)

**SerialEncode** (*data, filename*)

SerialEncode(data,filename)

Serialise and save a Python object using a binary pickle to file. The serialised object is saved as <filename>.pscdat in the directory defined by mod.model\_serial.

Arguments:

data: pickleable Python object filename: the output filename

**SetLoud** ()

SetLoud()

Turn on as much solver reporting noise as possible:  
 mod.\_\_settings\_\_['hybrd\_mesg'] = 1 mod.\_\_settings\_\_['nleq2\_mesg'] = 1  
 mod.\_\_settings\_\_['lsoda\_mesg'] = 1 mod.\_\_settings\_\_['mode\_state\_mesg'] = 1  
 mod.\_\_settings\_\_['solver\_switch\_warning'] = True

Arguments: None

**SetQuiet** ()

SetQuiet()

Turn off as much solver reporting noise as possible:  
 mod.\_\_settings\_\_['hybrd\_mesg'] = 0 mod.\_\_settings\_\_['nleq2\_mesg'] = 0  
 mod.\_\_settings\_\_['lsoda\_mesg'] = 0 mod.\_\_settings\_\_['mode\_state\_mesg'] = 0  
 mod.\_\_settings\_\_['solver\_switch\_warning'] = False

Arguments: None

**SetStateSymb** (*flux, metab*)

SetStateSymb(flux,metab)

Sets the individual steady-state flux and concentration attributes as mod.J\_<reaction> and mod.<species>\_ss

Arguments:

flux: the steady-state flux array metab: the steady-state concentration array

**SimPlot** (*plot='species', filename=None, title=None, log=None, format='lines'*)

Plot the simulation results, uses the new UPI pysces.plt interface:

- *plot*: output to plot (default='species')
- 'all' rates and species
- 'species' species
- 'rates' reaction rates
- [*'SI'*, *'RI'*, ] a list of model attributes (species, rates)
- *filename* if not None file is exported to filename (default=None)
- *title* the plot title (default=None)
- *log* use log axis for 'x', 'y', 'xy' (default=None)
- *format* line format, backend dependant (default='')

### **Simulate** (*userinit=0*)

PySCeS integration driver routine that evolves the system over the time. Resulting array of species concentrations is stored in the **mod.data\_sim** object Initial concentrations can be selected using *mod.\_\_settings\_\_[ 'mode\_sim\_init' ]* (default=0):

- 0 initialise with initial concentrations
- 1 initialise with a very small (close to zero) value
- 2 initialise with results of previously calculated steady state
- 3 initialise with final point of previous simulation

*userinit* values can be (default=0):

- 0: initial species concentrations initialised from (mod.S\_init), time array calculated from sim\_start/sim\_end/sim\_points
- 1: initial species concentrations initialised from (mod.S\_init) existing "mod.sim\_time" used directly
- 2: initial species concentrations read from "mod.\_\_inspec\_\_", "mod.sim\_time" used directly

### **State** ()

State()

PySCeS non-linear solver driver routine. Solve for a steady state using HYBRD/NLEQ2/FINTSLV algorithms. Results are stored in mod.state\_species and mod.state\_flux. The results of a steady-state analysis can be viewed with the mod.showState() method.

The solver can be initialised in 3 ways using the mode\_state\_init switch. mod.mode\_state\_init = 0 initialize with species initial values mod.mode\_state\_init = 1 initialize with small values mod.mode\_state\_init = 2 initialize with the final value of a 10-logstep simulation numpy.logspace(0,5,18)

Arguments: None

### **Stoich\_matrix\_SetValue** (*species, reaction, value*)

Stoich\_matrix\_SetValue(species, reaction, value)

Change a stoichiometric coefficient's value in the N matrix. Only a coefficient's magnitude may be set, in other words a coefficient's value must remain negative, positive or zero. After changing a coefficient it is necessary to Reanalyse the stoichiometry.

Arguments:

species: species name (s0) reaction: reaction name (R4) value: new coefficient value

**Stoichiometry\_Analyse** (*override=0, load=0*)

Stoichiometry\_Analyse(override=0,load=0)

Perform a structural analyses. The default behaviour is to construct and analyse the model from the parsed model information. Overriding this behaviour analyses the stoichiometry based on the current stoichiometric matrix. If load is specified PySCeS tries to load a saved stoichiometry, otherwise the stoichiometric analysis is run. The results of the analysis are checked for floating point error and nullspace rank consistency.

