The PySCeS Reference Manual (pre-release 0.1.4)

http://pysces.sourceforge.net

Brett G. Olivier*

Triple-J Group for Molecular Cell Physiology
National Bioinformatics Network
Stellenbosch University

*bgoli@users.sourceforge.net

Contents

PySCeS: the Python Simulator for Cellular Systems

In this document we shall describe the design and implementation of the Python Simulator for Cellular Systems[1] – PySCeS. In order to do this we use a multi-threaded approach that can conceptually be viewed as a composite of two 'virtual' strands:

- *The PySCeS Developers Reference* highlights the theory and algorithms used to satisfy our original design goals.
- *The PySCeS User Manual* contains all the user options, algorithm parameters and analysis methods available in PySCeS.

Using this strategy, the basic program elements are presented in the order in which you might encounter them when using PySCeS (installing PySCeS, loading a model, calculating a steady state, etc.). Each element is then, where applicable, further expanded into the theory on which it is based, how it is implemented and the options and methods PySCeS provides when using it. With this in mind let's begin by looking at the PySCeS setup process.

1 Installing PySCeS

PySCeS is implemented as a Python package and is installed using the Python Distributions Utilities (Distutils) [3]. Distutils allows for the installation of Python scripts as well as the automated compiling of C and C++-based extension libraries. SciPy provides an extension to Distutils, i.e. scipy_distutils, which provides the same facilities for Fortran extension libraries. As PySCeS uses such libraries, it is therefore necessary to have a working SciPy installation (version 0.3.0 or newer) before PySCeS can be built and installed (see Chapter **??** for details on installing SciPy).

The extension modules used by PySCeS have been designed to be compiled with the GNU Compiler Collective (GCC) compiler suite. The GCC suite, which includes C, C++ and Fortran compilers, is available on most Linux and Un*x type systems. On Microsoft Windows™ systems, PySCeS has been developed using the Minimalist GNU for Windows[2] (MinGW) GCC port. Using MinGW, native Windows libraries can be compiled without the need for a POSIX emulation layer. Once the required prerequisites have been met PySCeS can be built and installed simply by running the setup script (please note root or administrator privileges may be needed to install PySCeS).

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

[1] PySCeS has been co-developed with Jannie Hofmeyr and Johann Rohwer. Parts of this document have been published in [1, 2]

[2] http://www.mingw.org

1.1 Inside the PySCeS setup process

The file and directory structure of the PySCeS distribution has been arranged to make it compatible with the Distutils installer. However, building the Fortran extension libraries is only one part of the complete installation procedure. In this section we look, in detail, at the mechanics behind the PySCeS install process.

1.1.1 User configurable options

The first section of `setup.py` contains user configurable options where the user can decide which of the external modules should be installed. This is necessary, as PySCeS is distributed with the source code of certain extension modules (currently NLEQ2, see Appendix **??** for details) that are not distributed under the GPL. By disabling the installation of these libraries in the setup script, PySCeS can be used in situations which might contradict these alternate licences.

- `pitcon = 1`: compile the PITCON extension module
- `metatool = 1`: build the MetaTool binaries
- `nleq2 = 1`: compile NLEQ2 extension module
- `nleq2_byteorder_override = 0`: override NLEQ2 byteorder selection.

1.1.2 Installing Scientific Python

PySCeS uses Konrad Hinsen's FirstDerivatives module included with his Scientific Python[3]. and therefore this extremely useful package is distributed as part of the PySCeS. The first step in the setup process is to check whether Scientific is installed or not. If not found, the user is given the option to install Scientific. If requested, setup will install Scientific (by calling its setup script) after which PySCeS setup will continue.

```
D:\cvs\pysces-0.1.3>python setup.py build --compiler=mingw32 install

Checking for Scientific ...

PySCeS uses Konrad Hinsen's Scientific Python (in addition to
SciPy). I can install it now if you like: yes/no? yes
```

This arrangement has the advantage that existing Scientific installations are not changed by the PySCeS setup process and that Scientific is properly configured by its own setup utility. Once the Scientific check is complete PySCeS setup then tries to compile the MetaTool binaries.

[3] http://starship.python.net/~hinsen/ScientificPython/

1.1.3 Building MetaTool binaries

MetaTool [4] is supplied as two C++ source files (meta4.3_double.cpp and meta4.3_int.cpp) which must first be compiled into executable binaries before they can be used with PySCeS. As Distutils is tailored to building extension libraries and not executables, a customized solution using shell scripts was developed (see Appendix **??**) to enable compilation on both Linux and Windows operating systems. Both MetaTool binaries are then added to a list of data files which Distutils later installs (for licence details see Appendix **??**).

Next, setup adds the sample model files distributed with PySCeS to the data file list and converts the line termination characters of these files (e.g. `<CR><LF>` for Windows based systems) in order to correspond to the host operating system's default.

1.1.4 Creating the PySCeS configuration file

PySCeS uses configuration files to determine the location of its compiled components and external libraries. In order to do this, the setup process (using the methods and templates located in `PyscesConfig.py`) generates a configuration file specific to the Python installation an that is installing it. Configuration files are also specifically tailored to the operating system that PySCeS is being used on. Appendix **??** has examples of configuration files for Linux and Windows based operating systems.

As PySCeS has been designed to run as a single user installation on Windows, the default paths to the model and work directories are explicitly specified. However, on Linux type operating systems PySCeS runs as a multiuser application and the user and model directories are determined by expansion of the user's `HOME` shell variable at runtime. Both of these paths may be overwritten by user defined configuration files which will be discussed later in this document.

1.1.5 Compiling extension modules

Finally, setup attempts to build the two Fortran libraries using the scipy_extension mechanism and F2PY [5]. The PITCON extension is built directly using distutils. The NLEQ2 module uses machine constants which are CPU specific and must first be set in the Fortran source file (`nleq2.f`).

In an attempt to automate this process, setup first determines the byte-order used by the processor and then selects one of the provided NLEQ2 source files that contains the appropriate settings. Currently, source files (nleq2_big.f, nleq2_little.f) are provided for CPUs which use IEEE arithmetic and where the most significant (big-endian, e.g. Motorola 68000) or least significant (little-endian, e.g. Intel i386) byte is stored first [6]. It is, however, possible to override this behaviour by setting `nleq2_byteorder_override = 1`, in which case users may select the constants appropriate for their architecture[4] and man-

[4]More information can be found in the NLEQ2 source code `nleq2.f` distributed with PySCeS or from

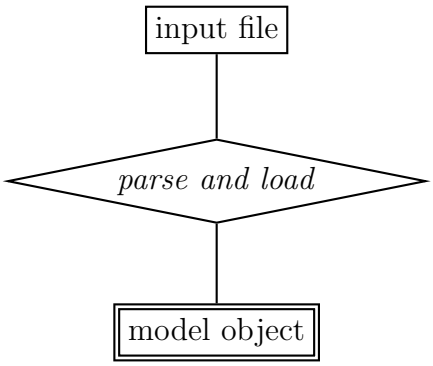

Fig. 1 *Loading a PySCeS model.* The input file is parsed and converted into a Python object

ually install the file `nleq2.f` which is the source file used by Distutils to build the NLEQ2 extension module.

2 The Tao of PySCeS: Part 1

> "software development is the process of converting human thought into ones and zeros" – unknown

In much the same way, the first step in the modelling process is to convert a description of a model into a computer model. Fig. 1 shows a schematic of the PySCeS loading process where a human readable/writable *input file* is loaded and parsed into a usable Python object. In the following sections we will look more closely at this process.

3 The PySCeS model input file

The PySCeS *input file* is an ASCII text file, similar to those used by simulation packages such as Scamp [7] and Jarnac [8], which describes a model in terms of its network structure, reaction rates and parameter values. Input files may have any filename (although filenames with spaces are not encouraged) except that, in order to ensure maximum cross platform portability between Windows, Macintosh and Linux systems, filenames must end with the extension (i.e. final four characters): `.psc`

3.1 Input file header and comments

There is no defined header in a `.psc` file but model details and comments can be placed at the beginning of an input file using comments. In general, comments may be inserted anywhere into an input file using a Python single line comment (`#`) for example:

ZIB (http://www.zib.de/SciSoft/ANT/nleq2.en.html)

```
# rohwer_sucrose1.psc
# Title Sucrose metabolism in sugar cane
# Johann M. Rohwer, Biochem. J. (2001) 358, 437-445
# Triple-J Group for Molecular Cell Physiology, Stellenbosch University
```

3.2 Fixed metabolites

In the first part of the input file the fixed or external metabolites (if any) need to be declared. The declaration line must begin with the keyword `FIX` followed directly by a colon which must be followed by a space separated list of parameter names. Parameter names can contain any sequence of letters (both capital or lowercase) or numbers, providing they begin with a letter and do not contain any spaces.

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

If no boundary metabolites need to be defined (e.g. a closed system) this declaration can be left out altogether.

3.3 Reaction stoichiometry and rate equations

The next part of the input file is used to describe the system's reaction stoichiometry and rate equations. Reactions are grouped together per reaction step and are defined as having a name (reaction identifier), a stoichiometry (substrates are converted to products) and rate equation (the catalytic activity). A formal description of the rate equation syntax is given in Appendix **??**.

3.3.1 Reaction name

Each reaction is given a unique name made up of letters and or numbers but must begin with a letter and not contain any spaces. A reaction name is declared and followed by a colon as show below.

```
Reac1a:

R2_P:
```

3.3.2 Reaction stoichiometry

On the line following the reaction name the reaction stoichiometry is defined. Reaction substrates are placed on the left hand side of an identifier which describes the reaction as either reversible (`=`) or irreversible (`>`), while products are placed on the right. The reversibility of a reaction is used in the structural analysis as part of the calculation of

the elementary modes and does not influence the chemical reversibility, as determined by the rate equation of the reaction step.

Each reagent's stoichiometric coefficient can be included in brackets {} immediately preceding the reagent name. If omitted, a coefficient of one is assumed. PySCeS is not limited to using integer coefficients and floating point stoichiometries {1.5} are also permitted.

```
{2}Hex_P = Suc6P + UDP   # reversible reaction

Fru_ex > Fru             # irreversible reaction
```

3.3.3 Reaction rate equation

Next, the rate equations should be written as a valid Python expression. Rate equations may fall across more than one line and all standard Python operators (`+-*/`) may be used, including the Python power operator (`**`) where, for example, 2^4 is written as `2**4`. There is no shorthand for multiplication in Python so $-2(a+b)^h$ would be written as `-2*(a+b)**h` and the normal Python operator precedence applies as summarized in Table 1.

`+, -`	Addition, subtraction
`*, /`	Multiplication, division
`+x, -x`	Positive, negative
`**`	Exponentiation

Table 1 Operator precedence increases from top to bottom and left to right, adapted from the *Python Language Reference* [9].

An example of a complete reaction step is shown below including the reaction name, stoichiometry and rate equation.

```
Reaction5:
   Fru + ATP = Hex_P + ADP
   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)
```

3.4 Initialization

A model property is declared and initialized using the form:

```
property = value
```

Initializations can be written in any order but should use neither shorthand floating point (`1.  `) nor shorthand exponential (`1.e-3`) syntax. Instead, full exponential (`1.0e-3`), decimal (`0.001`) and floating point syntax (`1.0`) should be used for initialization.

3.4.1 Parameters, external metabolites

The system's external metabolites and parameters may now be initialized. Although, generally speaking, these parameters are usually present in the rate equations they are not required to be. If such a parameter initialization is detected a harmless warning is generated when the model is parsed. If, on the other hand, an uninitialized parameter is detected a warning is generated and the model will not function properly.

```
# InitExt
X0 = 10.0
X4 = 1.0

# InitPar
Vf1 = 10.0
Ks1 = 1.0
```

3.4.2 Initializing variable metabolites

The initial concentrations of the models variable metabolites are used to calculate the moiety conserved sums [10] (if present) and for initializing various numerical routines. If you would like the metabolite pools to start empty, it is useful to initialize these values to a small value (e.g. 10^{-8}) rather than zero. This helps to prevent potential 'divide by zero' errors when the rate equations are evaluated.

```
# InitVar
S1 = 1.0e-03
S2 = 2.0e+02
```

Once a model has been formatted as an input file it can be loaded and analysed in either a PySCeS interactive session or program script.

4 Running PySCeS

PySCeS can either be used in a Python script or interactively from within a Python shell such as IDLE (default Python console) or IPython[5]. It is highly recommended

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

to use IPython, as it supports, amongst other things, a color terminal and `<TAB>`-style command completion on both Windows and Linux operating systems.

In order to streamline interactive sessions, PySCeS has been organized so that method or function calls begin with an uppercase letter, while attributes or properties begin with a lowercase letter. The general exceptions are the `doSomething()` and `showSomething()` methods where the operative word is preceded by `do` or `show` in lowercase.

PySCeS is installed as a Python package and starting a new session is simply a matter of importing it with `import pysces`.

```
Python 2.3.4 (#2, Jun 29 2004, 15:57:56)
[GCC 3.3.1 (Mandrake Linux 9.2 3.3.1-2mdk)] on linux2

>>> import pysces
MetaTool executables available
pitcon routines available
nleq2 routines available
You are using scipy version: 0.3.1_281.4213

PySCeS environment
******************
PySCeS ver 0.1.3 runtime: Mon, 10 Aug 2004 15:48:31
pysces.PyscesModel.model_dir = /home/bgoli/pscmodels
pysces.PyscesModel.output_dir = /home/bgoli/pysces

*********************************************************************
* Welcome to PySCeS (0.1.3) - Python Simulator for Cellular Systems *
* Copyright(C) Brett G. Olivier,2004 - http://pysces.sourceforge.net*
* Co-developed with J.-H.S. Hofmeyr and J.M. Rohwer                 *
* Triple-J Group for Molecular Cell Physiology                      *
* PySCeS is distributed under the GNU general public licence        *
* See README.txt for licence details                                *
*********************************************************************

>>>
```

When first imported, PySCeS displays some information about its environment and which of the extension modules are available. Section 1.1.4 of this document described how the PySCeS configuration file (`pyscfg.ini` see Appendix **??**) and default paths were created during the PySCeS installation. As part of the startup process PySCeS checks for the existence of a user configuration file. If this file is not found it is created in a default user directory, as specified in the main PySCeS configuration file. The user configurable paths are explicitly defined, depending on the operating system, as can be seen in Appendix **??**.

On a Linux system the default paths will be set to a *pysces* subdirectory in the directory referenced by the `HOME` shell variable. On Windows the default working directories are set to the *sys.prefix/site-packages/pysces* directory. The user configuration file allows you to customize the following two working directories:

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

The following guidelines should be followed when defining a path: spaces in pathnames are not supported, and on Windows double or escaped backslashes (\\) should be used as path separators. If, on the other hand, the user configuration file exists the working directories defined in this file are created (if necessary) and used for the PySCeS session.

By the use of dual configuration files a user's configuration data can be maintained between PySCeS installations, as once created the user configuration file is not overwritten. For maximum flexibility the `model_dir` path can be set for a PySCeS session by changing the module attribute directly:

```
pysces.PyscesModel.model_dir = "c:\\some-directory"
```

This value is used as the default model directory by model objects instantiated in this session. There are a number of general package functions available once PySCeS has been imported, however, these will be discussed later as part of Section 19.

4.1 Instantiating a PySCeS model object

Central to PySCeS is the `pysces.model` class, which, once instantiated with a model description can be used for further analysis. Model objects are instantiated with a PySCeS input file which contains a valid model description. As a convention, for the rest of this document, `mod` will be used as the instantiated model instance. To create a model instance, `mod`, using the model data from the input file `linear1.psc`:

```
>>> mod = pysces.model('linear1')
Assuming extension is .psc
Using model directory: C:\mypysces\pscmodels
C:\mypysces\pscmodels\linear1.psc loading ..... Done.

>>>
```

As illustrated in this example, a string containing the model file is used to instantiate the model object. All model files are assumed to have `.psc` as an extension which can be left out. By default the model directory, as specified in the previous section, is used to locate model files, however, the model name and directory and can also be explicitly supplied to the model object.

```
mod = pysces.model(File='linear1.psc',dir='c:\\mypysces\\pscmodels')
```

This might be especially useful in an application where automated model loading is required. If PySCeS does not find the specified model file, it displays a complete list of all the models in the the current model directory and allows the user another opportunity to input the model name.

```
>>> mod = pysces.model('wrong_name')
Assuming extension is .psc
Models available in your model_dir:
************
branch1.psc       linear1.psc        moiety1.psc
************

You need to specify a valid model file ...

Please enter filename: linear1.psc

Using model directory: /home/bgoli/pscmodels
/home/bgoli/pscmodels/linear1.psc loading ..... Done.
>>>
```

Once instantiated the following attributes are created.

- `mod.ModelOutput`, the default model output directory is initially read from the user configuration file but can be set on a per model basis by changing this path.
- Both the `mod.ModelFile` and `mod.ModelDir` are set when the model object is first instantiated and are provided for the user's convenience.

At this stage, the model object is associated with an input file, but it is not yet usable as the model attributes have not yet been transformed into usable Python instance attributes.

4.2 Loading a PySCeS model

In order to get a usable model object, the model description needs to be parsed from the input file and attached as attributes to the current model instance.

```
>>> mod.doLoad()

Parsing file: C:\mypysces\pscmodels\linear1.psc
>>>
```

Once the model has been loaded all the model file data, including parameters and rate equations, are made available as model attributes. For example, in the input file the following would have been initialized `s1 = 1.0 and k1 = 10.0` and are now available as

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

4.3 Inside the `mod.doLoad()` method

The `doLoad()` method is a meta-function which in turn calls `mod.ParseInputFile()` and `mod.ParseModel()` which together load the model.