Arguments:

override [default=0]: override stoichiometric analysis intialisation from parsed data

load [default=0]: load a presaved stoichiometry

**Stoichiometry\_Init** (*nmatrix, load=0*)

Stoichiometry\_Init(nmatrix,load=0)

Initialize the model stoichiometry. Given a stoichiometric matrix N, this method will return an instantiated PyscesStoich instance and status flag. Alternatively, if load is enabled, PySCeS will attempt to load a previously saved stoichiometric analysis (saved with Stoichiometry\_Save\_Serial) and test it's correctness. The status flag indicates 0 = reanalyse stoichiometry or 1 = complete structural analysis preloaded.

Arguments:

nmatrix: The input stoichiometric matrix, N load [default=0]: try to load a saved stoichiometry (1)

**Stoichiometry\_Load\_Serial** ()

Stoichiometry\_Load\_Serial()

Load a saved stoichiometry saved with mod.Stoichiometry\_Save\_Serial() and return a stoichiometry instance.

Arguments: None

**Stoichiometry\_ReAnalyse** ()

Stoichiometry\_ReAnalyse()

Reanalyse the stoichiometry using the current N matrix ie override=1 (for use with mod.Stoich\_matrix\_SetValue)

Arguments: None

**Stoichiometry\_Save\_Serial** ()

Serialize and save a Stoichiometric instance to binary pickle Stoichiometry\_Save\_Serial()

Serialise and save the current model stoichiometry to a file with name <model>\_stoichiometry.pscdat in the mod.\_\_settings\_\_[ 'serial\_dir' ] directory (default: mod.model\_output/pscdat)

Arguments: None

**TestSimState** (*endTime=10000, points=101, diff=1.0000000000000001e-005*)**Deprecated****User\_Function** ()**Deprecated**

**Write\_array** (*input, File=None, Row=None, Col=None, close\_file=0, separator=' '*)

Write\_array(input,File=None,Row=None,Col=None,close\_file=0,separator=' ')

Write an array to File with optional row/col labels. A ' ' separator can be specified to create a CSV style file.

mod.\_\_settings\_\_[ 'write\_array\_header' ]: add <filename> as a header line (1 = yes, 0 = no) mod.\_\_settings\_\_[ 'write\_array\_spacer' ]: add a space after the header line (1 = yes, 0 = no) mod.\_\_settings\_\_[ 'write\_arr\_lflush' ]: set the flush rate for large file writes

Arguments:

input: the array to be written File [default=None]: an open, writable Python file object Row [default=None]: a list of row labels Col [default=None]: a list of column labels close\_file [default=0]: close the file after write (1) or leave open (0) separator [default=' ']: the column separator to use

**Write\_array\_html** (*input, File=None, Row=None, Col=None, name=None, close\_file=0*)

Write\_array\_html(input,File=None,Row=None,Col=None,name=None,close\_file=0)

Write an array as an HTML table (no header/footer) or complete document. Tables are formatted with coloured columns if they exceed a specified size.

mod.\_\_settings\_\_[ 'write\_array\_html\_header' ]: write the HTML document header mod.\_\_settings\_\_[ 'write\_array\_html\_footer' ]: write the HTML document footer

Arguments:

input: the array to be written File [default=None]: an open, writable Python file object Row [default=None]: a list of row labels Col [default=None]: a list of column labels name [default=None]: an HTML table description line close\_file [default=0]: close the file after write (1) or leave open (0)

**Write\_array\_latex** (*input, File=None, Row=None, Col=None, close\_file=0*)

Write\_array\_latex(input,File=None,Row=None,Col=None,close\_file=0)

Write an array to an open file as a 'LaTeX' {array}

Arguments:

input: the array to be written File [default=None]: an open, writable Python file object Row [default=None]: a list of row labels Col [default=None]: a list of column labels close\_file [default=0]: close the file after write (1) or leave open (0)

**doEigen** ()

doEigen()

Calculate the eigenvalues, automatically performs a steady state and elasticity analysis.

Calls: State() EvalEvar() Evaleigen()

Arguments: None

**doEigenMca** ()

doEigenMca()

Calculate a full Control Analysis and eigenvalues, automatically performs a steady state, elasticity, control analysis.

Calls: State() EvalEvar() EvalCC() Evaleigen()

Arguments: None

**doEigenShow** ()

doEigenShow()

Calculate the eigenvalues, automatically performs a steady state and elasticity analysis and displays the results.

Calls: doEigen() showEigen()

Arguments: None

**doElas** ()

doElas()

Calculate the model elasticities, this method automatically calculates a steady state.

Calls: State() EvalEvar() EvalEpar()

Arguments: None

**doLoad** (*stoich\_load=0*)

doLoad(stoich\_load=0)

Load and instantiate a PySCeS model so that it can be used for further analyses.

Calls model loading subroutines: Stoichiometry\_Analyse() [override=0,load=stoich\_load] InitialiseModel()

Arguments:

stoich\_load [default=0]: try to load a stoichiometry saved with Stoichiometry\_Save\_Serial()

**doMca** ()

doMca()

Perform a complete Metabolic Control Analysis on the model, automatically calculates a steady state.

Calls: State() EvalEvar() EvalEpar() EvalCC()

Arguments: None

**doMcaRC** ()

doMca()

Perform a complete Metabolic Control Analysis on the model, automatically calculates a steady state.

Calls: State() EvalEvar() EvalEpar() EvalCC() EvalRC()

Arguments: None

**doSim** (*end=10.0, points=21*)

doSim(end=10.0,points=20.0)

Run a time simulation from t=0 to t=sim\_end with sim\_points.

Calls: Simulate()

Arguments:

end [default=10.0]: simulation end time points [default=20.0]: number of points in the simulation

**doSimPerturb** (*pl, end*)

**Deprecated:** use events instead

**doSimPlot** (*end=10.0, points=21, plot='species', fmt='lines', filename=None*)

Run a time simulation from t=0 to t=sim\_end with sim\_points and plot the results.

The required output data and format can be set:

- *end\** the end time (default=10.0)
- *points* the number of points in the simulation (default=20.0)
- *plot* (default='species') select output data
  - 'species'
  - 'rates'
  - 'all' both species and rates
- *fmt* plot format, UPI backend dependent (default='') or the *CommonStyle* 'lines' or 'points'.
- *filename* if not None (default) then the plot is exported as *filename.png*

Calls: - **Simulate()** - **SimPlot()**

#### **doState ()**

doState()

Calculate the steady-state solution of the system.

Calls: State()

Arguments: None

#### **doStateShow ()**

doStateShow()

Calculate the steady-state solution of a system and show the results.

Calls: State() showState()

Arguments: None

#### **showCC (File=None)**

showCC(File=None)

Print all control coefficients as 'LaTeX' formatted strings to the screen or file.

Arguments:

File [default=None]: an open, writable Python file object

#### **showConserved (File=None, screenwrite=1, fmt='%2.3f')**

showConserved(File=None,screenwrite=1,fmt='%2.3f')

Print the moiety conserved cycles present in the system.

Arguments:

File [default=None]: an open writable Python file object  
 screenwrite [default=1]: write results to console (0 means no response)  
 fmt [default='%2.3f']: the output number format string

#### **showEigen (File=None)**

showEigen(File=None)

Print the eigenvalues and stability analysis of a system generated with EvalEigen() to the screen or file.

Arguments:

File [default=None]: an open, writable Python file object

#### **showElas (File=None)**

showElas(File=None)

Print all elasticities to screen or file as 'LaTeX' compatible strings. Calls showEvar() and showEpar()

Arguments:

File [default=None]: an open writable Python file object

**showEpar** (*File=None*)

showEpar(File=None)

Write out all nonzero parameter elasticities as 'LaTeX' formatted strings, alternatively write to file.

Arguments:

File [default=None]: an open writable Python file object

**showEvar** (*File=None*)

showEvar(File=None)

Write out all variable elasticities as 'LaTeX' formatted strings, alternatively write results to a file.

Arguments:

File [default=None]: an open writable Python file object

**showFluxRelationships** (*File=None*)

showConserved(File=None)

Print the flux relationships present in the system.

Arguments:

File [default=None]: an open writable Python file object

**showK** (*File=None, fmt='%2.3f'*)

showK(File=None,fmt='%2.3f')

Print the Kernel matrix (K), including row and column labels to screen or File.

Arguments:

File [default=None]: an open, writable Python file object  
fmt [default='%2.3f']: output number format

**showL** (*File=None, fmt='%2.3f'*)

showL(File=None,fmt='%2.3f')

Print the Link matrix (L), including row and column labels to screen or File.

Arguments:

File [default=None]: an open, writable Python file object  
fmt [default='%2.3f']: output number format

**showModel** (*filename=None, filepath=None, skipcheck=0*)

showModel(filename=None,filepath=None,skipcheck=0)

The PySCeS 'save' command, prints the entire model to screen or File in a PSC format. (Currently this only applies to basic model attributes, ! functions are not saved).

Arguments:

filename [default=None]: the output PSC file  
filepath [default=None]: the output directory  
skipcheck [default=0]: skip check to see if the file exists (1) auto-averwrite

**showModifiers** (*File=None*)

showModifiers(File=None)

Prints the current value of the model's modifiers per reaction to screen or file.

Arguments:

File [default=None]: an open, writable Python file object

**showN** (*File=None, fmt='%2.3f'*)

showN(File=None,fmt='%2.3f')

Print the stoichiometric matrix (N), including row and column labels to screen or File.