4.3.1 `mod.ParseInputFile()`

This method is responsible for parsing a model input file and arranging the information so that PySCeS can build a model out of it. All lexical analysis and parsing is performed using David Beazley's PLY package which uses LR parsing and is closely modelled on the popular lex and yacc tools[6]. The lexer/parser is a derivative of Jannie Hofmeyr's `lexparse.py`. While originally implemented as a script, the PySCeS version of lexparse uses a combination of class method and module functions. This initially led to some magnificent lexing and parser conflicts, especially where, for example, a model was reloaded after the input file had been changed[7]. In order to eliminate potential lexer/parser conflicts `ParseInputFile()` first clears the model instance's namespace dictionary maintaining only essential information such as model name and path, before (re)parsing.

Once the instance namespace has been cleared and necessary information restored, the input file is read, lexically analysed and parsed into a Python *network dictionary* containing all the information needed to create a working model. As a Python dictionary is essentially an unordered set of key-value pairs, before it can be used, the *network dictionary's* information must be ordered into lists of model components which can be added as model attributes, both hidden (for internal use) and visible. These attributes are summarized in Section. 4.4.

Once the model attributes have been attached, redundant references to global or module variables are cleared for garbage collection, reducing the memory footprint of a model instance. Finally, the paths to the MetaTool floating point and integer binaries are set as defined in the main PySCeS configuration file.

[6] See the PLY documentation (http://systems.cs.uchicago.edu/ply/)

[7] It is still possible for this to happen if the parser crashes after being initialized with an improperly formatted input file.

```
>>> mod.eModeExe_dbl
'c:\\python23\\lib\\site-packages\\pysces\\metatool\\meta43_double.exe'

>>> mod.eModeExe_int
'/usr/lib/python2.3/site-packages/pysces/metatool/meta43_int'
```

In summary, a model object has been created, initialized with an input file containing a model description, the model has been parsed and the model properties have been added to the model instance as a set of ordered lists and dictionaries. In the second stage of the loading process, this information has to be initialized and converted into usable Python objects.

4.3.2 `mod.ParseModel()`

The `ParseModel()` method is a collection of methods which manage the run time initializations needed to convert a model description into a model. For the purposes of this discussion 'run time' means 'when the program is run' and although Python is essentially all run time I use it to differentiate from 'write time' which is 'when the code was written'. This is similar to the term compile time used in for languages like C and Java where code compilation is completely separated from code execution.

Using the Python functions `compile()` and `exec()` the code generation methods take full advantage of Python's scripted nature and allow for the efficient run time generation and evaluation of code. An easy way to understand this process is to compare it to what happens when Python is used interactively in console mode. You first type a line of Python code and enter it. Two things now happen, first the interpreter parses the line and converts it into a code object containing all the object references that it needs for evaluation (the `compile()` step). Next, the code object is evaluated by the Python virtual machine and the results returned (the `exec()` step) to the console. Working interactively, these two steps happen sequentially and you only see the returned result, however this same process can be used to dynamically generate run time code in a non-interactive environment. Additionally, run time code can be generated and compiled at one time and executed when needed. This may may also help in optimizing the run time generated code, because a function compiled once can then be directly executed many times. The following private methods are responsible for generating a Python representation of the model.

- `__initmodel__()`: creates the stoichiometric matrix.
- `__initvar__()`: initializes the variable metabolites and derivative functions.
- `__initfixed__()`: initializes the parameter values and array mapping functions.
- `__initREQ__()`: initializes the rate equation functions.

Once the stoichiometric matrix is constructed, the `__initvar__()` and `__initfixed__()` methods write the Python code which, when executed, creates the various model attributes (parameters, metabolites etc.), builds the concentration vector and creates functions that map the model attributes to arrays and vice versa. Mapping functions are critical to the interactive nature of PySCeS as the rate equations are defined in terms of model attributes and these need to be converted into arrays for use with the low-level numerical routines. These methods also construct the various code blocks that are function definitions used by the automatic derivative routines that will be discussed in Section 13.

The metabolite and parameter initialization methods return raw Python (text) code to `mod.ParseModel()` which compiles it, executes the portions of it which create the model properties and makes the rest available to PySCeS as hidden model attributes that can be executed when needed. Once the model properties are set, the rate equation functions can be initialized.

`__initREQ__()` is analogous to the previous methods except that instead of only assigning attributes, it constructs complete Python function definitions. Each rate equation is added to the model in three ways, as a string representation of the rate equation, as a lambda function which can be evaluated using the current parameter and metabolite concentration set and as part of a function definition that is used by the rate equation evaluation functions. Once created the raw code is again returned to `mod.ParseModel()` which compiles and executes it[8]. As all dynamically generated code is added to the model object after instantiation a model's data and methods are completely encapsulated. In this way multiple, independent model instances can be initialized from a single input file.

Finally, a structural analysis is performed (discussed in detail in Section 6.3) and all the various control attributes (algorithm defaults, analysis mode settings etc.) are created and initialized. After the model loading process is completed, we have a fully initialized model object that is ready to be used.

4.4 PySCeS basic model attributes

During the load process, the model elements and parameters read in from the input file are translated into Python objects in the following ways:

- Parameters including external metabolites are attached using their original name e.g. `k1` in the model input file would now be available as `mod.k1`.
- Variable metabolites are treated in the same but an additional attribute is created that represents their initial values. For example a metabolite `s1` would be created as `mod.s1`, the actual value of the metabolite at any point in time and `mod.s1i` its initial value. Initial values can be used to initialize numerical algorithms etc.

[8] Personally, I was absolutely astounded at the efficiency of Python's runtime code generation capabilities and nicknamed it automagic.

- Rate equations are attached as Python lambda functions. For example, for a reaction named `R1` calling `mod.R1(mod)` would return the current reaction rate. Note the model instance must be passed to the function as an argument. Additionally, `mod.R1r`, a string representation of the rate equation, is also created.
- Fixed metabolite names can be displayed using `mod.fixed_metabolites`.
- Parameter names (including fixed metabolites) can be displayed with `mod.parameters`.
- Variable metabolite names and order can be displayed with `mod.metabolites`.
- Reaction names and order can be displayed using `mod.reactions`.
- The floating point precision of the CPU is determined by PySCeS and made available as `mod.mach_floateps`.
- The display format of all numeric results for any model object can be set using `mod.mode_number_format`. The default format is exponential (`'%2.4e'`) but any valid Python format string can be used (for a floating point format use `'%2.4f'`)

With exception of the rate equations and reaction stoichiometry, which can only be changed in the input file, all other model attributes can be set interactively.

5 The kinetic model

Up until this point we have been investigating the construction of a model, first defined in an input file, next instantiated as a PySCeS model object. In the following sections will begin to show how we can use the PySCeS structural analysis methods inherited by the model class from the stoichiometric analysis class `Stoich` to analyse our system. As the implementation is closely coupled to the theory, the two will be developed together.

Any coupled reaction network can be described in terms of a set of non-linear differential equations. These equations contain both the structural information (how the reactions are connected to one another) as well as kinetic information (the dynamics of the conversion processes) themselves [11].

$$\frac{ds}{dt} = \mathbf{N}\mathbf{v}[s, p] \tag{1}$$

Using the newer formalism and notation as given by Hofmeyr in [12] the kinetic model, shown in Eqn. 1, is made up of a matrix containing the stoichiometric coefficients ($\mathbf{N}$), a column vector of reaction rates ($\mathbf{v}$) which is expressed in terms of variable metabolite concentrations ($\mathbf{s}$) and parameters, including fixed or constant metabolites ($\mathbf{p}$).

As an illustration of the kinetic model, as shown in Eqn. 1, consider a metabolic network as earlier described in Fig. **??**. In this system external (or boundary metabolites)

are indicated as $X_0 \ldots X_7$, internal (variable) metabolites are shown as $S_1 \ldots S_3$ and the enzyme catalysed reactions as $E_1 \ldots E_4$. The system in Fig. **??** can be rewritten as a stoichiometric matrix ($\mathbf{N}$, whose construction is described in more detail in the next section):

$$\mathbf{N} = \begin{array}{cccc} E_1 & E_2 & E_3 & E_4 \\ 1 & 0 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{array} \begin{array}{l} \\ S_1 \\ S_2 \\ S_3 \end{array}$$

Each row corresponds to a variable metabolite and each column to a reaction. A value of one means that that the metabolite is a produced by that reaction, while minus one means it is consumed. A zero means that the metabolite is neither a reactant or product of that reaction. The second part of the equation describing the kinetic model is the vector of reaction rates, ($\mathbf{v}$):

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

Using Eqn. 1 the kinetic model can be written as:

$$\frac{d\mathbf{s}}{dt} = \begin{bmatrix} 1 & 0 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

which when multiplied out, yields the individual ordinary differential equation expressing the change in concentration of each of the variable metabolites in terms of net reaction rates, v, each of which is a function of the kinetic parameters and variable metabolite concentrations:

$$\begin{aligned} \frac{dS_1}{dt} &= v_1 - v_3 - v_4 \\ \frac{dS_2}{dt} &= v_2 - v_1 \\ \frac{dS_3}{dt} &= v_1 - v_2 \end{aligned}$$

5.1 The stoichiometric matrix

In this section we investigate the first component of the kinetic model as shown in Eqn. 1, the stoichiometric matrix.

The stoichiometric matrix is an m by n matrix which describes the structure of the reaction network in terms of the coefficients (c_{ij}) of the reactions $\{v_1, v_2, v_3 \ldots, v_n\}$

that make up the differential equations describing the concentrations of the variable metabolites $\{\dot{s}_1, \dot{s}_2, \dot{s}_3 \ldots \dot{s}_m\}$ so that:

$$N = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & & \vdots \\ c_{m1} & \cdots & c_{mn} \end{bmatrix} \tag{2}$$

where as described in [12]:

- $c_{ij} < 0$ where s_i is a substrate of reaction v_j
- $c_{ij} > 0$ where s_i is a product of reaction v_j
- $c_{ij} = 0$ where s_i is neither a substrate nor a product of reaction v_j

Although traditionally integers, PySCeS allows the use of both integer and floating point coefficients.

6 Stoichiometric analysis

There are two structural properties that are contained in the stoichiometric matrix (**N**), both of which deal with the relationship between either the differential equations describing the metabolites or fluxes at steady state [11, 12]. Conserved relationships appear when systems of ODEs are linearly dependent on one another. Linear algebra provides powerful techniques for determining dependencies amongst systems of differential equations [13]. Let us begin by investigating how we can determine the moiety conserved relationships of a system from **N**.

6.1 Moiety conservation – calculating L

The detection of moiety conservation [14] is critical in many aspects of metabolic analysis and modelling, for example,

- in the computation of the steady state [15],
- calculation of control coefficients using the control matrix equation [12, 16, 17]
- and the understanding and reduction of complex reaction networks [18].

Central to the idea of moiety conservation is the *Link* or **L** matrix [19]. If conservation exists the **L** matrix can be partitioned into an identity and zero link matrix, $\mathbf{L_0}$:

$$\mathbf{L} = \begin{bmatrix} \mathbf{I} \\ \mathbf{L_0} \end{bmatrix} \tag{3}$$

There are various ways of finding the moiety conservations present in $\mathbf{N}$ including Gaussian elimination, Gauss-Jordan elimination and Singular Value Decomposition [20] (see [21] for a comparison of various computational techniques). PySCeS uses Gauss-Jordan elimination [13] to determine the conserved relationships between the differential equations. This is a variation of the technique that uses an augmented $\mathbf{N}$ matrix and Gaussian elimination to determine the conservation matrix [22] (for an example see [18]).

In general a Gauss-Jordan elimination works in the column space of a matrix, which in the case of the stoichiometric matrix are the reaction rates. What we need to calculate is the relations between the differential equations. This can be achieved by working in the row space of $\mathbf{N}$, i.e. the column space of $\mathbf{N}^T$. By performing a Gauss-Jordan reduction on $\mathbf{N}^T$ the linear relationships between the differential equations can be obtained. PySCeS calculates $\mathbf{L_0}$ directly from $\mathbf{N}^T$ using the following procedure [21]:

- transpose the input matrix,
- use LU factorization to reduce the system to echelon form,
- scale the resulting pivots to equal one,
- use Gauss-Jordan elimination to produce zeros above the pivots,
- extract $\mathbf{L_0}$ from the LU factorization result,
- build $\mathbf{L}$ from $\mathbf{L_0}$,
- using $\mathbf{L}$ and $\mathbf{L_0}$, compute the conservation matrix and form the reduced stoichiometric matrix.

6.1.1 Gauss-Jordan reduction of N

After transposing the input matrix ($\mathbf{N}$) PySCeS uses the LAPACK routine `DGETRF`[9], provided with SciPy, to perform an LU factorization. The elimination process takes $\mathbf{N}$ and factorizes it so that:

$$\mathbf{N}^T = \mathbf{PLU} \tag{4}$$

where $\mathbf{U}$ is the upper matrix or the echelon form of $\mathbf{N}$. One important factor to keep in mind is that, unlike many linear algebra applications, it is vitally important to keep track of any changes in the order of the rows and or columns (in this case only the columns). `DGETRF` is a partially pivoting algorithm which almost always maintains a small growth factor [23]. Unfortunately, partial pivoting also means that if a zero (singular value) is encountered in a pivot position (possibly as a result of row interchanges) during the factorization, the algorithm terminates. As stoichiometric matrices are generally not dense matrices, this situation is a distinct possibility and can lead to premature termination of the algorithm.

[9] http://www.netlib.org/

By wrapping `DGETRF` in a Python routine that can restart the algorithm if necessary and at the same time swap columns to generate a 'missing' pivot, we essentially wrap the partial pivoting algorithm and create a 'pseudo' full pivoting routine. Additionally, after every iteration zero rows are swapped to the bottom of the matrix and removed, thus reducing the overall matrix size as redundancies are detected. As PySCeS uses floating point arithmetic, care must also be taken to remove any floating point artifacts that might be a created during the reduction process.

In general, while being relatively resistant to numerical error [24, 25], using a pivoting approach does have the disadvantage that $\mathbf{N}$ is reordered as columns are swapped and it is therefore difficult to predict the final structure of $\mathbf{U}$ from $\mathbf{N}$. On the other hand, it does have the computational advantage that the resulting matrix is in perfect staircase form, with pivots only on the diagonal and no redundant rows of zeros.

$$echelon(\mathbf{N}^T) = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ & \cdot & \cdot & \cdot & \cdot \\ & & \cdot & \cdot & \cdot \end{bmatrix} \tag{5}$$

Having the factorized matrix in this form greatly simplifies any subsequent scaling operations.

The final step in the Gauss-Jordan elimination is to eliminate any values above the pivots by a process of back substitution; after this process has completed we have the reduced form of $\mathbf{N}$:

$$reduced(\mathbf{N}^T) = \begin{bmatrix} 1 & & & \cdot & \cdot \\ & 1 & & \cdot & \cdot \\ & & 1 & \cdot & \cdot \end{bmatrix} \tag{6}$$

6.1.2 Calculating $\mathbf{L_o}$ and $\mathbf{N_R}$

As shown in [21] the reduced form matrix shown in Eqn. 6 can generally be written as

$$reduced(\mathbf{N}^T) = \begin{bmatrix} \mathbf{I} & \mathbf{F} \\ 0 & 0 \end{bmatrix} \tag{7}$$

and that when formulated in this way

$$\mathbf{F}^T = \mathbf{L_o}$$

Having calculated $\mathbf{L_o}$, PySCeS can build $\mathbf{L}$ (using Eqn. 3) as well as construct the conservation matrix γ which is also defined in [21].

$$\gamma = [-\mathbf{L_o} \;\; \mathbf{I}]$$

The conservation matrix is useful for calculating the moiety totals (the vector $\mathbf{T}$) automatically from the initial concentrations of the variable metabolite (the concentration vector $\mathbf{S}$) by using the relationship

$$\gamma\mathbf{S} = \mathbf{T}$$

From **L** we can tell which of **N**'s differential equations are redundant; removing them forms an important new matrix, whose rows are linearly independent, the reduced stoichiometric matrix, $\mathbf{N_R}$. Now that we have determined one of the structural properties of our system we can move on to determining the next one, the relationship between the fluxes at steady state.

6.2 Flux conservation – calculating K

The second structural property of the stoichiometric matrix concerns the dependencies between the fluxes at steady state. These flux relationships are important for the calculation of the control coefficients using the control-matrix equation [12] and are expressed as the *Kernel* matrix, **K**, where:

$$\mathbf{K} = \begin{bmatrix} \mathbf{I} \\ \mathbf{K_0} \end{bmatrix} \tag{8}$$

In order for a metabolic system to be at steady state, the variable metabolite pools in the system need to be constant in time or:

$$\frac{d\mathbf{s}}{dt} = 0$$

If we apply this to our kinetic model we can rewrite Eqn. 1 as shown in [12]:

$$\mathbf{N_R v}[s_i, s_d, p] = 0 \tag{9}$$

or for the situation where there is no conservation

$$\mathbf{N v}[s, p] = 0 \tag{10}$$

The theory of linear algebra (see [13]) shows us that for any set of equations that have the form:

$$\mathbf{A x} = 0$$

the nullspace of **A** (if it exists) shows us the linear dependencies amongst the columns of **A** (in our case the fluxes). Fortunately, calculating the null space of **N** is almost identical to the procedure used to calculate $\mathbf{L_0}$ (as described in Section. 6.1).

From a programmer's perspective this is convenient as the same basic functions can be reused with a few minor variations in the routine. To determine the flux relationships, Gauss-Jordan reduction is now performed on **N** and $\mathbf{K_0}$ is extracted. After the reduction of **N** to its row reduced form:

$$reduced(\mathbf{N}) = [\mathbf{I} \;\; \mathbf{F}]$$

The basis for the nullspace (**F**) can be extracted and transformed into $\mathbf{K_0}$ using the following relationship:

$$\mathbf{K_0} = -\mathbf{F}$$

which follows directly from the definitions of the **K** matrix (Eqn. 8) and null space which is defined as:

$$null(\mathbf{A}) = \begin{bmatrix} \mathbf{I} \\ -\mathbf{F} \end{bmatrix}$$

This completes our analysis of the structural properties of the stoichiometric matrix. The entire stoichiometric analysis is automatically done by PySCeS when the `mod.doLoad()` is called and all the structural properties are attached as model attributes which are shown in the next section.

6.3 PySCeS structural attributes

Once the model is loaded and the stoichiometric analysis is complete, the following structural attributes are made available to be used or displayed.

6.3.1 N and $\mathbf{N_R}$

The following attributes represent **N** and $\mathbf{N_R}$ as numeric arrays that can be used for further calculations. PySCeS matrix attributes each have an associated row or column vector. These lists are expressed as relative to the rows or columns of the stoichiometric matrix. All the `mod.showX()` methods listed here accept an open file object as an argument in which case they write to the file and not to the screen.

- `mod.showN(File=None)` print **N** to screen including row and column labels
- `mod.nmatrix` – a SciPy array containing the stoichiometric matrix, **N**.
- `mod.nmatrix_row` – an index array representing the differential equations describing $\frac{ds}{dt}$, i.e. the rows of the stoichiometric matrix.
- `mod.nmatrix_col` – an index array representing the model's reactions, i.e. the columns of the stoichiometric matrix.
- `mod.showNr(File=None)` print $\mathbf{N_R}$ to screen including row and column labels
- `mod.nrmatrix` – the reduced stoichiometric matrix, $\mathbf{N_R}$ containing only the independent differential equations.
- `mod.nrmatrix_row` – independent differential equations describing $\frac{ds}{dt}$.
- `mod.nrmatrix_col` – the model's catalytic reactions $\mathbf{N_R}$.

6.3.2 L, L_o and conservation matrices

The properties related to moiety conservation can also be grouped together as follows.

- `mod.showL(File=None)` print **L** to screen with rows and columns, if no conservation is present a warning message will be printed instead.
- `mod.lmatrix` – link matrix, **L**. If there is no moiety conservation in the system this is an identity matrix.
- `mod.lmatrix_row` – the **L** row vector contains metabolites partitioned into independent and dependent.
- `mod.lmatrix_col` – the **L** column vector contains the independent metabolites.
- `mod.lzeromatrix` – the link zero matrix, $\mathbf{L_o}$. This matrix is an identity matrix if there is no moiety conservation present in the model.
- `mod.lzeromatrix_row` – the $\mathbf{L_o}$ rows contain the dependent metabolites as linear combinations of the independent ones.
- `mod.lzeromatrix_col` – the $\mathbf{L_o}$ columns contain the independent metabolites.

As we have seen, once the **L** has been determined the conservation matrix and moiety total vector can be formed.

- `mod.showConserved(File=None)` – print a human readable representation of the conservation matrix e.g. `+ {1.00}S2 + {1.00}S3 = 1.3\n`
- `mod.lconsmatrix` – the conservation matrix, γ
- `mod.lconsmatrix_row` – the dependent metabolites.
- `mod.lconsmatrix_col` – the system metabolites.
- `mod.Tvec` – the moiety total vector, **T** which is calculated from the initial concentrations of the variable metabolites. `Tvec` is updated when either the steady state solver or simulation method is called.

6.3.3 K and K_o

As in the case of moiety conservation, the relationships between the fluxes at steady state are available as:

- `mod.showK(File=None)` print **K** to screen with rows and columns.
- `mod.kmatrix` – a SciPy array containing the the **K** shows the relationship between the dependent and independent fluxes.

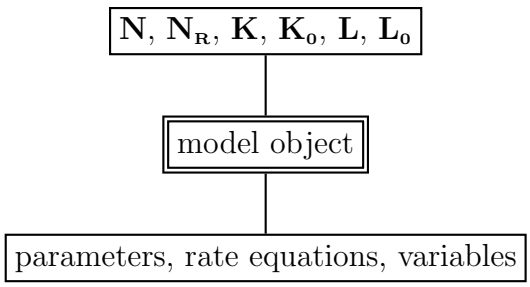

Fig. 2 *A PySCeS model object.* When a model object is initially created using the `doLoad()` method it has attributes representing both the basic and structural properties of the model

- `mod.kmatrix_row` – fluxes ordered with independent fluxes first followed by dependent ones.
- `mod.kmatrix_col` – independent fluxes.
- `mod.kzeromatrix` – the dependent fluxes as linear combinations of the independent ones.
- `mod.kzeromatrix_row` – array containing the dependent fluxes.
- `mod.kzeromatrix_col` – array containing the independent fluxes.

7 The Tao of PySCeS: Part 2

"An algorithm must be seen to be believed." – Donald E. Knuth

Taking heart from Donald Knuth's observation we shall now proceed to investigate various other aspects of our model object (a schematic of which is shown in Fig. 2). PySCeS provides a number of different analysis methods that can be applied to the basic model object which will again be described in terms of the theory behind the analysis, how PySCeS implements the theory and how the user can access the attributes, parameters and methods related to the analysis.

The most general of these methods is the `doSomething()` methods, which are a collection of high level methods that calculate amongst other things:

- a steady state: `mod.doState()`,
- elasticity coefficients: `mod.doElas()`,

These high level methods allow PySCeS to be used interactively without the user having to type too many commands; however, each of these methods is made up of lower level routines which may be used and customized individually. For example the `doElas()` method shown above is made up of the following subroutines:

- `mod.State()`: calculate the steady state,
- `mod.EvalEvar()`: calculate elasticities to variable metabolites,
- `mod.EvalEpar()` calculate elasticities to parameters.

This design philosophy can be summarized as *providing high level modules with low level access* and in this way PySCeS tries to be both 'easy to use' and 'flexible'. In the sections that follow we will describe various analyses which can be performed on the basic model object.

8 Calculating elementary flux modes

In the previous sections we have seen how by analysing the stoichiometry of a metabolic pathway we can derive specific properties of a metabolic reaction network from its structure. In this section we show how PySCeS can be used for another type of stoichiometric analysis, namely calculating the elementary flux modes [26, 27]. In Fig. 3 we see how the elementary flux modes can describe the sets of enzymes that are needed to convert one fixed metabolite to another. For example in Fig. 3, we see that assuming all reactions are reversible, Y can be created either from X, via reactions 1,2,3,4 (elementary mode a) or from Z, via reactions 6,5,3,4 (elementary mode c). This description of elementary flux modes is based on one given in [28]. More specifically, elementary flux modes are minimal sets of enzymes that can each generate a steady state, taking into account the possible irreversibility of reactions. An elementary flux mode cannot be decomposed further and any steady state flux is a non-negative combination of elementary flux modes.

Elementary modes have been extensively used to determine biologically meaningful relationships amongst the steady state fluxes [26, 29–31]. An analogous procedure has also been proposed for the determination of 'metabolically meaningful pools' [32].

8.1 Using MetaTool with PySCeS

In order to calculate the elementary flux modes PySCeS incorporates an interface to a stand-alone application: MetaTool [4]. During the PySCeS installation process two versions of the MetaTool executable[10] are compiled, one for models with a purely integer stoichiometry and one for models which contain floating point coefficients.

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

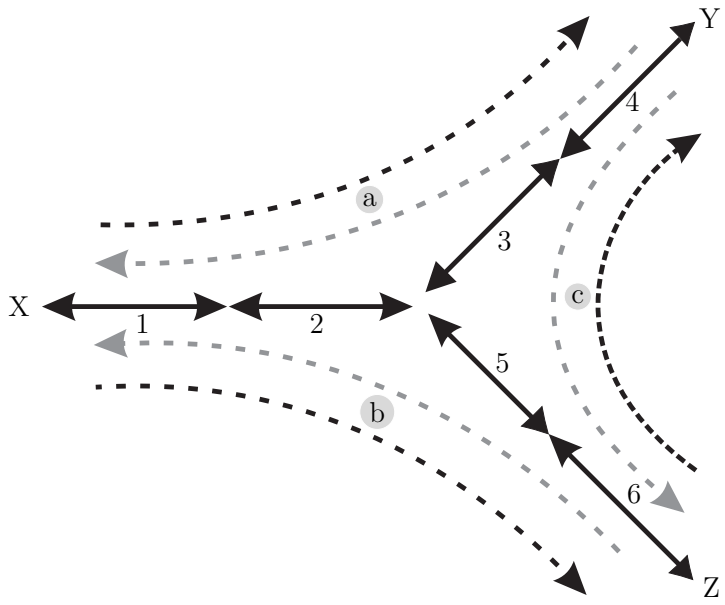

Fig. 3 *Elementary flux modes.* The solid lines represent six reversible enzyme catalysed reactions that form a branched metabolic pathway. In this system there are 6 elementary modes shown as three pairs a–c of black and grey dashed lines. This system is based on a similar one described in [28].

As a stand-alone application MetaTool works by reading in a user defined input file and writing the results to a data file. PySCeS simulates this procedure by first generating a MetaTool input file and then calling the relevant executable to process it. By default, PySCeS uses the integer executable but if a non-integer coefficient is detected in the stoichiometric matrix PySCeS automatically switches to the floating point version. Once MetaTool has finished running and has generated a result file, PySCeS parses it and stores the resulting elementary modes. The following are the commands and settings that can be used to calculate the elementary modes.

- `mod.doModes()`: calculate the elementary flux modes.
- `mod.showModes(File=None)`: display the calculated modes to screen or to an open file object if one is supplied as an argument.
- `mod.emode_intmode`: defines which executable should be used for calculations, setting this option to one (default is zero) forces the use of the floating point binary.
- `mod.emode_userout`: with the default setting (zero) all intermediary files generated by MetaTool are deleted, if this option is set to one, both the input and output file are saved in the PySCeS working directory.

- `mod.emode_file`: the filename of the MetaTool intermediate files. The default is to use `<model_name> + _emodes`.

9 Evaluating the differential equations

Analysing the kinetic model whether dynamically or at steady steady state involves solving the differential equations as expressed in Eqn. 9. Note that this form of the equation uses only the reduced stoichiometric matrix ($\mathbf{N_R}$) and so implicitly is expressed only in terms of the independent differential equations and concentrations.

PySCeS implements this equation directly so that all numerical algorithms work only with the reduced (linearly independent) set of differential equations. Although this does not influence the workings of the integration routines it is of critical importance in the calculation of the steady state. This is due to the fact that if a numerical solution is sought to a model described by a set of differential equations that are not linearly independent (the model contains moiety conservation), non-linear solvers tend to 'drift' and do not converge to a solution. The problem that arises when implementing Eqn. 9

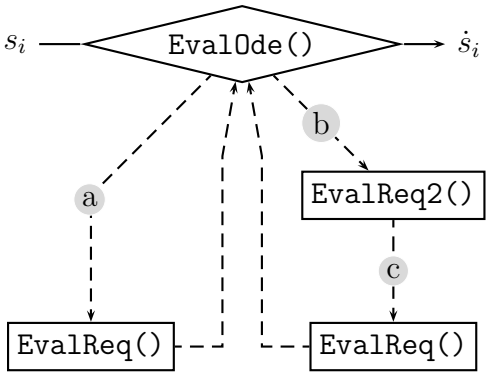

Fig. 4 *Evaluating the differential equations.* If there is no moiety conservation `EvalOde()` uses only `EvalReq()`, (a) while when there is moiety conservation `EvalReq2()`, (b) and `EvalReq()`, (c) are used to determine $\dot{s}_i$ from a concentrations vector s_i

in terms of only the independent differential equations is that the rate equations which make up the terms of the differential equations are expressed in terms of both dependent and independent concentrations. In systems where no moiety conservation exists this is not a problem, as all the differential equations and concentrations are independent, in fact the simpler form of Eqn. 9 namely $\mathbf{Nv}$ can be used directly as $\mathbf{s} = \mathbf{s}_i$, as is shown in Fig. 4 (a). On the other hand, for systems where moiety conservation does exist an additional step is needed to first calculate $\mathbf{s}$ from $\mathbf{s}_i$ before the rate equations are evaluated. This is shown in Fig. 4 (b) and (c).

The following list details the individual methods used to evaluate the differential equations (in all cases Vtemp is a storage array used to hold a rate vector).

- `EvalODE(s, Vtemp)` calls `EvalReq2()` if there is moiety conservation and evaluates $\mathbf{N_R v}$; otherwise it uses `EvalReq()` and evaluates $\mathbf{Nv}$ to return $\dot{s}_i$.
- `EvalREq2(s, Vtemp)` uses the relationship $\mathbf{s_D} = \mathbf{L_0} + \mathbf{T}$ to calculate the full concentration vector and calls `EvalREq()`.
- `EvalREq(s, Vtemp)` uses the concentration vector to evaluate the rate equations and returns a rate vector.

PySCeS also makes provision for the use of so called forcing functions that are commonly used to represent equilibrium blocks, monitoring functions and other fixed relationships in reaction networks. Generally, these extra functions are not defined as rate equations but need to be evaluated at the same time as a rate equation evaluation takes place. PySCeS provides an empty method definition, `mod.Forcing_Function()`, which is called each time the rate equations are evaluated. This empty method can however be replaced by a Python function that contains any user defined forcing functions (this function should neither take any arguments nor return a value and only operate on instance attributes), as illustrated in the following example.

```
>>> mod = pysces.model('chemostat')
>>> mod.doLoad()

>>> def chemostat_forcing_function():
        mod.Vi = mod.alpha_V*mod.X
        mod.Vo = mod.Vt - mod.alpha_V*mod.X

>>> mod.Forcing_Function = chemostat_forcing_function

>>> mod.doState()
```

10 Time simulation

In order to study a system's evolution over time it is necessary to integrate the kinetic model. PySCeS uses the Livermore Solver for Ordinary Differential Equations with Automatic method switching for stiff and non-stiff problems: LSODA [33, 34]. LSODA[11] is a powerful integration routine, accessible via the following high level simulation methods:

- `mod.doSim(end=10.0, points=20.0)`: run a simulation from time zero to time `end` using the specified number of `points`.
- `mod.doSimPlot(end=10.0, points=20.0, plot='met')` run a simulation as above but now `plot` the metabolite concentrations (the plot syntax is explained as part of the `SimPlot()` command which can be used for plotting the results of a simulation)

[11] http://www.netlib.org/

Once again, these high level methods utilize lower level routines which can be used and configured individually if necessary:

- `mod.Simulate(userinit=0)`: the core simulation routine.
- `mod.sim_end`: simulation end time.
- `mod.sim_points`: number of points in the simulation.
- `mod.mode_sim_init`: defaults to zero and the initial concentrations are used as initial values for the `Simulation`. A value of one initializes the variables with a small (almost zero) value.
- `mod.sim_time`: the time array calculated using the user defined time interval and number of points
- `mod.SimSet()`: method that creates the time array. Except as described below this method is automatically called by `Simulate()` and does not need to be explicitly called.

By default `mod.Simulate()` uses the value of `mod.mode_sim_init` to determine what it should use as its starting values. If a model's initial metabolite values are used (e.g. `mod.S1i`) then they are automatically collected and placed into an array, `mod.s_init`, which is used in turn, to initialize LSODA. It is however possible to override this behaviour by setting `mod.Simulate()`'s `userinit` argument:

- userinit = 0: (default) generate both `mod.sim_time` and `mod.s_init`
- userinit = 1: generate `mod.s_init` but not `mod.sim_time`. This allows `mod.sim_time` to be predefined by the user ... perhaps a logarithmic timescale?
- userinit = 2: use user defined values of `mod.s_init` and `mod.sim_time`, the only check performed is the length of the initialization vector.

Please note using `userinit = 2` can be dangerous and is not recommended for general modelling purposes! The reason is that, with each simulation the moiety conserved totals are automatically recalculated from the metabolite's initial values (e.g. `mod.S1i`). A user supplied initialization vector bypasses this calculation so it is therefore possible to break the moiety conserved cycles present in a system. If this happens, biologically meaningless results can be produced. One application where this type of initiation could be used with relative safely is where, assuming the moiety conserved totals remain unchanged, a simulation is directly initialized with a previously calculated steady-state concentration vector. This can be used to effectively automate a study of the transient changes that occur between steady states through a series of parameter changes.

So far we have been looking at the initialization of the LSODA routine. There are, however, a number of parameters[12] that control the operation of the algorithm itself (a value of zero means the algorithm determines the parameter value).

- `mod.LSODA(initial)`: the basic interface to LSODA called by `Simulate()`. Provided with a set of initial values, it returns an array of solutions and a status flag. As this method 'plugs' into 'Simulate()', it makes provision for the future addition of different integration algorithms using a standard framework.
- `mod.lsoda_mxstep` (default = zero): maximum number of internally defined steps. By default LSODA auto-adjusts this parameter as necessary (to a maximum of 500) but in systems that require more integrator steps between time points, the `mod.Simulate()` method automatically tries to adjust this parameter to a larger value and reruns the simulation.
- `mod.lsoda_atol` (default = 10^{-10}): absolute tolerance.
- `mod.lsoda_rtol` (default = 10^{-5}): relative tolerance.
- `mod.lsoda_h0` (default = 0.0): the step size to be attempted on the first step.
- `mod.lsoda_hmax` (default = 0.0): the maximum absolute step size allowed.
- `mod.lsoda_hmin` (default = 0.0): the minimum absolute step size allowed.
- `mod.lsoda_mxordn` (default = 12): maximum order to be allowed for the nonstiff (Adams) method.
- `mod.lsoda_mxords` (default = 5): maximum order to be allowed for the stiff (BDF) method.
- `mod.lsoda_mesg` (default = 1): print the exit status message.