Arguments:

File [default=None]: an open, writable Python file object  
fmt [default='%2.3f']: output number format

**showNr** (*File=None, fmt='%2.3f'*)

showNr(File=None,fmt='%2.3f')

Print the reduced stoichiometric matrix (Nr), including row and column labels to screen or File.

Arguments:

File [default=None]: an open, writable Python file object  
fmt [default='%2.3f']: output number format

**showODE** (*File=None, fmt='%2.3f'*)

showODE(File=None,fmt='%2.3f')

Print a representation of the full set of ODE's generated by PySCeS to screen or file.

Arguments:

File [default=None]: an open, writable Python file object  
fmt [default='%2.3f']: output number format

**showODEr** (*File=None, fmt='%2.3f'*)

showODEr(File=None,fmt='%2.3f')

Print a representation of the reduced set of ODE's generated by PySCeS to screen or file.

Arguments:

File [default=None]: an open, writable Python file object  
fmt [default='%2.3f']: output number format

**showPar** (*File=None*)

showPar(File=None)

Prints the current value of the model's parameter values (mod.P) to screen or file.

Arguments:

File [default=None]: an open, writable Python file object

**showRate** (*File=None*)

Prints the current rates of all the reactions using the current parameter values and species concentrations

•File an open, writable Python file object (default=None)

**showRateEq** (*File=None*)

showRateEq(File=None)

Prints the reaction stoichiometry and rate equations to screen or File.

Arguments:

File [default=None]: an open, writable Python file object

**showSpecies** (*File=None*)

showSpecies(*File=None*)

Prints the current value of the model's variable species (mod.X) to screen or file.

Arguments:

File [default=None]: an open, writable Python file object

**showSpeciesFixed** (*File=None*)

showSpeciesFixed(*File=None*)

Prints the current value of the model's fixed species values (mod.X) to screen or file.

Arguments:

File [default=None]: an open, writable Python file object

**showSpeciesI** (*File=None*)

showSpeciesI(*File=None*)

Prints the current value of the model's variable species initial values (mod.X\_init) to screen or file.

Arguments:

File [default=None]: an open, writable Python file object

**showState** (*File=None*)

showState(*File=None*)

Prints the result of the last steady-state analyses. Both steady-state flux's and species concentrations are shown.

Arguments:

File [default=None]: an open, writable Python file object

**class ReactionObj** (*mod, name, kl, klrepl='self'*)

Defines a reaction with a KineticLaw *kl*, *\*formula* and *name* bound to a model instance, *mod*.

**setKineticLaw** (*kl, klrepl='self'*)

**class ScanDataObj** (*par\_label*)

New class used to store parameter scan data (uses StateDataObj)

**addModData** (*mod, \*args*)

**addPoint** (*ipar, ssdata*)

takes a list/array of input parameter values and the associated ssdata object

**closeScan** ()

**getAllScanData** (*lbls=False*)

**getFluxes** (*lbls=False*)

**getModData** (*lbls=False*)

**getRules** (*lbls=False*)

**getScanData** (*\*args, \*\*kwargs*)

getScanData(*\*args*) feed this method species/flux/rule/mod labels and it will return an array of [parameter(s), sp1, f1, ....]

**getSpecies** (*lbls=False*)

**getXData** (*lbls=False*)

**setLabels** (*ssdata*)

**class StateDataObj** ()

New class used to store steady-state data.

**getAllStateData** (*lbls=False*)

Return all available data as species+fluxes+rules if *lbls=True* returns (array,labels)  
else just array

**getFluxes** (*lbls=False*)

return flux array

**getRules** (*lbls=False*)

Return rule array

**getSpecies** (*lbls=False*)

return species array

**getStateData** (*\*args, \*\*kwargs*)

getSimData(*\*args*) feed this method species/rate labels and it will return an array of  
[time, sp1, r1, ....]

**getXData** (*lbls=False*)

Return xdata array

**setFluxes** (*fluxes, lbls=None*)

set the flux array

**setRules** (*rules, lbls=None*)

Set the results of rate rules

**setSpecies** (*species, lbls=None*)

Set the species array

**setXData** (*xdata, lbls=None*)

Sets extra simulation data

**class StructMatrix** (*array, ridx, cidx, row=None, col=None*)

This class is specifically designed to store structural matrix information give it an array  
and row/col index permutations it can generate its own row/col labels given the label src.