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

```
mod.Fix_Sim(mod.sim_res,flux=0)
```

The `mod.Fix_Sim()` method outputs an array with time as the first column followed by either the metabolite concentrations (no second argument or `flux = 0`) or reaction rates (second argument `flux = 1`) whose order is given by `mod.metabolites` or `mod.reactions` respectively. Once a simulation has been completed a quick way to visualize the results is to use the `mod.SimPlot()` method:

[12]Parameter names and descriptions are based on those found in the LSODA source code.

```
mod.SimPlot(plot='met',filename='',title='title',logx='',logy='',cmdout=0)
```

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

- `plot` can be: `'met'` all metabolites or `rate` all rates, or a user supplied list of model variables: `['s1','s2','R3','R5']`. This argument can also be passed directly to the `mod.doSimPlot()` method mentioned earlier.
- If a filename argument is provided, `mod.SimPlot()` will in addition to displaying the result on the screen, try to write a PNG image of the plot to a PNG file named *filename.png*
- By default `mod.SimPlot()` generates a title for the plot using the model file name plus the time the plot is generated. Custom titles can be set with `title = 'mytitle'` argument.
- If `logx` and `logy` are given a string value (`logx = 'on'`) the respective axis is displayed using a logarithmic scale.
- If the `cmdout` argument (default = 0) is set to one, GnuPlot is not called and the SciPy plotting string that would have been used is returned in its place.

These methods allow us to study a systems dynamical behaviour in time. In the next sections we will see how PySCeS enables us to calculate and study the steady-state (or time invariant) properties of the system.

11 Solving for the steady state

Conceptually, calculation of the steady state is easy: for any kinetic model as shown in Eqn. 1 the steady state condition assumes that there is no change in the metabolite pools in time so that

$$\frac{ds}{dt} = 0$$

and the model reduces to a set of non-linear algebraic equations as in Eqn. 9. All that we need to do to determine the steady-state solution is to find the roots of this equation. Unfortunately, it is in calculating the steady state where the general non-linearity of biological systems becomes apparent. This coupled with the fact that all non-linear solvers have to be given an initial estimate of the final solution to work from (which is usually not available) makes it challenging to find a 'one size fits all' non-linear solver.

There are a variety of non-linear solvers, each with their own strengths and weaknesses, which are able to numerically approximate the solutions to highly non-linear systems of algebraic equations, PySCeS uses three: HYBRD, NLEQ2 and forward integration. Before looking at the different solvers in more detail, let's look at the basic options and method used to calculate the steady state.

- `mod.doState()`: calculate a steady state solution
- `showState(File=None)` displays the current steady-state values of the metabolites and fluxes.
- `mod.mode_state_init` (default = 0): This option causes the selected solver algorithm to be initialized with either the initial metabolite concentrations (0), a small value (1), the final value obtained by a quick time simulation (2) or a previously calculated steady-state solution multiplied by a factor (3).
- `mod.zero_val` (default = 10^{-8}) is the small value used with `mod.mode_state_init = 1`
- `mod.mode_state_init2_array`: time array used for the mini simulation when using `mod.mode_state_init = 2`. By default the range is set to `scipy.logspace(0,5,18)`.
- `mod.mode_state_init3_factor` (default = 0.1) the factor used to scale the previous steady state when `mod.mode_state_init = 3`
- `mod.mode_state_mesg` (default = 1): print an exit status message when a steady state is successfully calculated.

Once a steady state has been solved for, the results are accessible via the following model attributes:

- `mod.state_metab`: a vector containing the steady-state concentrations in the same order as `mod.metabolites`.
- `mod.state_flux`: a vector containing the steady-state fluxes stored in the order given by `mod.reactions`.
- For each reaction (e.g. `R2`) a new attribute (e.g. `mod.JR2`) is created containing its steady-state flux value.
- Similarly, for each variable metabolite (e.g. `mod.s2`) an attribute containing its steady-state value (e.g. `mod.s2ss`) is created.

Now that we have looked at the generic aspects of the steady-state interface, let's investigate the three underlying non-linear solver algorithms.

11.1 PySCeS steady-state solver algorithms

PySCeS has been equipped with a pluggable solver framework, which means that each of the steady-state solver routines is wrapped in a standard interface that can be used by the `doState()` method. This allows a measure of flexibility in how the individual algorithms can be combined to obtain a steady-state solution. Currently, one such combination is available, namely 'solver fallback'.

If active, the fallback algorithm checks both the error messages returned by the individual algorithms and the validity of the solution (i.e. it checks for negative concentrations). If either test fails, it ignores the erroneous result and tries the next solver in the fallback chain. Fig. 5 shows the arrangement of the solvers in the PySCeS fallback chain. Note that the same set of initial values is used for a steady-state calculation irrespective of which solver is used and that the user is explicitly informed when fallback switches from one solver to the next. By doing this, the user is always able to see exactly which solver is finding a steady-state solution for a particular model. This is useful as fallback can also be disabled and PySCeS instructed to use only the optimal solver for the system being studied. If, on the other hand, PySCeS cannot find a solution with any of the algorithms, an error message is generated. As NLEQ2 is an external library subject to specific licence conditions it may either not be installed or disabled, in which case fallback moves directly from HYBRD, Fig. 5 a to FINTSLV, Fig. 5 c.

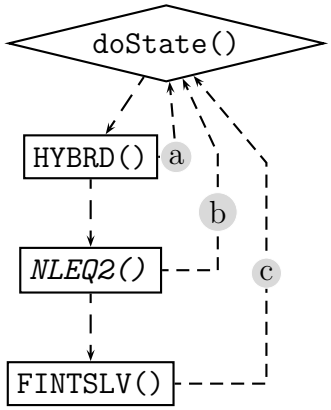

Fig. 5 *Steady-state solvers in a fallback configuration.* Three different algorithms are available to solve for a steady state where the dashed lines indicate optional routes to a solution. a HYBRD, b NLEQ2 and c FINTSLV. As NLEQ2 is an external solver it can be disabled and is therefore shown in italics.

- `mod.mode_solver` (default = 0): this option determines which solver PySCeS should use. HYBRD (0), NLEQ2 (2) or FINTSLV (1).
- `mod.mode_solver_fallback` (default = 1): activate (1) or deactivate (0) solver fallback.

- `mod.mode_solver_fallback_integration` (default = 1): this is a 'paranoid designer' option which either enables (1) or disables (0) the use of FINTSLV in the fallback chain.

Although 'solver fallback' is currently the only strategy implemented in PySCeS (more advanced ones involving simulations and a combination of solvers are being planned) it has been shown to be effective when tested with a selection of numerically sensitive models. This is in part due to the complementary (but different) algorithms employed by the various non-linear solvers, which will be discussed in the remainder of this section.

11.1.1 HYBRD

The first algorithm used by PySCeS is distributed as part of SciPy where it is available as `scipy.optimize.fsolve()`. `Fsolve` is a wrapper for the MINPACK function HYBRD and HYBRDJ, the only difference being that HYBRDJ needs a user supplied Jacobian while HYBRD calculates one by a forward-difference approximation [35] .

HYBRD uses a modified Hybrid Powell method [36] and generally is able to converge quickly even when the initial estimation of the solution is far from the actual solution [37]. PySCeS includes the following options relating to HYBRD[13].

- `mod.HYBRD(initial)`: the HYBRD interface. Takes an initial array as an argument and returns a solution array and status flag.
- `mod.hybrd_xtol` (default = 10^{-12}): the relative error tolerance
- `mod.hybrd_maxfev` (default = 0): maximum number of calls, the default (0) means $100 \times$ (number of metabolites) + 1
- `mod.hybrd_epsfcn`: A suitable step length for the forward-difference approximation of the Jacobian, defaults to the current machine floating point precision
- `mod.hybrd_factor` (default = 100): A parameter determining the initial step bound in interval (0.1,100).
- `mod.hybrd_mesg` (default = 1): switch off (0) or on (1) printing an exit status message.

By inspection, we found that HYBRD manages to find a solution to the majority of our test systems. However, we did find that with numerically sensitive systems it converged to an invalid solution or failed to find a solution. This was noticeable especially when terms in the rate equations are raised to a negative power. This also appears to happen when HYBRD is initialized with values close to the final solution. In some of these situations modifying the algorithms parameters led to a solution, but this was not considered to be a practical alternative for normal modelling situations.

[13] Parameter names and descriptions are based on those found in the HYBRD source code (hybrd.f) available from http://www.netlib.org/minpack/

11.1.2 NLEQ2

In order to overcome HYBRD's limitations it was decided to include an alternate non-linear solver into PySCeS. One algorithm that looked promising but was not GPL'd was the Konrad-Zuse-Zentrum fuer Informationstechnik Berlin's (ZIB) NLEQ2[14]. After contacting ZIB they agreed to allow NLEQ2 to be distributed as source code with PySCeS, under the terms of their non-commercial licence (see Appendix **??** for details). Subsequently, NLEQ2 is included with PySCeS as an optional solver that can be disabled if necessary.

Another advantage of using NLEQ2 is that as it is written in Fortran which meant it could be wrapped and compiled as a Python extension library using F2PY [5]. Once an F2PY wrapper (included in Appendix **??**) had been generated it could then be automatically compiled and installed using the SciPy DistUtils extensions. The process of creating extension libraries using F2PY will be discussed in Section 12.

NLEQ2 is one of a family of algorithms including NLEQ1 and NLEQ1S which are based on Deuflhard's affine invariant damped Newton techniques [38, 39]. In addition to the damping strategies employed by NLEQ1, NLEQ2 incorporates algorithms which serve to extend the convergence domain of the algorithm [40]. NLEQ2 is described as being designed for highly non-linear and numerically sensitive problems and can be customized using the following parameters[15].

- `mod.NLEQ2(initial)`: the interface to the NLEQ2 algorithm. It takes an initial guess array as an argument and returns a solution array and status flag. The NLEQ2 library interface can be accessed directly by calling `pysces.nleq2.nleq2()`.
- `mod.nleq2_iter` (default = 2): the number of iterations to loop the solver through. The default value should be sufficient for most applications.
- `mod.nleq2_rtol` (default = 10^{-8}): the initial relative error tolerance.
- `mod.nleq2_jacgen` (default = 2): Method of Jacobian generation, user supplied Jacobian (1), not supported, numerical differentiation (0 and 2), numerical differentiation with feedback control (3).
- `mod.nleq2_iscal` (default = 0): the lower threshold of the scaling vector is a user defined vector (0) or an internally defined scaling vector (1).
- `mod.nleq2_mprerr` (default = 1): NLEQ2's internal level of user output, no output (0), error messages only (1), error messages and warnings (2), errors and warnings with extra information (3).

[14] http://www.zib.de/SciSoft/ANT/nleq2.en.html

[15] The names and descriptions of the following parameters are based on those found in the NLEQ2 source code (nleq2.f) available for download from http://elib.zib.de/pub/elib/codelib/nleq2/

- `mod.nleq2_nonlin` (default = 4): the non-linearity of the system, linear (1), mildly non-linear (2), highly non-linear (3), extremely non-linear(4).
- `mod.nleq2_qrank1` (default = 0): Rank-1 updates by Broyden-approximation are not allowed (0) or allowed (1).
- `mod.nleq2_qnscal` (default = 0): Automatic row scaling is active (0) or inactive (1).
- `mod.nleq2_ibdamp` (default = 0): bounded damping strategy is, automatic and dependent on the non-linearity of the system (0), always on (1), disabled (2).
- `mod.nleq2_iormon` (default = 0): convergence order monitor. Convergence order is not checked and the algorithm proceeds until the error equals RTOL or an error occurs (1), use weak stop criterion – convergence order is monitored and may terminate if slowdown occurs (0 and 2), use additional hard stop criterion – algorithm may terminate due to superlinear convergence slowdown (3).
- `mod.nleq2_mesg` (default = 1): print the exit status message.

PySCeS implements NLEQ2 as an iterative solver in the sense that NLEQ2 is run in a loop. Although this marginally decreases the performance of the solver, NLEQ2 makes use of scaling vectors and work arrays to optimize many of its parameters when it is executed. By feeding the returned work arrays and scaling vectors from the first iteration back into NLEQ2 as input for a second iteration, NLEQ2 effectively optimizes itself for the problem under consideration.

This preconditioning might be responsible for our limited observation that NLEQ2 does solve most of our numerically sensitive test problems when HYBRD fails. In contrast to HYBRD, NLEQ2 seems to work better when given an initial estimate that is close to the final solution. These two properties make NLEQ2 an excellent algorithm to use in conjunction with HYBRD. Of course the situation might arise where both HYBRD and NLEQ2 fail to find a solution and in this case PySCeS switches over to the final steady-state solver – FINTSLV.

11.1.3 FINTSLV

The final 'steady-state' solver included with PySCeS is the forward integration solver or FINTSLV. This algorithm is not a non-linear solver in itself but instead uses the functional definition of the steady state

$$\frac{d\mathbf{s}}{dt} = 0$$

to approximate a steady solution using integration (i.e. LSODA). To do this PySCeS integrates the kinetic model over a 'long' time period and tracks the changes in the

metabolite concentrations over time. If at any point, the maximum rate of change amongst the metabolite concentrations falls below a defined threshold then the system is assumed to be in steady state. The following parameters control the operation of this algorithm.

- `mod.FINTSLV(initial)`: the algorithm interface. Given an initial guess returns the steady-state solution and status flag.
- `mod.fintslv_tol` (default = 10^{-3}): the threshold deviation used to check for a steady state.
- `mod.fintslv_step` (default = 5): FINTSLV works by comparing the maximum difference between the variable metabolite concentrations, calculated for successive time points. For each step where the maximum difference is below a threshold value, defined by `mod.fintslv_tol`, a 'point' is added to a counter. If this counter reaches the value defined by `mod.fintslv_step`, the system is judged to be in steady state.
- `mod.fintslv_range`: the range[16] over which to integrate.
- `mod.fintslv_rmult` (default = 1.0): the integration range multiplier. This multiplier can easily be used to scale the time range used for the integration.

By default PySCeS switches to FINTSLV when both HYBRD/NLEQ2 have failed; therefore, the default parameters for FINTLSV have been chosen for accuracy rather than performance reasons. For example, the deviation threshold is set to a small value to maximize the algorithm's accuracy. Additionally, the step size used in the time range has been chosen to be large, but not so large that LSODA cannot find a solution from one time point to the next. In keeping with the general PySCeS philosophy, the range and scaling factors can be customized for a particular system by the end user.

11.2 Summary

This section has shown how by using a fallback solver configuration the strengths of both HYBRD, which allows for bad initial guesses and converges quickly to a solution, and NLEQ2, which is more robust and better at numerically sensitive problems, can be leveraged to form a flexible combination of non-linear solvers. Together the two non-linear solvers, combined with the slower but less sensitive FINTSLV, give PySCeS a powerful set of tools to determine the steady-state solution of a system.

[16] [1, 10, 100, 1000, 5000, 10000, 50000, 50100, 50200, 50300, 50400, 50500, 50600, 50700, 50800, 50850, 50900, 50950, 51000]

12 Continuation using PITCON

In the previous section we have seen how we can calculate a steady state solution for a kinetic model thereby solving the equation:

$$\mathbf{Nv} = 0$$

There is however no reason to assume that there should only be one possible solution to this equation. When more than one solution is possible i.e. there is more than one steady state for a particular set of parameters, it allows for the possibility of switching or hysteretic behaviour, as can be seen in Fig. 6 (the PySCeS code used to generate this figure is given in Appendix **??**). This type of behaviour will be looked at in more detail in Chapter **??**.

In the rest of this section we will be dealing exclusively with multiple steady-state solutions or static bifurcations that can be calculated using parameter continuation methods (see [19, 42]). Two widely used techniques for investigating systems that exhibit multiple solution are homotopy and continuation methods [42]. As in the case of NLEQ2 we were looking for a public domain algorithm, written in Fortran which would allow us to generate a Python extension library using F2PY. One algorithm seemed to satisfy all our criteria – the University of Pittsburgh continuation program (PITCON) [43–45]. The PITCON algorithm (also known as CONTIN) is freely available from Netlib[17] while a newer Fortran 90 version can be downloaded from the author's web-site[18].

12.1 Generating interfaces to Fortran libraries using F2PY

The Fortran to Python Interface Generator (F2PY)[19] is an open source utility that can automatically generate Python extension libraries from source code written in the Fortran programming language [5]. F2PY uses interface files (so called `pyf` files) to customize construction of the wrapped Fortran routines. For simple routines F2PY can automatically generate C extension libraries directly from Fortran sources. In practice the interface must first be generated using F2PY and then customized by hand. This process involves defining external or callback functions, declaring whether Fortran function arguments are meant as input or output and specifying their type. Once complete, the `pyf` file can be used by F2PY with the Fortran source code to compile the extension libraries.

This procedure was used to wrap the NLEQ2 library, as discussed earlier, after solving one minor problem. It turned out F2PY was incapable of handling a certain type of Fortran initialization used by NLEQ2. Luckily, F2PY is actively maintained by its author and in true Open Source style the necessary capability was added to F2PY ... 48 hours after being made aware of the problem.

[17] http://www.netlib.org/contin/
[18] http://www.psc.edu/~burkardt/src/pitcon/pitcon.html
[19] http://cens.ioc.ee/projects/f2py2e/

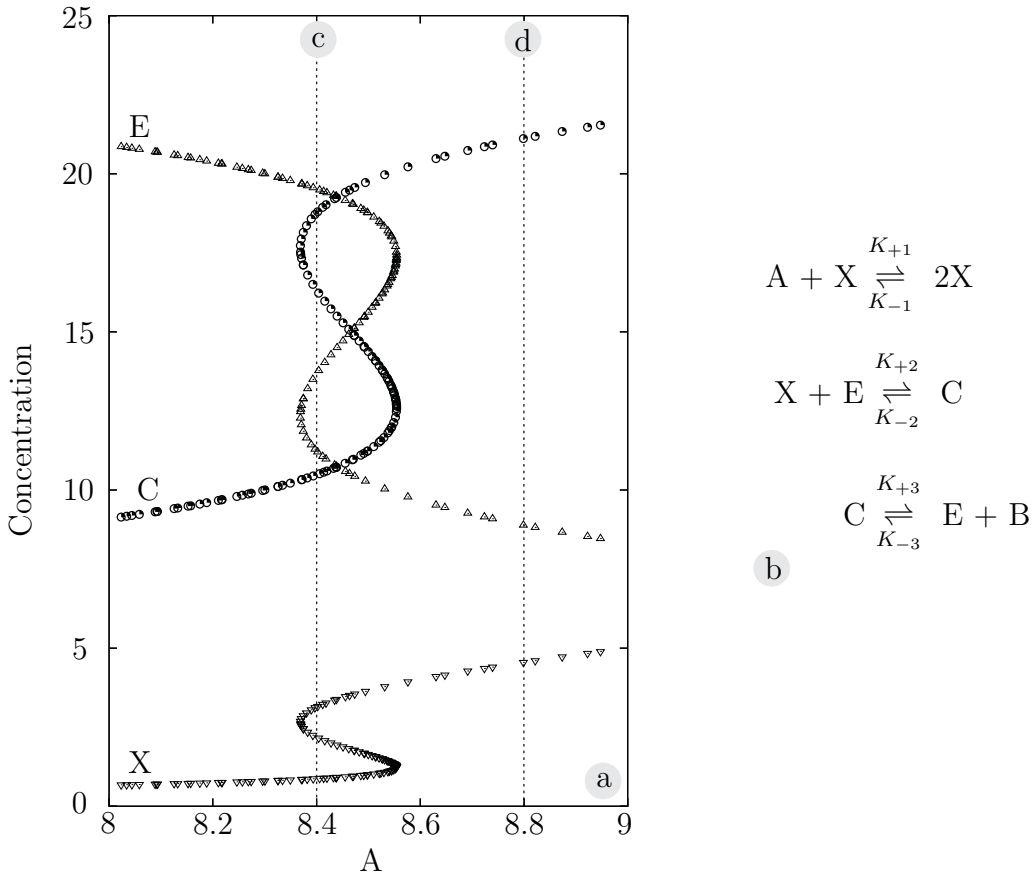

Fig. 6 *Parameter scan generated using PySCeS continuation.* The Edelstein model (whose reaction steps are shown in b) [41] exhibits multiple steady-state solutions (represented by the steady-state metabolite concentrations **X**, **E** and **C**) at different values of the parameter, **A**. In a this can be seen by looking at the intersection between the steady-state concentration curves with the values of **A** indicated by the lines c and d. In the case of c there are three intersections between the dashed line and steady-state concentration curves, i.e. three potential steady-state solutions, while at the value represented by line d only a single steady-state solution possible.

PITCON proved to be more of a challenge, as its Fortran function definition had a Fortran external function name as an argument, something which F2PY was incapable of dealing with. The argument `SLNAME` is used to specify which of two solvers (DENSLV or BANSLV) PITCON should use depending on whether the Jacobian is banded or not. One possible solution would have been to 'fix' the algorithm to only use one solver and remove the argument from the function signature. However, this would mean changing the Fortran source code and we considered this to be unacceptable as the elegance of using F2PY is precisely that the original source code can be used unaltered.

Fortunately, an alternate solution was suggested by the F2PY author[20] and this involved creating a Fortran subroutine (PITCON1) which would be used to call PITCON with the correct arguments. This subroutine (shown in Appendix **??**) has the same function signature as PITCON, except that, instead of the `SLNAME` function argument, an integer argument (`IMTH`) is introduced. The wrapper subroutine (PITCON1) has been written so that if `IMTH` has a value of one (its default value) PITCON is called with the standard solver, while if it is zero the banded solver is used. It was then possible to wrap PITCON1 (including PITCON itself) using F2PY.

12.2 Implementing the PITCON algorithm

PySCeS implements the PITCON continuation algorithm as a parameter scan, where a system parameter and range is supplied to the routine and a parameter plot with the supplied system parameter as the independent variable is returned. To see why the continuation was implemented in this way it is necessary to understand how the algorithm itself works, the following explanation is an extract from the PITCON source code.

> PITCON computes a sequence of solution points along a one dimensional manifold of a system of nonlinear equations F(X)=0 involving NVAR-1 equations and an NVAR dimensional unknown vector X.
>
> The operation of PITCON is somewhat analogous to that of an initial value ODE solver. In particular, the user must begin the computation by specifying an approximate initial solution, and subsequent points returned by PITCON lie on the curve which passes through this initial point and is implicitly defined by F(X)=0. – *extract from the PITCON source code file dpcon61.f*[21]

From this description three factors regarding the generic implementation of this algorithm become clear: the algorithm needs a initial guess, needs to run in an iterative loop and reduces the number of differential equations by one. The first two factors do not present a problem as we can provide the continuation algorithm an exact solution by

[20] http://cens.ioc.ee/pipermail/f2py-users/2003-December/000611.html
[21] Available online at http://www.netlib.org/contin/dpcon61.f

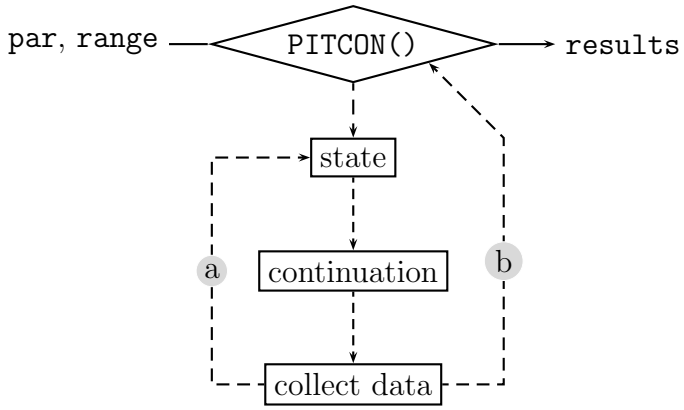

Fig. 7 *Continuation algorithm using PITCON.* For every point in the supplied `range` a steady state is calculated and used to initiate a continuation for the parameter (`par`), a. Once completed the complete data set is returned to the calling method, b and output as the continuation `results`.

calculating a steady state and then using it as an initial value, then repeat the continuation routine an arbitrary number of times. The last factor was problematic as PySCeS does not solve the differential equations directly, which meant that if the independent concentration vector, $\mathbf{s_I}$ (whose length corresponds to NVAR) was used with PITCON, it turned the final value of this vector into a parameter. As a parameter it was excluded from the calculated solution, which in turn made it impossible to properly evaluate the rate equations. After various attempts, no way was found to use $\mathbf{s_I}$ directly.

However, the solution to this problem was not far away and is one of the main reasons why the PITCON algorithm was implemented as a parameter scan. First of all, it was necessary to allow all the elements of the concentration vector to be treated as free variables. This meant extending $\mathbf{s_I}$ by one (equivalent in length to NVAR+1). As continuations are generally used as part of a parameter scan, the concentration vector was extended by appending a parameter (λ) thus forming an extended concentration vector, $\mathbf{s}_{\mathbf{I},\lambda}$

$$\mathbf{s}_{\mathbf{I},\lambda} = [\mathbf{s_I}\ \lambda]^T$$

Using $\mathbf{s}_{\mathbf{I},\lambda}$ with the ODE evaluation function meant that PITCON could use this extra element as a parameter (which it in fact was). PySCeS could, in turn, trim $\mathbf{s}_{\mathbf{I},\lambda}$ to $\mathbf{s_I}$ and dynamically assign λ a value before evaluating the rate equations using $\mathbf{s}$ which could be generated from $\mathbf{s_I}$. However, creating such a generic interface to the PITCON algorithm had a few meaningful consequences. First, the value of λ had to be stored with each solution generated by PITCON. Additionally, the data points generated by PITCON over the parameter range are not sequential, as for each initial point the algorithm generated a number of solutions which were spread over a range of λ values. This can roughly be thought of as having a mini-parameter scan generated by repeatedly calling the algorithm (the `continuation` block of Fig. 7) inside a parameter scan defined by

the user (shown in Fig. 7 a).

The only real effect this has is that the data generated by PITCON can only be plotted using data points and not by using lines (for an example see Fig. 6). On the other hand, this method provides an effective 'shotgun' approach that can be used to search for static bifurcations over an extended parameter range. As shown in Fig. 7 b, once the main parameter scan is completed the data is returned as an array of parameter values, concentrations and fluxes. PITCON can also act as a standard non-linear solver and it is possible (by manipulating the control parameters) to generate a complete parameter portrait using only the `PITCON()` method. Fig. 6 was generated in this way.

12.3 Using PITCON in PySCeS

This section describes how PITCON can be called and configured. To begin with, the following options control various aspects of the the PySCeS continuation method.

- `mod.PITCON(scanpar,scanpar3d=None)`: The main interface to the PITCON library. It takes `scanpar`, a string representing a model parameter and `scanpar3d`, a floating point argument that can be used to generating three dimensional parameter plots. This routine calls the native PITCON interface which is accessible via `pysces.pitcon.pitcon1()`
- `mod.pitcon_par_space` (default = scipy.logspace(-1,3,10)): Defines the user defined parameter scan range.
- `mod.pitcon_iter` (default = 10): Sets the number of iterations to go through for every point in user defined parameter range.
- `mod.pitcon_flux_gen` (default = 1): Return the steady-state fluxes and concentrations (1) or only concentrations (0).
- `mod.pitcon_allow_badstate` (default = 0): The continuation loop can be initialized with non steady-state values (1) or only valid steady states (1).
- `mod.pitcon_fix_small` (default = 0): In the rate equation evaluation function, return values smaller than 10^{-15} are set to a value of 10^{-15} (1) or allowed any value (0).
- `mod.pitcon_filter_neg` (default = 1): Do not postprocess physically invalid solutions containing negative concentrations (1) or process all solutions (0).
- `mod.pitcon_filter_neg_res` (default = 0): Remove all solutions containing negative concentrations (1) or return all results (0).
- `mod.pitcon_target_points` (default = `[ ]`): When `mod.PITCON` is used to calculated target points, they are stored in this list.

- `mod.pitcon_limit_points` (default = `[ ]`): If `mod.PITCON` is used to calculated limit points, they are stored in this list.

The following are all integer parameters that are passed directly to the PITCON algorithm[22]. It is important to note that any indexes are Fortran indexes where the first element has an index of one (i.e. Python index plus one).

- `mod.pitcon_init_par` (default = 1): An index indicating which component of the current continuation point which is to be used as the continuation parameter. PITCON sets this automatically unless overridden.

- `mod.pitcon_par_opt` (default = 0): Allow the algorithm to choose its local parameter (highly recommended) from step to step (0) or force it to use the parameter defined in `mod.pitcon_init_par` (1).

- `mod.pitcon_jac_upd` (default = 0): This option controls the frequency with which the PITCON algorithm should attempt to update the Jacobian during the Newton iterations. Evaluate the Jacobian at every Newton step (0), evaluate the Jacobian when the algorithm is started and then once every `mod.pitcon_max_steps` Newton steps (1) evaluate the Jacobian on the first step and then only when the algorithm fails (2). The default value, although computationally the more expensive option, seems suitable for highly non-linear systems.

- `mod.pitcon_targ_val_idx` (default = 0): This option causes PITCON to operate in a target seeking mode. A 'target point' is defined as a solution where a specific component, whose (FORTRAN style) index is defined by this argument, of $\mathbf{s}_{\mathrm{I},\lambda}$ has a value defined by the `mod.pitcon_targ_val` option. Results generated in this mode are not processed normally, but instead stored in the `mod.pitcon_target_points` list.

- `mod.pitcon_limit_point_idx` (default = 0): Seek limit points in the component of $\mathbf{s}_{\mathrm{I},\lambda}$ which has a (FORTRAN style) index represented by this parameter, a value of zero disables this search. Results are not processed normally but instead stored in the `mod.pitcon_limit_points` list.

- `mod.pitcon_output_lvl` (default = 0): Control the amount of intermediary output generated by the algorithm. There are four output levels (0,1,2,3).

- `mod.pitcon_jac_opt` (default = 1): Jacobian choice option. A user supplied Jacobian (0) is not currently supported. Valid options are either using forward difference approximation (1) or central difference approximation (2) to evaluate the Jacobian.

[22] Parameter names and descriptions are based on those found in the PITCON source code (dpcon61.f) available from http://www.netlib.org/contin/

- `mod.pitcon_max_steps` (default = 10*(len($\mathbf{s_I}$)+1)): Maximum number of newton steps allowed during a single run of the newton process.

Used in conjunction with the integer controls the following floating point parameters can also be used to modify PITCON's behaviour.

- `mod.pitcon_abs_tol` (default = 10^{-5}): Absolute error tolerance.
- `mod.pitcon_rel_tol` (default = 10^{-5}): Relative error tolerance.
- `mod.pitcon_min_step` (default = 0.01): Minimum stepsize.
- `mod.pitcon_max_step` (default = 30.0): Maximum stepsize.
- `mod.pitcon_start_step` (default = 0.3): Initial stepsize.
- `mod.pitcon_start_dir` (default = 1.0): Starting direction, +1.0 or -1.0.
- `mod.pitcon_targ_val` (default = 0.0): If PITCON is asked to search for a target value using `mod.pitcon_targ_val_idx` solution(s) containing this value in the relevant index are searched for.
- `mod.pitcon_max_grow` (default = 3.0): Maximum growth factor for the predictor step size.

By judicious use of these parameters, PITCON can be used as a flexible tool to study biological systems which have multiple steady-state solutions. Once regions of multi-stability have been identified, PITCON can be switched into a search mode and an attempt can be made to find specific steady-state solutions and limit points. Chapter **??** contains worked out examples which further demonstrate how `mod.PITCON()` can be used to study the stability and control of such a system.

13 Metabolic Control Analysis

So far in this document we have seen how to calculate the structural, dynamic and steady-state solutions for a kinetic model describing a cellular system. These solutions have been in terms of basic structural relationships and properties such as metabolite concentrations and enzyme reaction rates. The methods and algorithms described in this section combine a variety of these properties and by using the framework of Metabolic Control Analysis allow us to investigate higher order systemic properties such as flux and concentration control. Metabolic Control Analysis (MCA) is a theoretical framework that can be used to quantitatively understand the control and regulation of a system when it is in steady state [46, 47]. In order to do this, MCA defines two classes or types of relationships.

The first of these are the local properties of the individual steps in the system; the elasticity coefficients. An elasticity coefficient relates how a change in concentration of either a substrate, product or parameter affects the overall rate of an enzyme. When measuring the effect of this change, everything else in the system is assumed to stay constant, thus ensuring that the effect is local to the enzyme being studied. If these changes are assumed to be very small, elasticities can be represented as a partial derivatives where all other variable metabolites (**s**), external metabolites (**x**) and parameters (**p**) are constant quantities [12].

$$\varepsilon_{s_j}^{v_i} = \left(\frac{\partial v_i / v_i}{\partial s_j / s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} = \left(\frac{s_j}{v_i}\right)\left(\frac{\partial v_i}{\partial s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} = \left(\frac{\partial \ln v_i}{\partial \ln s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} \tag{11}$$

Eqn. 11 shows three equivalent versions of the scaled (dimensionless) elasticity ($\varepsilon_{s_j}^{v_i}$). The corresponding unscaled form of this elasticity ($\tilde{\varepsilon}_{s_j}^{v_i}$) would be:

$$\tilde{\varepsilon}_{s_j}^{v_i} = \left(\frac{\partial v_i}{\partial s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} \tag{12}$$

The definitions of the elasticity coefficient as shown in Eqn. 11 suggest that there are at least three possible ways to calculate the partial derivatives of the rate equations. By algebraic (symbolic) differentiation, automatic differentiation (an algorithmic approach used by Scientific Python) and numeric differentiation (perturbation). As described in Section 13.1, PySCeS implements the latter two methods. Once the elasticities with respect to the systems variable metabolites have been calculated ($\boldsymbol{\varepsilon}$) PySCeS has all the information necessary to investigate the second class of relationships in MCA: the response coefficients.

While the elasticity dealt with the sensitivity of a single reaction to changes in a metabolite concentration, a response coefficient is a measure of the effect of a change in a parameter or reaction rate on the system's steady state solution. It is therefore possible to define a response coefficient as the effect on a steady-state variable (**z**) of a change in a system parameter (**p**).

$$R_p^z = \frac{\partial z / z}{\partial p / p} = \frac{\partial z}{\partial p} \cdot \frac{p}{z} = \frac{\partial \ln z}{\partial \ln p} \tag{13}$$

Using the partitioned response property:

$$R_p^z = C_v^z \varepsilon_p^v$$

a steady-state flux (**J**) control coefficient,

$$C_v^J = \frac{R_p^J}{\varepsilon_p^v} = \frac{\delta \ln J}{\delta \ln v}$$

and steady-state concentration ($\mathbf{s}$) control coefficient,

$$C_v^s = \frac{R_p^s}{\varepsilon_p^v} = \frac{\delta \ln s}{\delta \ln v}$$

can be defined.

Whereas response coefficients are not calculated directly by PySCeS, the control coefficients are determined by implementing a method that uses the *control-matrix equation*[23] [12, 16, 17].

$$\begin{bmatrix} \tilde{\mathbf{C}}^J \\ \tilde{\mathbf{C}}^s \end{bmatrix} [\mathbf{K} \;\; -\tilde{\boldsymbol{\varepsilon}}_\mathbf{s}\mathbf{L}] = \begin{bmatrix} \mathbf{K} & 0 \\ 0 & \mathbf{L} \end{bmatrix} \tag{14}$$

which can be simplified to:

$$\tilde{\mathbf{C}}^i\tilde{\mathbf{E}} = \mathbf{I}$$

and as both $\tilde{\mathbf{C}}^i$ and $\tilde{\mathbf{E}}$ are square, invertible matrices [16, 51]:

$$\tilde{\mathbf{C}}^i = \tilde{\mathbf{E}}^{-1} \tag{15}$$

In Section 13.2 it will be seen how PySCeS uses the relationship shown in Eqn. 15 to calculate the matrix of independent control coefficient by inverting the elasticity matrix.