**getByIdx** (*row, col*)

**getByName** (*row, col*)

**getColsByIdx** (*\*args*)

Return the columns referenced by index (1,3,5)

**getColsByName** (*\*args*)

Return the columns referenced by label ('s','x','d')

**getIndexes** (*axis='all'*)

Return the matrix indexes ([rows],[cols]) where axis='row'/'col'/'all'

**getLabels** (*axis='all'*)

Return the matrix labels ([rows],[cols]) where axis='row'/'col'/'all'

**getRowsByIdx** (*\*args*)

Return the rows referenced by index (1,3,5)

**getRowsByName** (*\*args*)

Return the rows referenced by label ('s','x','d')

**setByIdx** (*row, col, val*)

**setName** (*row, col, val*)

**setCol** (*src*)

Assuming that the col index array is a permutation (full/subset) of a source label array by supplying that src to setCol maps the row labels to cidx and creates self.col (col label list)

**setRow** (*src*)

Assuming that the row index array is a permutation (full/subset) of a source label array by supplying that source to setRow it maps the row labels to ridx and creates self.row (row label list)

**shape** ()

**class WasteManagement** ()

**chkmdir** ()

chkmdir()

Import and grab pysces.model\_dir

Arguments: None

**chkpsc** (*File*)

chkpsc(File)

Check whether the filename "File" has a '.psc' extension and adds one if not.

Arguments:

File: filename string

## PyscesScan

PySCeS classes for continuations and multi-dimensional parameter scans

**class PITCONScanUtils** (*model*)

Static Bifurcation Scanning utilities using PITCON, call with loaded model object. Hopefully nobody else was trying to use the older class as it was horrible. This new one is is leaner, meaner and pretty cool ;-)

**analyseData** (*analysis='elas'*)

Performs “analysis” on the PITCON generated set of steady-state results where analysis is:

- ‘elasv’ = variable elasticities
- ‘elasp’ = parameter elasticities
- ‘elas’ = all elasticities
- ‘mca’ = control coefficients
- ‘resp’ = response coefficients
- ‘eigen’ = eigen values
- ‘all’ = all of the above

Higher level analysis types automatically enable the lower level analysis needed e.g. selecting ‘mca’ implies ‘elasv’ etc. User output defined with `mod.setUserOutput()` is stored in the `mod.res_user` array.

**getArrayListAsArray** (*array\_list*)

Stack (concatenate) the list of arrays into a single array.

**runContinuation** (*parameter, low, high, density, par3d=None*)

Run the continuation using the following parameters:

Args:

- parameter = str(the parameter to be scanned)
- low = float(lower bound)
- high = float(upper bound)
- density = int(the number of initial points)
- par3d = float(extra 3d parameter to insert into the output array) this parameter is not set ONLY used in output

After running the continuation the results are stored in numpy arrays

- `mod.res_idx` = scan parameter values (and optionally `par3d`)

- `mod.res_metab` = steady-state species concentrations
- `mod.res_flux` = steady-state flux values

**setUserOutput** (*\*args*)

Set the user output required as n string arguments.

**class Scanner** (*mod*)

Arbitrary dimension generic scanner. This class is initiated with a loaded PySCeS model and then allows the user to define scan parameters see `self.addScanParameter()` and user output see `self.addUserOutput()`. Steady-state results are always stored in `self.SteadyStateResults` while user output can be found in `self.UserOutputResults` - brett 2007.

**Analyze** ()

The analysis method, the mode is automatically set by the `self.addUserOutput()` method but can be reset by the user.

**Run** (*ReRun=False*)

Run the parameter scan

**RunAgain** ()

While it is impossible to change the generator/range structure of a scanner (just build another one) you can ‘in principle’ change the User Output and run it again.

**StoreData** ()

Internal function which concatenates and stores the data generated by `Analyze`.

**addScanParameter** (*name, start, end, points, log=False, slave=False*)

Add a parameter to scan (an axis if you like) input is:

- `str(name)` = model parameter name
- `float(start)` = lower bound of scan
- `float(end)` = upper bound of scan
- `int(points)` = number of points in scan range
- `bool(log)` = Use a logarithmic (base10) range
- `bool(slave)` = Scan parameters can be masters i.e. an independent axis or a “slave” which moves synchronously with the previously defined parameter range.

The first `ScanParameter` cannot be a slave.

**addUserOutput** (*\*kw*)

Add output parameters to the scanner as a collection of one or more string arguments (`'O1'`, `'O2'`, `'O3'`, `'On'`). These are evaluated at each iteration of the scanner and stored in the `self.UserOutputResults` array. The list of output is stored in `self.UserOutputList`.

**getOutput** ()

Will be the new output function.

**getResultMatrix** (*stst=False*)

Returns an array of result data. I’m keepin this for backwards compatibility but it will be replaced by a `getOutput()` method when this scanner is updated to use the new `data_scan` object.

- stst* add steady-state data to output array

If *stst* is True output has dimensions [scan\_parameters]+[state\_species+state\_flux]+[Useroutput] otherwise [scan\_parameters]+[Useroutput].