13.1 Calculating the elasticities

PySCeS includes two methods for calculating the partial derivatives (i.e. the unscaled elasticities) with respect to the variable metabolites and parameters, namely automatic and numerical differentiation. The default method (as shown in Fig. 8 a) is to use the automatic differentiation functions provided by Konrad Hinsen's Scientific Python[24]. This package contains a `FirstDerivative` function which works by defining a derivative variable (`DeriVar`) and then, using operator overloading, automatically expands and evaluates the partial derivatives of all other variables in the expression with respect to the `DeriVar`. PySCeS includes methods that utilize automatic derivation to obtain the unscaled elasticities towards both the variables, $\tilde{\boldsymbol{\varepsilon}}_\mathbf{v}$, as well as the system parameters, $\tilde{\boldsymbol{\varepsilon}}_\mathbf{p}$. The actual code used to calculate the derivatives is generated and compiled during the model loading process and then evaluated when required, making automatic differentiation an efficient way of obtaining the unscaled elasticities. However, there are situations where using automatic differentiation without algebraic substitution can lead to an incomplete set of derivatives. One known situation where this occurs is when forcing functions are used to represent equilibrium blocks in a system. As PySCeS has the philosophy of not altering the original rate equations in any way, elasticities that appear in forced function blocks will have a value of zero when the rate equations are differentiated automatically. In order to cater for this situation PySCeS includes methods for the

[23] Various alternatives to this equation have been proposed, see for example [11, 48–50])

[24] http://starship.python.net/~hinsen/ScientificPython/

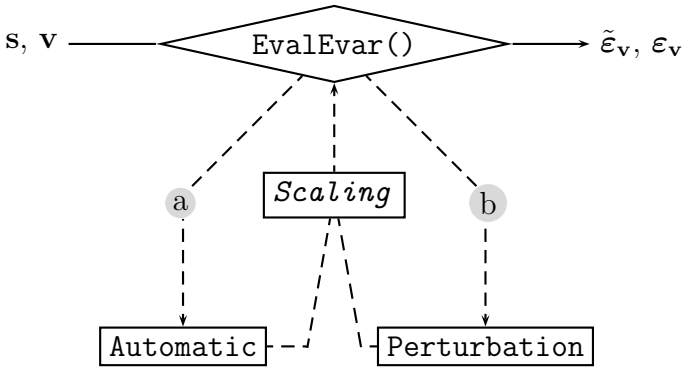

Fig. 8 *Differentiating the rate equations* Given a concentration (**s**) and rate vector (**v**) `EvalEvar()` generates the elasticities with respect to the variable metabolites ($\tilde{\varepsilon}_{\mathbf{v}}$) by either automatic, a, or numeric differentiation, b. Once generated $\tilde{\varepsilon}_{\mathbf{v}}$ can optionally be scaled to $\varepsilon_{\mathbf{v}}$.

numerical approximation of the elasticities (as shown in Fig. 8 b). In doing so PySCeS circumvents this particular problem as any forcing function is always co-evaluated with the rate equations and any effect on the system is therefore taken into account.

Numerical differentiation is performed by way of a stepwise perturbation of the rate equations using the `scipy.derivative`[25] function. When supplied a function and a reference point, `scipy.derivative` uses an N point central difference formula with a fixed step size to numerically approximate the function's partial derivatives. After running a series of comparisons between the automatic and numerical differentiation methods, it was found that a three point derivative with a scaled step size produced the most accurate results. However, determining the ideal step size proved to be problematic. If it was too large it could give inaccurate results when the reference point, around which the derivative was being determined, was small. On the other hand, if it was set too small it could introduce significant error into the result. The problem of finding a universal step size could however be eliminated[26] by using a scaled step size which was obtained by multiplying the reference point with a fixed 'derivation factor'. By doing so the step size was always set relative to the absolute magnitude of the reference point thereby ensuring reasonable accuracy for both small and large initial values. Although slower than the previously described method, numerical differentiation provides a reliable method for calculating the elasticities.

So far we have seen how PySCeS calculates the matrices of unscaled elasticities towards both the variables and parameters. However, as it is more common to use the scaled forms of these elasticities PySCeS, by default, scales the elasticity matrices and attaches the individual elasticities as model attributes. Alternatively, by setting

[25] http://www.scipy.org
[26] This solution was inspired by Herbert Sauro's Jarnac [8]

`mod.mode_mca_scaled`, unscaled elasticities can instead be attached. The following properties and methods are related to the calculation and display of model elasticities:

- `mod.doElas()` is a a high-level method which, when called, calculates a steady state as well as the elasticities towards both the variable metabolites and parameters.
- `mod.EvalEvar(input=None,input2=None)` calculates the elasticities towards the variable metabolites. If called with no arguments the current steady-state concentrations and fluxes are used as input. User supplied values can be used where `input1` is a valid concentration and `input2` a valid rate vector. No checks on the validity of user supplied input vectors are performed.
- `mod.EvalEpar(input=None,input2=None)` calculates the elasticities towards the system parameters. See `mod.EvalEvar()` for a description of this method's arguments.
- `mod.ecRate_Metabolite` attributes represent scaled variable metabolite elasticities e.g. `mod.ecR4_s2`.
- `mod.ecRate_Parameter` attributes represent scaled parameter elasticities e.g. `mod.ecR4_k1`.
- `mod.uecRate_Metabolite` are the unscaled variable metabolite elasticities e.g. `mod.uecR4_s2`.
- `mod.uecRate_Parameter` are the unscaled parameter elasticities e.g. `mod.uecR4_k1`.
- `mod.mode_mca_scaled` (default = 1): scale all elasticities and control coefficients (1) or return only unscaled (0) values.
- `mod.mode_elas_deriv` (default = 0): calculate the elasticities using automatic differentiation (0) or perturbation (1).
- `mod.mode_elas_deriv_order` (default = 3): when using numerically determined derivatives this sets the number of points to use for the approximation.
- `mod.mode_elas_deriv_factor` (default = 10^{-4}): the factor used to determine the perturbation step size (dx) so that $dx = So \times factor$.
- `mod.mode_elas_deriv_min` (default = 10^{-12}): this is the minimum value that dx is allowed to have after scaling by the derivation factor.
- `mod.elas_evar_upsymb` (default = 1): allow `mod.EvalEvar()` to attach individual elasticity attributes (1) or not (0).

- `mod.elas_epar_upsymb` (default = 1): allow `mod.EvalEpar()` to attach individual elasticity attributes (1) or not (0).

The elasticities are also available as matrices and can be displayed using a `show()` method.

- `mod.elas_var`: scaled variable elasticity matrix, $\boldsymbol{\varepsilon}_{\mathbf{v}}$.
- `mod.elas_var_u`: unscaled variable elasticity matrix, $\tilde{\boldsymbol{\varepsilon}}_{\mathbf{v}}$.
- `mod.elas_par`: scaled parameter elasticity matrix, $\boldsymbol{\varepsilon}_{\mathbf{p}}$.
- `mod.elas_par_u`: unscaled parameter elasticity matrix, $\tilde{\boldsymbol{\varepsilon}}_{\mathbf{p}}$.
- `mod.elas_var_row`: variable elasticity matrix row labels (reaction names).
- `mod.elas_var_col`: variable elasticity matrix columns labels (metabolites names).
- `mod.elas_par_row`: parameter elasticity matrix row labels (reaction names).
- `mod.elas_par_col`: parameter elasticity matrix columns labels (parameter names).
- `mod.showEvar(File=None)`: print the variable metabolite elasticities to the screen or file (`F`).
- `mod.showEpar(File=None)`: print the parameter metabolite elasticities to the screen or file (`F`).
- `mod.showElas(File=None)`: print all elasticities to the screen or file (`F`).

When the `show()` methods print the elasticities they format them in the following, easy to read, syntax:

```
'R1'
\ec{R1}{s0} = -3.0043e-001

\ec{R1}{k1} = 1.3004e+000
```

This syntax has the additional advantage that it doubles as a LaTeX macro and when used in conjunction with the macro definition file[27] included with PySCeS, elasticities can be 'copied and pasted' directly into LaTeX documents.

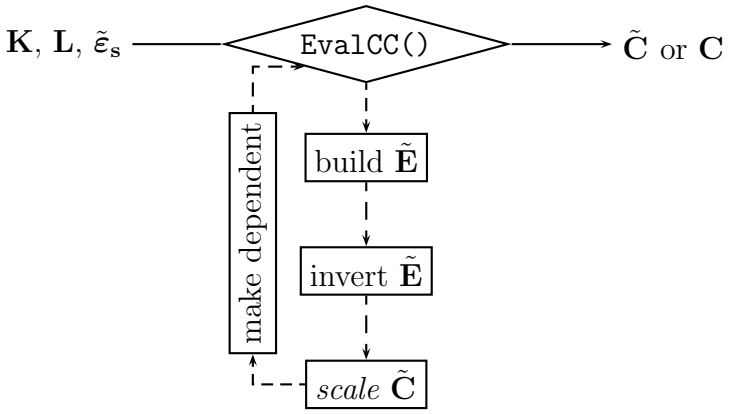

Fig. 9 *Calculating the control coefficients.* Using the **K**, **L** and elasticity matrix $\tilde{\varepsilon}$, `EvalCC()` constructs $\tilde{\mathbf{E}}$ using $[\mathbf{K} \ -\tilde{\varepsilon}_s\mathbf{L}]$. The control-matrix equation, $\tilde{\mathbf{E}}$ is inverted to form $\tilde{\mathbf{C}}^i$, a matrix of independent control coefficients. This matrix is then optionally scaled and used to regenerate the dependent coefficients.

13.2 Calculating the control coefficients

Once a system's elasticities have been obtained the control coefficients can be calculated by using the control-matrix equation. Fig. 9 gives an overview of the method which is used to implement Eqn. 15 and calculate a system's control coefficients. The initial steps (constructing **E** and inversion) carried out by the evaluation method, `mod.EvalCC()` are a direct implementation of the control-matrix equation and show once again the interaction and blending of a system's structural information, in the form of the **K** and **L** matrices, with its kinetic properties ε to form (using Eqn. 14) the **E** matrix. Although it might seem to be computationally inefficient to use such an all or nothing approach to calculate the control coefficients, it does have some advantages. The first is that the inversion is performed using the LAPACK functions DGETRF and DGETRI both of which are ALTAS linked, external libraries that are optimized for performance. As no perturbations are performed, there is a minimal introduction of numerical error and control coefficients can potentially be calculated for unstable steady states. Of course, although numerically correct (and unlike an elasticity), whether a control coefficient is actually defined, in a biological sense, for an unstable steady state is a different question entirely.

It is perhaps interesting to note at this point that all scaling is left until after matrix inversion. Although perhaps not algorithmically the most obvious route to follow, it would be more efficient to use the scaled form of the elasticity matrix (which already exists) and simply scale **K** and **L**, to form a scaled **E**. By inverting a scaled **E** the resulting matrix would contain scaled control coefficients. Unfortunately, in practice this is not a viable option; if the model has, for example, blocks of reactions that are in

[27] `pyscesmacros.tex` can be found in the docs/ subdirectory of the PySCeS distribution and is based on Jannie Hofmeyr's macro definition file `macros.tex`.

equilibrium, the fluxes through these blocks are zero, so that the elasticities of the steps in this block are infinite (either positive or negative)[28]. If scaling is done before inversion the infinite elasticities that appear in **E**, cause the inversion to fail. This scenario can be (and is) completely avoided if post-inversion scaling is used. A similar situation is also encountered when forcing functions are used to model equilibrium blocks and automatic differentiation is used to determine the elasticities. This also leads to inversion problems and is solved by using elasticities calculated by numeric differentiation.

Once the inversion has taken place (and if requested), PySCeS immediately scales the resultant $\tilde{\mathbf{C}}^i$ matrix to form the scaled matrix of independent control coefficients, $\mathbf{C^i}$. Programmatically this is almost as effective as scaling **E**, as the number of divisions that need to be made to scale the matrix is still only equivalent to the number of independent metabolites.

After scaling, the dependent control coefficient can be regenerated using the following relationships [12]:

$$\begin{aligned} \mathbf{C^{s_d}} &= \mathcal{L}_0 \mathbf{C^{s_i}} \\ \mathbf{C^{J_d}} &= \mathcal{K}_0 \mathbf{C^{J_i}} \end{aligned} \tag{16}$$

Once the dependent control coefficient have been generated, PySCeS stores all the control coefficients in a variety of matrices as well as attaching the individual control coefficients as model attributes.

- `mod.doMca()`: a high-level method that calculates a steady state, elasticities and control coefficients.
- `mod.EvalCC()`: the control analysis evaluation method, described in Fig. 9, which calculates the systems control coefficients.
- `mod.ccMetabolite_Rate`: a scaled concentration control coefficient attribute, e.g. `mod.ccs1_R4`, where `Metabolite` represents a steady-state metabolite concentration.
- `mod.ccJFlux_Rate`: a scaled flux control coefficient attribute, e.g. `mod.ccJR1_R4`.
- `mod.uccMetabolite_Rate`: an unscaled concentration control coefficient attribute, e.g. `mod.uccs1_R4`.
- `mod.uccJFlux_Rate`: an unscaled flux control coefficient attribute, e.g. `mod.uccJR1_R4`.
- `mod.mode_mca_scaled` (default = 1): scale all elasticities and control coefficients (1) or return only unscaled (0) values.

[28] Personal communication J.-H.S. Hofmeyr, J.M. Rohwer and J.L. Snoep

- `mod.mca_ccj_upsymb` (default = 1): attach the individual flux control coefficients as accessible model object properties (1), or skip doing so (0).
- `mod.mca_ccs_upsymb` (default = 1): attach the individual concentration control coefficients as accessible model object properties (1), or skip doing so (0).
- `mod.showCC(File=None)`: print the control coefficients to the screen, or open `File` object, according to the following settings.
- `mod.mca_ccall_fluxout` (default = 1): `mod.showCC()` prints the flux control coefficients (1) or not (0).
- `mod.mca_ccall_concout` (default = 1): `mod.showCC()` prints the concentration coefficients (1) or not (0).
- `mod.mca_ccall_altout` (default = 0): `mod.showCC()` prints the control coefficients grouped per steady-state metabolite or flux (0) or alternatively groups the coefficients by reaction (1).

As was the case with the elasticity `mod.show()` methods, `mod.showCC()` prints the control coefficients using an easy to read syntax which doubles as a LaTeX macro:

```
'JR6'
\cc{JR6}{R7} = 1.1147e-002
\cc{JR6}{R1} = 4.6856e-002

's9'
\cc{s9}{R7} = -4.9304e-003
\cc{s9}{R1} = 1.4914e-001
```

Assuming they exist, PySCeS stores the control coefficients in the following set of matrices, but unlike the structural matrices considered earlier in this document, the row and column labels for the MCA matrices all contain the actual label and not an index:

- `mod.mca_ci`: matrix of independent flux and concentration control coefficients.
- `mod.mca_ci_row`: independent flux and concentrations.
- `mod.mca_ci_col`: reaction steps ordered as independent and dependent rates.
- `mod.mca_cjd`: matrix of dependent flux control coefficients.
- `mod.mca_cjd_row`: dependent fluxes.
- `mod.mca_cjd_col`: reaction steps ordered as independent and dependent rates.

- `mod.mca_csd`: if moiety conservation exists this matrix holds the dependent concentration control coefficients.
- `mod.mca_csd_row`: dependent concentrations
- `mod.mca_csd_col`: reaction steps divided into independent and dependent rates.
- `mod.cc_all`: the full control coefficient matrix
- `mod.cc_all_row`: steady-state fluxes and concentrations arranged as dependent and independent fluxes, dependent and independent concentrations.
- `mod.cc_all_col`: reaction steps ordered as independent and dependent rates.
- `mod.cc_conc`: matrix containing all concentration control coefficients
- `mod.cc_conc_row`: independent and dependent steady-state concentrations.
- `mod.cc_conc_col`: reaction steps ordered as independent and dependent rates.
- `mod.cc_flux`: matrix containing all flux control coefficients.
- `mod.cc_flux_row`: dependent and independent steady-state fluxes.
- `mod.cc_flux_col`: reaction steps ordered as independent and dependent rates.

14 Stability analysis

PySCeS provides elementary support for studying the stability of a system by providing methods which can calculate the eigenvalues of the Jacobian matrix. As shown in [12] it is possible to formulate the Jacobian matrix ($\mathbf{M}$) for the kinetic model as:

$$\mathbf{M} = \mathbf{N_R}\tilde{\boldsymbol{\varepsilon}}_\mathbf{s}\mathbf{L} \tag{17}$$

SciPy provides a function `scipy.linalg.eig` which is an interface to the LAPACK routine GEEV which computes the eigenvalues of a square matrix. By applying this method to the Jacobian the eigenvalues and alternatively left and right eigenvectors can easily be calculated. The formal definitions of the left and right eigenvectors are provided in Appendix **??** which contains an extract from the GEEV source code[29].

- `mod.doEigen()`: a high level method that calculates a steady state, elasticities and eigenvalues.
- `mod.EvalEigen()`: the eigenvalue analysis method assumes the the elasticity matrix, $\tilde{\boldsymbol{\varepsilon}}_\mathbf{s}$, exists.

[29] http://www.netlib.org/

- `mod.eigen_values`: a vector attribute that holds the eigenvalues.
- `mod.lambda#`: an attribute containing an individual eigenvalue where `#` is a number.
- `mod.mode_eigen_output (default = 0)`: normally the `mod.EvalEigen()` only solves for the eigenvalues (0), if this switch is set to (1) both the left and right eigenvectors are also determined.
- `mod.eigen_vecleft`: the left eigenvector.
- `mod.eigen_vecright`: the right eigenvector.
- `mod.showEigen(File=None)`: Print the eigenvalues and statistics to screen or `File`.

By comparing the eigenvalue components, as determined by `mod.EvalEigen()`, the linear stability of the system's response to a perturbation around a particular steady state can be determined. `mod.showEigen()` tries to characterize a system's stability using the following definitions [52]:

- if all eigenvalues have exclusively negative real parts then the steady state is asymptotically *stable* and will relax back to a steady state after being perturbed.
- if any of the real parts of the eigenvalues are positive the steady state is *unstable* and will move away from the original steady state once perturbed.
- if any of the eigenvalues are zero then the steady state is *undetermined* and no stability information can be extracted.

PySCeS also calculates the stiffness of system as the ratio between the absolute values of the largest and smallest real part of the eigenvalues. Stiffness is a measure of the timescale separation between the fastest and slowest reactions in a system when relaxing back to a steady state. PySCeS summarizes the stability properties of the system as a report:

```
mod.showEigen()

Eigen values
[-7.65858678+0.j -4.09911499+0.j -1.24229823+0.j]

Eigen statistics
Max real part: -1.2423e+000
Min real part: -7.6586e+000
Max absolute imaginary part: 0.0000e+000
```

```
Min absolute imaginary part: 0.0000e+000
Stiffness: 6.1649e+000

## Stability --> Stable state
Purely real: 3
Purely imaginary: 0
Zero: 0
Positive real part: 0
Negative real part: 3
```

15 The Tao of PySCeS: part 3

"The purpose of computation is insight, not numbers." – Richard Hamming[30]

So far in this document we have been working our way out from the inside of a PySCeS model object. We have seen how the basic model attributes are generated, structural properties analysed, how we can analyse the time dependent and steady-state behaviour and finally investigate the metabolic control analysis and stability of a system. This is, of course, the core of any modelling package namely – its data generating capabilities. However, as mentioned by Richard Hamming all this data is meaningless if it can't be further interpreted and used to provide insight into our model.

16 Single dimension parameter scan

Earlier in this document we have seen how we can investigate the system's behaviour concentrations over time by way of a time simulation. Once in steady state, such transient changes are zero. A related question is how a change in parameter other than time affects the steady state of a system. An excellent example of such a plot is the rate characteristic which can be used to understand regulation in complex cellular systems [53]. A slightly more esoteric example of a parameter scan has already been seen in Fig. 6 which formed part of the discussion on parameter continuation.

PySCeS includes methods for the quick generation and plotting of single dimension parameter scans. Two high level methods are provided for this purpose; they entail defining the parameter scan in terms of the range over which PySCeS should scan as well as the input and output data:

- `mod.Scan1(range1=[])`: generates the scan data; `range1` is an array of points to scan over.
- `mod.Scan1Plot()` plots the results of a scan.

[30] Numerical Methods for Scientists and Engineers (1962)

- `mod.scan_in`: a string defining the parameter to be scanned e.g. `'x0'`
- `mod.scan_out`: a list of strings representing the attribute names one would like to track in the output eg. `['JR1','JR2','s1ss','s2ss']`
- `mod.scan_res`: once the parameter scan is completed this holds the results. The input parameter is stored in the first column followed by the requested output.
- `mod.scan1_dropbad` (default = 0): a switch that causes the Scan1 method to either drop (1) or maintain (0) invalid steady-state results (assuming all the solvers fail).
- `mod.scan1_mca_mode` (default = 0): a switch that either turns on (1) or switches off (0) a control analysis with every steady-state evaluation.

SciPy includes two useful functions that can be used to generate the scan range (`range1`):

- `scipy.linspace(start, end, points)`: generates a linear range.
- `scipy.logspace(start, end, points)`: generates a logarithmic range.

Both `scipy.linspace` and `scipy.logspace` use the number of points (including the start and end points) in the interval as an input. However, the start and end values of `scipy.logspace` must be entered as powers of base 10. For example to start the range at 0.1 and end it at 100 one would write `scipy.logspace(-1, 2, steps)`. Setting up a scan session might look something like:

```
mod.scanIn = 'x0'
mod.scanOut = ['JR1','JR6','s2ss','s7ss']
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, e.g. `mod.JR1` is created when `mod.doState()` method is executed. If `mod.Scan1()` detects that elasticities and or control coefficients are needed as output, it automatically calculates them. An example scan session might look something like:

```
mod.Scan1(scan_range)

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

done.
```

When the scan has been completed successfully, the results are stored in an array. If PySCeS does not recognize an input parameter or output value as a model property, it is skipped and will not be used in the scan. Once the parameter scan data has been generated, the data can be visualized using the `mod.Scan1Plot()` method:

```
mod.Scan1Plot(plot=[],filename='',title='title',logx='',\
              logy='',cmdout=0,fmt='w l')
```

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

- `plot=[]`, if an empty list (default) it is assumed to be `mod.scan_out` and everything is plotted. However, any combination of `mod.scan_out` elements can be entered as a list and plotted.
- If a filename is set `mod.Scan1Plot()` will, in addition to displaying the results on the screen, try to write a PNG image of the plot to a file: *filename.png*
- By default `Scan1Plot()` generates a title for the plot using the model file name plus the time the plot is generated. If required, custom graph titles can be set with `title='custom title'`
- If `logx` and `logy` are given a string value (`logx = 'on'`) the respective axis is displayed using a logarithmic scale.
- The `cmdout` switch (default = 0), if enabled (1), returns the GnuPlot plotting string in place of drawing the graph.
- `fmt` (default = 'w l') is the GnuPlot line format string used to plot the data.

Currently PySCeS only contains built in support for single dimension parameter scans but it is of course relatively simple to write customized, multi-dimensional scan functions using elementary Python, SciPy and PySCeS functions. For an example of a two dimensional scan see Appendix **??**.

17 PySCeS: data formatting functions

Throughout the previous sections various `showSomething()` methods that either print a formatted string representing a PySCeS model attribute to the screen or file. In the first part of this section these methods are collected together with some additional ones that have not yet been described. The second part describes methods used to output data as formatted arrays.

17.1 Displaying and printing model attributes

All of the `showSomething()` methods, with the exception of `mod.showModel()`, operate in exactly the same way. If called without an argument, they display the relevant attribute information to the screen. Alternatively if given an open, writable (ASCII mode) file object as an argument, these methods instead write their information to file. This makes it possible to generate customized reports by calling a selection of these methods using a single file object as an argument.

- `mod.showModel(filename=None)`: print the entire model as a formatted PySCeS input file to the screen or a file, *filename.psc*.
- `mod.showRateEq(File=None)`: print the reaction stoichiometry and rate equations.
- `mod.showODE(File=None)`: print the full set of differential equations.
- `mod.showODEr(File=None)`: print the reduced set of differential equations.
- `mod.showConserved(File=None)`: print any moiety conserved relationships (if present).
- `mod.showMet(File=None)`: print the current value of the model metabolites (mod.M).
- `mod.showMetI(File=None)`: print the initial, parsed in, value of the model metabolites (mod.Mi).
- `mod.showPar(File=None)`: print the current value of the model parameters.
- `mod.showRate(File=None)`: print the current set of reaction rates
- `mod.showState(File=None)`: print the current steady-state fluxes and metabolites.
- `mod.showN(File=None)`: print the stoichiometric matrix.
- `mod.showNr(File=None)`: print the reduced stoichiometric matrix.
- `mod.showL(File=None)`: print the L matrix
- `mod.showK(File=None)`: print the K matrix
- `mod.showElas(File=None)`: print all the elasticities
- `mod.showEvar(File=None)`: print the elasticities to the variable metabolites.
- `mod.showEpar(File=None)`: print the elasticities to the parameters.

- `mod.showCC(File=None)`: print the control coefficients.
- `mod.showModes(File=None)`: print the elementary modes.
- `mod.showEigen(File=None)`: print the eigenvalues.

The following example writes a report containing the model stoichiometry, steady-state results and elasticities to a file (`results.txt`) represented by an open file object (`rFile`).

```
rFile = open('results.txt','w') # open a writable, ASCII file object

mod.showState()      # print the steady-state results to screen
mod.showN(rFile)     # write the stoichiometric matrix to file
mod.showState(rFile) # write the steady-state results to file
mod.showEvar(rFile)  # write the elasticities to file

rFile.close() # close the file object
```

17.2 Writing formatted arrays

The `showSomething()` methods described in the previous section provide the user with a convenient way to write model attributes to either screen or file. PySCeS also includes a suite of generic array writers that enable the writing of any array to a file in a variety of formats. Unlike the `showSomething()` methods, the `Write_array` methods are designed to write to an open file object and do not print information to the screen.

When modelling cellular systems it is rare that any array needs to be stored or displayed without specific labels detailing its rows and columns. All the `Write_array` methods, therefore, take list arguments that can contain either the row or column labels. Obviously in order to work properly, 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 with either single dimension arrays or a matrices.

17.2.1 Writing formatted text

The most basic array writer is the `Write_array()` method. By default this method writes a tab delimited array to a file and is called as follows:

```
Write_array(input, File=None, Row=None, Col=None, close_file=0, separator='  ')
```

- `input`: the source array
- `File`: an open file object

- `Row`: a list of row labels
- `Col`: a list of column labels
- `close_file` (default = 0): close the file object (1) or leave it open (0) after writing the array.
- `separator`: by default a tab separator is used but any column separator can be specified here.

If column headings are supplied 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 names are written as a comment that follows the array data. The following switches can also be used to control this method's behaviour.

- `mod.write_array_header` (default = 1): write an id header before the array (1) or skip this step (0)
- `mod.write_array_spacer` (default = 1): write a empty line before the array (1) or not (0)
- `mod.mode_number_format` (default = `'%2.4e'`): this method uses the general PySCeS number format which can be set using this switch (please note this affects all number formatting).
- `mod.misc_write_arr_lflush` (default = 5): how many lines to write before flushing them to disk, this is a performance tweaking option.

The following is an example of the output from the `Write_array()` method:

```
## 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
```

17.2.2 Writing formatted 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. The method is called as follows:

```
Write_array_latex(input, File=None, Row=None, Col=None, close_file=0)
```

which generates

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

which when typeset 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

17.2.3 Writing formatted hypertext

`Write_array_html()` functions in a similar way to the LaTeX array writer but allows even more control over its output formatting. This method has been designed to quickly generate complete, web-ready, hypertext (HTML) pages for easy viewing and distribution of modelling results. Similarly to the methods previously described in this section, it makes provision for row and column headings.

As the method's output is destined primarily for display `Write_array_html()` includes certain visual enhancements to make viewing large matrices easier, as shown in Fig. 10. Assuming that row and column labels are requested, if any matrix dimension becomes larger than 15 the method automatically writes titles on two sides of the matrix. For example, if a matrix has more than 15 rows the column titles are placed on both the bottom and top of the matrix. In addition, if any dimension of the matrix exceeds 18 every 6th row or column is highlighted as needed. The method is called with:

```
Write_array_html(input, File=None, Row=None, Col=None, name=None, close_file=0)
```

As noted earlier this method generates complete HTML pages that can be viewed directly using a web browser. If, however, multiple arrays need to be viewed in the same file the HTML headers and footers can be selectively enabled or disabled.

- `mod.write_array_html_header` turns the header on or off – default (1) is on

Control coefficients

	Ru5Pk	TPT_PGA	TPT_GAP	TPT_DHAP	LReac	Rubisco	PGK
Ru5Pk	0.001	-0.000	-0.000	-0.000	0.001	0.001	0.000
TPT_PGA	0.171	0.786	-0.011	-0.237	-0.504	0.031	-0.000
TPT_GAP	-0.174	0.172	1.008	0.166	0.567	-0.031	0.000
TPT_DHAP	-0.174	0.172	0.008	1.166	0.567	-0.031	0.000
LReac	-0.005	-0.050	-0.000	-0.007	0.011	0.000	0.000
Rubisco	0.001	-0.000	-0.000	-0.000	0.001	0.001	0.000
PGK	-0.011	-0.059	0.001	0.017	0.039	-0.001	0.000
G3Pdh	-0.011	-0.059	0.001	0.017	0.039	-0.001	0.000
TPI	-0.032	-0.044	-0.004	0.156	0.106	-0.005	0.000
Aldl	0.023	-0.225	-0.015	-0.320	-0.078	0.006	-0.000
FBPase	0.023	-0.225	-0.015	-0.320	-0.078	0.006	-0.000
PGI	-0.733	7.620	0.504	10.808	2.658	-0.149	0.000
PGM	-0.733	7.620	0.504	10.808	2.658	-0.149	0.000

Fig. 10 *PySCeS HTML output.* Using the `Write_array_html()` method, arrays can be saved as complete, web-ready HTML pages. This example is an extract from a model described in [54].

- `mod.write_array_html_footer` turns the footer on or off – default (1) is on
- `mod.write_array_html_format` sets the number format of the output – default is '%2.4f'

An attempt has also been made to keep the HTML source code in a 'human readable format'. This allows easy editing of the generated HTML code and simplifies merging array data with other documents.

18 Miscellaneous model methods

The final set of PySCeS model methods include a number of utility methods which are used for data conversion, algorithm testing etc.

- `mod.Fix_S_fullinput(s_vec)`: generate a full length concentration vector (containing independent and corrected dependent concentrations) using a full length concentration vector (containing only correct independent concentrations) as input.
- `mod.Fix_S_indinput(s_vec)`: generate a full length concentration vector from a vector of independent concentrations.
- `mod.FluxGen(s)`: calculate the reaction rates from a concentration vector. This method works on both vectors and arrays.
- `mod.SetQuiet()`: disable all solver and integrator output messages.
- `mod.SetLoud()`: enable all solver and integrator output messages.
- `mod.DelAttrType(attr=None)`: delete a specific class of model attributes. `attr` can have a value of either 'ec', 'uec', 'cc' or 'ucc'.
- `mod.TestSimState(endTime=10000, points=101, diff=1.0e-5)`: test the results of the steady-state solver routines against a 'long' time simulation. It takes the arguments `endTime`, the end time of the simulation, `points`, number of points and `diff` the maximum difference allowed between the two sets of results.
- `mod.TestElasCalc(testSlice=1.0e-5)`: compares the elasticity matrices generated using either numeric or automatic differentiation. `testSlice` is the maximum allowed tolerance allowed before an error is recorded.

Both of the testing methods have a relatively low tolerance for error, and it is sometimes more useful to look at the relative error shown as a percentage by these two methods. This is especially true when an elasticity might have a large absolute value.

19 PySCeS module functions

So far this document has dealt with the the properties and functions that form the core of PySCeS, namely the model object. In the rest of this section we will deal with PySCeS module functions which include amongst others plotting functions, testing frameworks and HTML formatting functions. All of these generic functions are not bound to the model object itself but are available via the PySCeS module interface, `pysces.*`.

19.1 Plotting and formatting graphs using `pysces.plt.*`

Currently, all plotting in PySCeS uses GnuPlot[31] via the `scipy.gplt` interface. The practical implications of this are that any calls to `scipy.gplt` functions act on the open, active graph. Although it is possible to explicitly import `scipy.gplt` and call these methods directly they sometimes have an obscure syntax and generally have very little documentation.

`pysces.plt` extends `scipy.gplt` in two important ways, it adds new methods for plotting and saving graphs and it wraps some of the basic formatting methods, simplifying their syntax and making them more convenient to use. In general the method names follow the equivalent GnuPlot syntax although sometimes they have been shortened to speed up interactive, command line usage. All PySCeS plotting functions are available via the `pysces.plt.*` interface which is an instance of the `PyscesGPlot` class which is contained in the file `PyscesPlot.py`.

19.1.1 Drawing graphs

There are three plotting methods for generating two and three dimensional plots. All three drawing methods methods accept the following arguments:

- `ginput`: a required matrix of input data
- `fmt`: a string containing the GnuPlot style data format. The default is to plot with lines ('w l').
- `cmdout` (default = 0): if active (1) PySCeS does not plot the graph, instead it returns the SciPy command that the method would have run, otherwise the data is plotted (0).
- `name`: an argument used in conjunction with `cmdout`. If `name` is not set the array name returned in the `cmdout` plot command will be ginput. If the `name` argument is supplied it will be used as the array name in the command string instead.

The first of the PySCeS plotting methods is the `pysces.plt.plot2D()` method which can be used to draw two dimensional plots.

[31] http://www.gnuplot.info/

```
pysces.plt.plot2D(ginput, x, ylist, cmdout=0, name=None, fmt='w l', ykey=None)
```

In addition to the argument described above `pysces.plt.plot2D()` also accepts the following arguments:

- `x`: the (Python style) numeric index of the x-axis data range.
- `ylist`: a list containing the indexes of the y-axis data ranges.
- `ykey`: (optional) a list equal in length and in the same order as `ylist`, that can contain data labels that should be used in the graph key. If this argument is not present the column index is used as the key.

Similarly, three-dimensional surface plots can be generated using the `plot3D()` method.

```
pysces.plt.plot3D(ginput, x, y, z, cmdout=0, name=None, fmt='w l',\
                  zkey=None, arrayout=0)
```

A restriction inherent in the `scipy.gplt` plotting interface is that only a single surface can be plotted at a time. Aside from the generic arguments to this function `plot3D()` takes the following arguments.

- `x, y, z`: the `ginput` column indices of the data to plot
- `zkey`: (optional) a string containing the key name
- `arrayout` (default = 0, optional): works with `cmdout`. The `scipy.plot3d` only plots an array with three columns, so the returned scipy.gplt command will refer to an array with these dimensions (as the method builds the array from the ranges supplied as arguments). If enabled (1), `arrayout` causes a tuple to be returned containing the plot command and the array.

Finally, the fastest plotting method of all is `pysces.plt.plotX()`. This method, when given a matrix, uses the first column for the x-axis data range and the rest of the columns as y-axis data. This method is meant as a 'quick' way of visualizing data and only uses the default plotting arguments.