**makeRange** (*start, end, points, log*)

Should be pretty self evident it defines a range:

- float(start)
- float(end)
- int(points)
- bool(log)

**rangeGen** (*name, start, end, points, log*)

This is where things get more interesting. This function creates a cycling generator which loops over a parameter range.

- parameter* name
- start* value
- end* value
- points*
- log* scale

**resetInputParameters** ()

Just remembered what this does, I think it resets the input model parameters after a scan run.

**setModValue** (*name, value*)

An easy one, assign value to name of the instantiated PySCeS model attribute

**stepGen** (*offset*)

Another looping generator function. The idea here is to create a set of generators for the scan parameters. These generators then all fire together and determine whether the range generators should advance or not. Believe it or not this dynamically creates the matrix of parameter values to be evaluated.

**testInputParameter** (*name*)

This tests whether a str(name) is an attribute of the model



## PyscesInterfaces

Interfaces converting to and from PySCeS models - makes use of Brett's Core2

**class Core2interfaces ()**

Defines interfaces for translating PySCeS model objects into and from other formats.

**convertSBML2PSC** (*sbmlfile*, *sbmdir=None*, *pscfile=None*, *pscdir=None*)

Convert an SBML file to a PySCeS MDL input file.

- sbmlfile*: the SBML file name
- sbmdir*: the directory of SBML files (if None current working directory is assumed)
- pscfile*: the output PSC file name (if None *sbmlfile.psc* is used)
- pscdir*: the PSC output directory (if None the *pysces.model\_dir* is used)

**readMod2Core** (*mod*, *iValues=True*)

Convert a PySCeS model object to core2

- iValues*: if True then the models initial values are used (or the current values if False).

**readSBMLToCore** (*filename*, *directory=None*)

Reads the SBML file specified with filename and converts it into a core2 object *pysces.interface.core*

- filename*: the SBML file
- directory*: (optional) the SBML file directory None means try the current working directory

**writeCore2PSC** (*filename=None*, *directory=None*, *getstrbuf=False*)

Writes a Core2 object to a PSC file.

- filename*: writes <filename>.xml or <model\_name>.xml if None
- directory*: (optional) an output directory
- getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

**writeCore2SBML** (*filename=None*, *directory=None*, *getdocument=False*)

Writes Core2 object to an SBML file.

- filename*: writes <filename>.xml or <model\_name>.xml if None
- directory*: (optional) an output directory
- getdocument*: if True an SBML document object is returned instead of writing to disk or

**writeMod2PSC** (*mod, filename=None, directory=None, iValues=True, getstrbuf=False*)

Writes a PySCeS model object to a PSC file.

- filename*: writes <filename>.psc or <model\_name>.psc if None
- directory*: (optional) an output directory
- iValues*: if True then the models initial values are used (or the current values if False).
- getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

**writeMod2SBML** (*mod, filename=None, directory=None, iValues=True, getdocument=False, getstrbuf=False*)

Writes a PySCeS model object to an SBML file.

- filename*: writes <filename>.xml or <model\_name>.xml if None
- directory*: (optional) an output directory
- iValues*: if True then the models initial values are used (or the current values if False).
- getdocument*: if True an SBML document object is returned instead of writing to disk or
- getstrbuf*: if True a StringIO buffer is returned instead of writing to disk

## PyscesStoich

PySCeS stoichiometric analysis classes.

**class MathArrayFunc ( )**

PySCeS array functions - used by Stoich

**MatrixFloatFix** (*mat, val=1.0000000000000001e-015*)

MatrixFloatFix(mat,val=1.e-15)

Clean an array removing any floating point artifacts defined as being smaller than a specified value. Processes an array inplace

Arguments:

mat: the input 2D array val [default=1.e-15]: the threshold value (effective zero)

**MatrixValueCompare** (*matrix*)

MatrixValueCompare(matrix)

Finds the largest/smallest abs(value) > 0.0 in a matrix. Returns a tuple containing (smallest,largest) values

Arguments:

matrix: the input 2D array

**SwapCol** (*res\_a, r1, r2*)

SwapCol(res\_a,r1,r2)

Swap two columns using BLAS swap, arrays can be (or are upcast to) type double (d) or double complex (D). Returns the colswapped array

Arguments:

res\_a: the input array r1: the first column to be swapped r2: the second column to be swapped

**SwapCold** (*res\_a, c1, c2*)

SwapCold(res\_a,c1,c2)

Swaps two double (d) columns in an array using BLAS DSWAP. Returns the colswapped array.

Arguments:

res\_a: input array c1: column index 1 c2: column index 2

**SwapColz** (*res\_a, c1, c2*)

SwapColz(res\_a,c1,c2)

Swaps two double complex (D) columns in an array using BLAS ZSWAP. Returns the colswapped array.

Arguments:

res\_a: input array c1: column index 1 c2: column index 2

**SwapElem** (*res\_a, r1, r2*)

SwapElem(res\_a,r1,r2)

Swaps two elements in a 1D vector

Arguments:

res\_a: the input vector r1: index 1 r2: index 2

**SwapRow** (*res\_a, r1, r2*)

SwapRow(res\_a,r1,r2)

Swaps two rows using BLAS swap, arrays can be (or are upcast to) type double (d) or double complex (D). Returns the rowswapped array.

Arguments:

res\_a: the input array r1: the first row index to be swapped r2: the second row index to be swapped

**SwapRowd** (*res\_a, r1, r2*)

SwapRowd(res\_a,c1,c2)

Swaps two double (d) rows in an array using BLAS DSWAP. Returns the rowswapped array.

Arguments:

res\_a: input array c1: row index 1 c2: row index 2

**SwapRowz** (*res\_a, r1, r2*)

SwapRowz(res\_a,c1,c2)

Swaps two double complex (D) rows in an array using BLAS ZSWAP. Returns the rowswapped array.

Arguments:

res\_a: input array c1: row index 1 c2: row index 2

**assertRank2** (*\*arrays*)

assertRank2(\*arrays)

Check that we are using a 2D array

Arguments:

\*arrays: input array(s)

**castCopyAndTranspose** (*type, \*arrays*)

castCopyAndTranspose(type, \*arrays)

Cast numeric arrays to required type and transpose

Arguments:

type: the required type to cast to \*arrays: the arrays to be processed

**commonType** (*\*arrays*)

commonType(\*arrays)

Numeric detect and set array precision (will be replaced with new scipy.core compatible code when ready)

Arguments:

\*arrays: input arrays

**class Stoich** (*input*)

PySCeS stoichiometric analysis class: initialized with a stoichiometric matrix N (input)

**AnalyseK** ()

AnalyseK()

Evaluate the stoichiometric matrix and calculate the nullspace using LU decomposition and backsubstitution . Generates the MCA K and Ko arrays and associated row and column vectors

Arguments: None

**AnalyseL** ()

AnalyseL()

Evaluate the stoichiometric matrix and calculate the left nullspace using LU factorization and backsubstitution. Generates the MCA L, Lo, Nr and Conservation matrix and associated row and column vectors

Arguments: None

**BackSubstitution** (*res\_a, row\_vector, column\_vector*)

BackSubstitution(res\_a,row\_vector,column\_vector)

Jordan reduction of a scaled upper triangular matrix. The returned array is now in the form [I R] and can be used for nullspace determination. Modified row and column tracking vectors are also returned.

Arguments:

res\_a: unitary pivot upper triangular matrix row\_vector: row tracking vector column\_vector: column tracking vector

**GetUpperMatrix** (*a*)

GetUpperMatrix(a)

Core analysis algorithm; an input is preconditioned using PivotSort\_initial and then cycles of PLUfactorize and PivotSort are run until the factorization is completed. During this process the matrix is reordered by column swaps which emulates a full pivoting LU factorization. Returns the pivot matrix P, upper factorization U as well as the row/col tracking vectors.

Arguments:

a: a stoichiometric matrix

**GetUpperMatrixUsingQR** (*a*)

GetUpperMatrix(a)

Core analysis algorithm; an input is preconditioned using PivotSort\_initial and then cycles of PLUfactorize and PivotSort are run until the factorization is completed. During this process the matrix is reordered by column swaps which emulates a full pivoting LU factorization. Returns the pivot matrix P, upper factorization U as well as the row/col tracking vectors.

Arguments:

a: a stoichiometric matrix

**K\_split\_R** (*R\_a, row\_vector, column\_vector*)

K\_split\_R(R\_a,row\_vector,column\_vector)

Using the R factorized form of the stoichiometric matrix we now form the K and Ko matrices. Returns the r\_ipart,Komatrix,Krow,Kcolumn,Kmatrix,Korow,info

Arguments:

R\_a: the Gauss-Jordan reduced stoichiometric matrix  
 row\_vector: row tracking vector  
 column\_vector: column tracking vector

**L\_split\_R** (*Nfull, R\_a, row\_vector, column\_vector*)

L\_split\_R(Nfull,R\_a,row\_vector,column\_vector)

Takes the Gauss-Jordan factorized  $N^T$  and extract the L, Lo, conservation (I - Lo) and reduced stoichiometric matrices. Returns: lmatrix\_col\_vector, lomatrix, lomatrix\_row, lomatrix\_co, nrmatrix, Nred\_vector\_row, Nred\_vector\_col, info

Arguments:

Nfull: the original stoichiometric matrix N  
 R\_a: gauss-jordan factorized form of  $N^T$   
 row\_vector: row tracking vector  
 column\_vector: column tracking vector

**PLUfactorize** (*a\_in*)

PLUfactorize(a\_in)

Performs an LU factorization using LAPACK D/ZGtrf. Returns LU - combined factorization, IP - rowswap information and info - Gtrf error control.