```
pysces.plt.plotX(ginput, cmdout=0, name=None, fmt='w l')
```

19.1.2 Saving graphs

`pysces.plt` provides two methods for saving plots as images using the PNG format. Any active GnuPlot plot can be saved using this method:

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

- `filename`: a string representing the file name so that the final image name is *filename.png*
- `path`: an optional argument specifying the path (e.g. `'c:\\temp'`). If not supplied the default work directory is used.

The second method that can be used to save a graph is `pysces.plt.save_html()`. This method saves a complete HTML page, including the graph, allowing it to be displayed using a web browser. The method call and arguments are:

```
pysces.plt.save_html(imagename, File=None, path=None, name=None, close_file=1)
```

- `imagename`: this argument is the name used for the image and HTML file i.e. the method produces `imagename.png` and `imagename.html`
- `File`: when specified, the HTML output of the method is written to `File` while `Imagename` is still used to generate the image file name.
- `path` (optional): if supplied allows the output to be written to a specific directory.
- `name`: this is a graph title which is added to the the HTML page below the image itself.
- `close_file` (default = 1): by default this method closes the HTML file after writing to it (1). Disabling this option (0) leaves the HTML text file object open.

The overall behavior of the save methods can be controlled by setting the following module parameters.

- `pysces.plt.save_html_header` (default = 1): when saving graphs as HTML either enable (1) or disable (0) the writing of the HTML page header.
- `pysces.plt.save_html_footer` (default = 1): when saving graphs as HTML either enable (1) or disable (0) the writing of the HTML page footer.
- `pysces.plt.mode_gnuplot4` (default = 0): Between versions 3.7 and 3.8–4.0 GnuPlot changed the way it saved images to disk and unfortunately the new and old calls are incompatible. Enabling this option (1) allows PySCeS to work with GnuPlot versions 3.8 or newer.

As mentioned earlier the ability to switch the page headers and footers on and off allows multiple plots to be saved as a single HTML page.

19.1.3 Formatting graphs

The following methods are GnuPlot primitives that have been wrapped to simplify their interactive use from the command line. The first set affects the overall appearance of the graph.

- `gridon()`: enable grid display
- `gridoff()`: disable grid display
- `ticslevel(x=0.0)`: set value where tics begin
- `title(l='')`: the graph title

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

- `logx()`: x-axis uses a logarithmic scale
- `logy()`: y-axis uses a logarithmic scale
- `linx()`: x-axis uses a linear scale
- `liny()`: y-axis uses a linear scale
- `linxlogy()`: x-axis linear, y-axis logarithmic scale
- `logxliny()`: x-axis logarithmic, y-axis linear scale
- `logxy()`: both x-axis and y-axis have a logarithmic scale
- `linxy()`: both x-axis and y-axis have a linear scale

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

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

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

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

19.2 Generating web ready reports using `pysces.html.*`

The PySCeS web interface `pysces.html.*`, an instance of the `PyscesHTML` class found in `PyscesWeb.py`, can be used in conjunction with HTML generating methods that have already been described to create basic, web-ready reports.

- `pysces.html.h1(str, File, align='l', txtout=0)`: write a large text heading.
- `pysces.html.h2(str, File, align='l', txtout=0)`: write a medium text heading.
- `pysces.html.h3(str, File, align='l, txtout=0')`: write a small text heading.
- `pysces.html.par(str, File, align='l', txtout=0)`: write paragraph text.

All the methods described in this section take a string and an open ASCII file object as an argument as well as an optional argument which specifies the HTML text alignment. This argument which can be `l`, `c` or `r` is equivalent to the HTML left, center and right alignment. By default the string is formatted, aligned and written to the `File` object. If, however, the `txtout` switch is enabled (1) the File object is not required and the formatted string is simply returned by the method.

Although these methods might seem trivial, care has been taken to program them so that they return 'well formatted, human readable' HTML code. For example, long code lines are wrapped at the first whitespace found after 75 characters. Creatively combined with a web server, these basic formatting methods in conjunction with `Write_array_html()` and `pysces.plt.save_html()` PySCeS can be used to dynamically generate web-ready, HTML pages and might form the basis of a more web-interactive version of PySCeS.

19.3 PySCeS unit test framework: `pysces.test`

It is important for any but the most trivial software projects to have a basic and reliable testing framework. This becomes essential in modular projects where different groups might be working on different modules. Although PySCeS is a relatively small project, the `PyscesTest` class (found in `PyscesTest.py`) has been implemented to provide a quick way of checking the software's algorithmic integrity.

The test framework has been implemented using the Python `unittest` module and has two testing levels. Level one tests are run using the three basic models provided with PySCeS. For each of the three models the test methods calculate a steady state, the elasticities and control coefficients. The results of these analysis are then compared to the results of the same analyses that have previously been generated with a different software package. If the results are the equivalent the test passes. Level 2 tests are targeted towards extension libraries and contains a completely self contained unit test for the PITCON algorithm.

- `pysces.CopyModels()`: a utility function that copies the default models, supplied with PySCeS, into the user work directory. If the models exist they are skipped and not overwritten.
- `pysces.test(lvl)`: run the PySCeS unit tests where `lvl` can be run basic tests (1), extended tests (2) or all tests (10).

The unit tests are implemented as a PySCeS module method, `pysces.test`, which can be 'run' by instantiating an anonymous class instance: `pysces.test()`. An example test session is shown in Appendix **??**.

19.4 The PySCeS utility functions

The last PySCeS module is a collection of utility functions, including tools to copy models, convert line termination characters and create custom timers.

- `pysces.ConvertFileD2U(Filelist)`: convert, in place, the line termination characters for a list of files (`FileList`) from DOS `<CR><LF>` to UNIX `<CR>`.
- `pysces.ConvertFileU2D(Filelist)`: convert, in place, the line termination characters for a list of files (`FileList`) from UNIX `<CR>` to DOS `<CR><LF>`.
- `pysces.CopyModels(dirIn, dirOut)`: copy PySCeS model files from `dirIn` to `dirOut`.

19.4.1 The `TimerBox` class

When modelling with PySCeS it is often the case that a timer or counter routine is needed to keep track of a loop or set of nested loops. Custom writing a counter routine into every such loop can be tedious and prone to errors. PySCeS provides the `TimerBox` class to avoid these problems.

```
TimeBox = pysces.TimerBox()
```

Once instantiated `TimeBox` acts as a container object that can hold multiple independent timers. Each timer is a Python generator function which is instantiated with a reference time. Every time the next method is called on the generator the reference time is subtracted from the current time giving the elapsed time. In this way it is possible to have a 'timer' that keeps track of elapsed time without using a separate thread and with very low memory and processor overhead. Two type of timers can be created in a `TimeBox`:

- `timeBox.normal_timer(name)`: where `name` is a string. This will create a new timer, `TimeBox.name`,

- `timeBox.step_timer(name2,maxsteps)`: will create a step counting timer with a `name`, *name2* and a maximum number of iterations (`maxsteps`). This will create a new timer, `TimeBox.name2` as well as a step count attribute, `TimeBox.name2_cstep`.

Once normal timers are initialized the elapsed time can be output using the `next` method:

```
>>> TimeBox.name.next()
'23 min 40 sec'
```

Before a call to a step timer is made, the current step needs to be incremented:

```
>>> TimeBox.Step_timer_name2_cstep += 1
>>> TimeBox.name2.next()
'step 1 of 20 (24 min 50 sec)'
```

Multiple timers can co-exist in a `TimerBox` and can easily be manipulated using the following methods:

- `TimeBox.reset(name2)`: a step counter specific method which resets the step count attribute associated with `name2` to zero.
- `TimeBox.stop(name)`: will delete the timer `TimeBox.name`. In the case of a step timer the step attribute is deleted as well.

It is perhaps fitting that we end this discussion on PySCeS with perhaps the most frivolous (yet highly addictive) function, which happens to be based on a TimerBox method, `pysces.session_time()`:

```
>>> pysces.session_time()
'It is now Wed 14:23, this PySCeS session has been active for 234 min 53 sec'
```

20 The Tao of PySCeS: coda

> "any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected." – Eric Raymond [55]

> "The next best thing to having good ideas is to recognize good ideas from your users. Sometimes the latter is better." – Eric Raymond [55]

References

[1] Olivier, B. G., Rohwer, J. M., and Hofmeyr, J.-H. S. (2005). Modelling Cellular Systems with PySCeS. *Bioinformatics 21*, 560–561. 3

[2] Olivier, B. G. (2005). *Simulation and Database Software for Computational Systems Biology:* PySCeS and JWS Online. PhD thesis, Stellenbosch University. 3

[3] Ward, G. (2004). *Distributing Python Modules (2.3.4)*. url: http://www.python.org/doc/2.3.4/. 3

[4] Pfeiffer, T., Sanchez-Valdenebro, I., Nuno, J. C., Montero, F., and Schuster, S. (1999). METATOOL: For Studying Metabolic Networks. *Bioinformatics 15*, 251–257. 5, 25

[5] Peterson, P. (2000–). *f2py: Fortran to Python Interface Generator*. url: http://cens.ioc.ee/projects/f2py2e/. 5, 35, 38

[6] Brown, C. and Barr, M. (2002). Introduction to Endianness. *Embedded Systems Programming* , 55–56. 5

[7] Sauro, H. M. (1993). SCAMP: a general-purpose simulator and metabolic control analysis program. *Comput. Appl. Biosci. 9*, 441–450. 6

[8] Sauro, H. M. (2000). JARNAC: A System for Interactive Metabolic Analysis. In Hofmeyr, J.-H. S., Rohwer, J. M., and Snoep, J. L., editors, *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*, pp. 221–228. Stellenbosch University Press, Stellenbosch, South Africa. 6, 47

[9] van Rossum, G. (2004). *Python Language Reference (2.3.4)*. url: http://www.python.org/doc/2.3.4/. 8

[10] Hofmeyr, J.-H. S., Kacser, H., and der Merwe, K. J. V. (1986). Metabolic Control Analysis of Moiety Conserved Cycles. *Eur. J. Biochem. 155*, 631–641. 9

[11] Reder, C. (1988). Metabolic Control Theory: A Structural Approach. *J. Theor. Biol. 135*, 175–201. 16, 18, 46

[12] Hofmeyr, J.-H. S. (2001). Metabolic control analysis in a nutshell. In Yi, T.-M., Hucka, M., Morohashi, M., and Kitano, H., editors, *Proceedings of the* 2^{nd} *International Conference on Systems Biology*, pp. 291–300. Omnipress, Madison,WI, USA. 16, 18, 21, 45, 46, 51, 53

[13] Strang, G. (1993). *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA. 18, 19, 21

[14] Reich, J. and Selkov, E. (1981). *Energy Metabolism of the Cell*. Academic Press, London. 18

[15] Sauro, H. (1994). Moiety-Conserved Cycles and Metabolic Control Analysis: Problems in Sequestration and Metabolic Channeling. *Biosystems 33*, 15–28. 18

[16] Hofmeyr, J.-H. S. and Cornish-Bowden, A. (1996). Co-response analysis: a new strategy for experimental metabolic control analysis. *J. Theor. Biol. 182*, 371–380. 18, 46

[17] Hofmeyr, J.-H. S., Cornish-Bowden, A., and Rohwer, J. M. (1993). Taking enzyme kinetics out of control; putting control into regulation. *Eur. J. Biochem. 212*, 833–837. 18, 46

[18] Cornish-Bowden, A. and Hofmeyr, J.-H. S. (2002). The Role of Stoichiometric Analysis in Studies of Metabolism: An Example. *J. Theor. Biol. 216*, 179–191. 18, 19

[19] Rheinboldt, W. (1986). *Numerical Analysis of Parametrized Nonlinear Equations*. John Wiley and Sons, New York. 18, 38

[20] Famili, I. and Palsson, B. O. (2003). Systemic Metabolic Reactions are Obtained by Singular Value Decomposition of Genome-Scale Stoichiometric Matrices. *J. Theor. Biol. 224*, 87–96. 19

[21] Sauro, H. M. and Ingalls, B. P. (2004). Computational Analysis in Biochemical Networks: Computational Issues for Software Writers. *Biophys. Chem. 109*, 1–15. 19, 20

[22] Hofmeyr, J.-H. S. (1986). Steady state modelling of metabolic pathways: A guide for the prospective simulator. *Comput. Appl. Biosci. 2*, 5–11. 19

[23] Higham, N. J. (1996). Recent Developments in Dense Numerical Linear Algebra. Technical Report No. 288, University of Manchester, Manchester, England. 19

[24] Stewart, G. W. (1995). The Triangular Matrices of Gaussian Elimination and Related Decompositions. Technical Report TR-95-91, University of Maryland. 20

[25] Chang, X.-W. and Paige, C. C. (1998). On the Sensitivity of the LU Factorization. *BIT 38*, 486–501. 20

[26] Schuster, S. and Hilgetag, C. (1994). On Elementary Flux Modes in Biochemical Reaction Systems at Steady State. *J. Biol. Syst. 2*, 165–182. 25

[27] Heinrich, R. and Schuster, S. (1998). The Modelling of Metabolic Systems. Structure, Control and Optimality. *Biosystems 47*, 61–77. 25

[28] Cornish-Bowden, A. and Cardenas, M. L. (2002). Metabolic Balance Sheets. *Nature 420*, 129–130. 25, 26

[29] Schuster, S., Hilgetag, C., Woods, J. H., and Fell, D. A. (1996). Elementary Modes of Functioning in Biochemical Networks. In Cuthbertson, R., Holcombe, M., and Paton, R., editors, *Computation in Cellular and Molecular Biological Systems*, pp. 151–165. World Scientific, Singapore. 25

[30] Klamt, S. and Stelling, J. (2003). Two Approaches for Metabolic Pathway Analysis? *Trends Biotechnol. 21*, 64–69. 25

[31] Papin, J. A., Price, N. D., Wiback, S. J., Fell, D. A., and Palsson, B. O. (2003). Metabolic Pathways in the Post-Genome Era. *Trends Biochem. Sci. 28*, 250–258. 25

[32] Famili, I. and Palsson, B. O. (2003). The Convex Basis of the Left Null Space of the Stoichiometric Matrix Leads to the Definition of Metabolically Meaningful Pools. *Biophys. J. 85*, 16–26. 25

[33] Hindmarsh, A. C. (1983). ODEPACK, a Systematized Collection of Ode Solvers. In Stepleman, R. S., editor, *Scientific Computing*, pp. 55–64. North-Holland, Amsterdam. 28

[34] Petzold, L. R. (1983). Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations. *Siam J. Sci. Stat. Comput. 4*, 136–148. 28

[35] More, J. J., Garbow, B. S., and Hillstrom, K. E. (1980). User Guide for MINPACK-1. Technical Report ANL-80-74, Argonne National Laboratory. 34

[36] Powell, M. J. D. (1970). A Hybrid Method for Nonlinear Equations. In Rabinowitz, P., editor, *Numerical Methods for Nonlinear Algebraic Equations*, pp. 87–114. Gordon and Breach, London. 34

[37] Garbow, B. S., Hillstrom, K. E., and More, J. J. (1980). Implementation Guide for MINPACK-1. Technical Report ANL-80-68, Argonne National Laboratory. 34

[38] Deuflhard, P. (1974). A Modified Newton Method for the Solution of Ill-Conditioned Systems of Nonlinear Equations with Application to Multiple Shooting. *Numer. Math. 22*, 289–315. 35

[39] Deuflhard, P. (2004). *Newton Methods for Nonlinear Problems*. Springer Series in Computational Mathematics. Springer-Verlag, New York. 35

[40] Nowak, U. and Weimann, L. (1990). A Family of Newton Codes for Systems of Highly Nonlinear Equations – Algorithm, Implementation, Application. Technical Report TR 90–10, Konrad-Zuse-Zentrum fuer Informationstechnik Berlin (ZIB). 35

[41] Edelstein, B. B. (1970). Biochemical Model with Multiple Steady States and Hysteresis. *J. Theor. Biol. 29*, 57–62. 39

[42] Allgower, E. L. and Georg, K. (1990). *Numerical Continuation Techniques: An Introduction*. Springer Series in Computational Mathematics. Springer-Verlag, New York, USA. 38

[43] Rheinboldt, W. C. and Burkardt, J. V. (1983). Algorithm 596: A Program for a Locally Parametrized Continuation Process. *ACM Trans. Math. Software 9*, 236–241. 38

[44] Rheinboldt, W. (1980). Solution Field of Nonlinear Equations and Continuation Methods. *SIAM J. Numer. Anal. 17*, 221–237. 38

[45] Rheinboldt, W. and Burkardt, J. (1983). A Locally Parameterized Continuation Process. *ACM Trans. Math. Software 9*, 215–235. 38

[46] Kacser, H., Burns, J. A., and Fell, D. A. (1995). The control of flux: 21 years on. *Biochem. Soc. Trans. 23*, 341–366. 44

[47] Heinrich, R. and Rapoport, T. A. (1974). A linear steady-state treatment of enzymatic chains: General properties, control and effector strength. *Eur. J. Biochem. 42*, 89–95. 44

[48] Sauro, H. M., Small, J. R., and Fell, D. A. (1987). Metabolic Control and its Analysis: Extensions to the Theory and Matrix Method. *Eur. J. Biochem. 165*, 215–221. 46

[49] Westerhoff, H. V. and Kell, D. B. (1987). Matrix Method for the Determining Steps Most Rate-Limiting to Metabolic Fluxes in Biotechnological Processes. *Biotechnol. Bioeng. 30*, 101–107. 46

[50] Westerhoff, H. V., Hofmeyr, J.-H. S., and Kholodenko, B. N. (1994). Getting to the inside of cells using metabolic control analysis. *Biophys. Chem.* , 273–283. 46

[51] Heinrich, R. and Schuster, S. (1996). *The Regulation of Cellular Systems*. Chapman & Hall, New York. 46

[52] Seydel, R. (1988). *From Equilibrium to Chaos: Practical Bifurcation and Stability Analysis*. Elsevier, Amsterdam. 54

[53] Hofmeyr, J. H. S. (1995). Metabolic regulation: a control analytic perspective. *J. Bioenerg. Biomembr. 27*, 479–490. 55

[54] Poolman, M. G., Olcer, H., Lloyd, J. C., Raines, C. A., and Fell, D. A. (2001). Computer modelling and experimental evidence for two steady states in the photosynthetic Calvin cycle. *Eur. J. Biochem. 268*, 2810–6. 62

[55] Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly, Sebastopol, CA, USA. 70