Arguments:

a\_in: the matrix to be factorized

**PivotSort** (*a, row\_vector, column\_vector*)

PivotSort(a,row\_vector,column\_vector)

This is a sorting routine that accepts a matrix and row/column vectors and then sorts them so that: there are no zero rows (by swapping with first non-zero row) The abs(largest) pivots are moved onto the diagonal to maintain numerical stability. Row and column swaps are recorded in the tracking vectors.

Arguments:

a: the input array  
 row\_vector: row tracking vector  
 column\_vector: column tracking vector

**PivotSort\_initial** (*a, row\_vector, column\_vector*)

PivotSort\_initial(a,row\_vector,column\_vector)

This is a sorting routine that accepts a matrix and row/column vectors and then sorts them so that: the abs(largest) pivots are moved onto the diagonal to maintain numerical stability i.e. the matrix diagonal is in descending max(abs(value)). Row and column swaps are recorded in the tracking vectors.

Arguments:

a: the input array  
 row\_vector: row tracking vector  
 column\_vector: column tracking vector

**SVD\_Rank\_Check** (*matrix=None, factor=10000.0, resultback=0*)

SVD\_Rank\_Check(matrix=None,factor=1.0e4,resultback=0)

Calculates the dimensions of L/L0/K/K) by way of SVD and compares them to the Gauss-Jordan results. Please note that for LARGE ill conditioned matrices the SVD can become numerically unstable when used for nullspace determinations

Arguments:

matrix [default=None]: the stoichiometric matrix default is self.Nmatrix  
 factor [default=1.0e4]: factor used to calculate the 'zero pivot' mask = mach\_eps\*factor  
 resultback [default=0]: return the SVD results, U, S, vh

**ScalePivots** (*a\_one*)

ScalePivots(*a\_one*)

Given an upper triangular matrix U, this method scales the diagonal (pivot values) to one.

Arguments:

*a\_one*: an upper triangular matrix U

**SplitLU** (*plu, row, col, t=None*)

SplitLU(*plu, row, col, t*)

PLU takes the combined LU factorization computed by PLUfactorize and extracts the upper matrix. Returns U.

Arguments:

*plu*: LU factorization *row*: row tracking vector *col*: column tracking vector *t* [default=None]: typecode argument (currently not used)



## PyscesLink

Interfaces to external software and API's, has replaced the PySCeS contrib classes.

**class METATOOLink** (*mod*, *\_\_metatool\_path\_\_=None*)

New interface to METATOOL binaries

**doEModes** ()

doEModes()

Calculate the elementary modes by way of an interface to MetaTool.

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). <http://www.biologie.hu-berlin.de/biophysics/Theory/tpfeiffer/metatool.html>

Arguments: None

**getEModes** ()

getEModes()

Returns the elementary modes as a linked list of fluxes

**showEModes** (*File=None*)

showEModes(*File=None*)

Print the results of an elementary mode analysis, generated with doEModes(), to screen or file.

Arguments: *File* [default=None]: Boolean, if True write parsed elementary modes to file

**class SBWLayoutWebLink** ()

Enables access to DrawNetwork and SBMLLayout web services at [www.sys-bio.org](http://www.sys-bio.org)

**drawNetworkGetSBMLwithLayout** ()

**drawNetworkLoadSBML** ()

**getSBML** ()

**getSBMLlayout** ()

**getSVG** ()

**getVersion** ()

**layoutModuleGetSVG** ()

**layoutModuleLoadSBML** ()

**loadSBMLFileFromDisk** (*File, Dir=None*)

**loadSBMLFromString** (*str*)

**setProxy** (*\*\*kwargs*)

Set as many proxy settings as you need. You may supply a user name without a password in which case you will be prompted to enter one (once) when required (NO guarantees, implied or otherwise, on password security AT ALL). Arguments can be:

user = 'daUser', pwd = 'daPassword', host = 'proxy.paranoid.net', port = 3128

**urlGET** (*host, urlpath*)

**urlPOST** (*host, urlpath, data*)

**class SBWLink** ()

Generic access for local SBW services using SBWPython

**SBW\_exposeAll** (*module*)

**SBW\_getActiveModules** ()

**SBW\_loadModule** (*module\_name*)

# **Part XI**

## **Indices and tables**



- *Index*
- *Module Index*
- *Search Page*