Simulation and Database Software for Computational Systems Biology: PySCeS and JWS Online

Brett Gareth Olivier

Dissertation presented for the Degree of
Doctor of Philosophy (Biochemistry)
at the University of Stellenbosch

Promoter: Prof J.-H.S. Hofmeyr
Co-promoter: Prof J.M. Rohwer
April 2005

Declaration

I, the undersigned, hereby declare that the work contained in this dissertation is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature

Date

Summary

Since their inception, biology and biochemistry have been spectacularly successful in characterising the living cell and its components. As the volume of information about cellular components continues to increase, we need to ask how we should use this information to understand the functioning of the living cell?

Computational systems biology uses an integrative approach that combines theoretical exploration, computer modelling and experimental research to answer this question. Central to this approach is the development of computational models, new modelling strategies and computational tools. Against this background, this study aims to: (i) develop a new modelling package: PySCeS, (ii) use PySCeS to study discontinuous behaviour in a metabolic pathway in a way that was very difficult, if not impossible, with existing software, (iii) develop an interactive, web-based repository (JWS Online) of cellular system models.

Three principles that, in our opinion, should form the basis of any new modelling software were laid down: *accessibility* (there should be as few barriers as possible to PySCeS use and distribution), *flexibility* (PySCeS should be extendable by the user, not only the developers) and *usability* (PySCeS should provide the tools we needed for our research). After evaluating various alternatives we decided to base PySCeS on the freely available programming language, Python, which, in combination with the large collection of science and engineering algorithms in the SciPy libraries, would give us a powerful modern, interactive development environment.

PySCeS has been designed to run on both Windows and Linux operating systems. More specifically, PySCeS was developed around a 'command line interface' and will run in any type of console that supports Python. This means that it can be included as a modelling component in other Python based applications and should run on any operating system that supports Python. PySCeS has a modular design which allows it to be modified and extended by the user.

Using PySCeS, we investigated the causes of hysteresis in the steady-state behaviour of a linear, end-product inhibited metabolic pathway. We had previously discovered this interesting behaviour but were unable to investigate it with the modelling software available at the time. Using the parameter continuation capability if PySCeS we were able to study and characterise the causes of this interesting behaviour. We also develop a new way of visualizing this behaviour as the interaction between the system's subcomponents.

During the course of our research we noticed that it was extremely difficult to recreate biological models that had been published/described in the literature. This was partially as a result of the fact that no freely available repository existed for cellular models as it did, for example, for genomic and protein sequence and structure information. The JWS Online system was specifically developed to address this issue. JWS Online is a freely accessibly web-based, interactive model repository. It allows a user to interrogate published models via the internet without installing any specialist modelling software. In this way JWS Online has proven to be a valuable resource for both teaching and research.

Opsomming

Sedert hul totstandkoming was biologie en, meer spesifiek, biochemie uiters suksesvol in die karakterisering van die lewende sel se komponente. Steeds groei die hoeveelheid informasie oor die molekulêre bestanddele van die sel daagliks; ons moet onself dus afvra hoe ons hierdie informasie kan integreer tot 'n verstaanbare beskrywing van die lewende sel se werking.

Om dié vraag te beantwoord gebruik rekenaarmatige sisteembiologie 'n geïntegreerde benadering wat teorie, rekenaarmatige modellering en eksperimenteele navorsing kombineer. Sentraal tot die benadering is die ontwikkeling van nuwe modelle, strategieë vir modellering, en sagteware. Teen hierdie agtergrond is die hoofdoelstelling van hierdie projek: (i) die ontwikkeling van 'n nuwe modelleringspakket, PySCeS (ii) die benutting van PySCeS om diskontinue gedrag in ń metaboliese sisteem te bestudeer (iets wat met die huidiglik beskikbare sagteware redelik moeilik is), (en iii) die ontwikkeling vanń interaktiewe, internet-gebaseerde databasis van sellulêre sisteem modelle, JWS Online.

Ons is van mening dat nuwe sagteware op drie belangrike beginsels gebaseer behoort te wees: *toeganklikheid* (die sagteware moet maklik bekombaar en bruikbaar wees), *buigsaamheid* (die gebruiker moet self PySCeS kan verander en ontwikkel) en *bruikbaarheid* (al die funksionalitiet wat ons vir ons navorsing nodig moet in PySCeS ingebou wees). Ons het verskeie opsies oorweeg en besluit om die vrylik verkrygbare programmeringstaal, Python, in samehang die groot kolleksie wetenskaplike algoritmes, SciPy, te gebruik. Hierdie kombinasie verskaf ń kragtige, interaktiewe ontwikkelings- en gebruikersomgewing.

PySCeS is ontwikkel om onder beide die Windows en Linux bedryfstelsels te werk en, meer spesifiek, om gebruik te maak van ń 'command line interface'. Dit beteken dat PySCeS op enige interaktiewe rekenaar-terminaal Python ondersteun sal werk. Hierdie eienskap maak ook moontlik die gebruik van PySCeS as 'n modelleringskomponent in ń groter sagteware pakket onder enige bedryfstelsel wat Python ondersteun. PySCeS is op ń modulêre ontwerp gebaseer, wat dit moontlik vir die eindgebruiker maak om die sagteware se bronkode verder te ontwikkel.

As ń toepassing is PySCeS gebruik om die oorsaak van histeretiese gedrag van 'n lineêre, eindproduk-geïnhibeerde metaboliese pad te ondersoek. Ons het hierdie interessante gedrag in ń vorige studie ontdek, maar kon nie, met die sagteware wat op daardie tydstip tot ons beskikking was, hierdie studie voortsit nie. Met PySCeS se ingeboude vermoë om parameter kontinuering te doen, kon ons die oorsaake van hierdie diskontinuë gedrag volledig karakteriseer. Verder het ons 'n nuwe metode ontwikkel om hierdie gedrag te visualiseer as ń interaksie tussen die volledige sisteem se subkomponente.

Tydens PySCeS se ontwikkeling het ons opgemerk dat dit baie moeilik was om metaboliese modelle wat in die literature gepubliseer is te herbou en te bestudeer. Hierdie situasie is grotendeels die gevolg van die feit dat nêrens 'n sentrale databasis vir metaboliese modelle bestaan nie (soos dit wel bestaan vir genomiese data of proteïen strukture). Die JWS Online databasis is spesifiek ontwikkel om hierdie leemte te vul. JWS Online maak dit vir die gebruiker moontlik om, via die internet en sonder die installasie van enige gespesialiseerde modellerings sagteware, gepubliseerde modelle te bestudeer en ook af te laai vir gebruik met ander modelleringspakkette soos bv. PySCeS. JWS Online het alreeds ń onmisbare hulpbron vir sisteembiologiese navorsing en onderwys geword.

For my parents

Gary and Ronalee Olivier

Acknowledgements

I would like to thank:

Jannie Hofmeyr, a true inspiration, thank you for allowing me and this work to find our own way.

Johann Rohwer, thanks for invaluable discussion and moral support.

Jacky Snoep, for essential collaboration with the JWS Online project.

Norbert and Gisela Kolar and Tom Cunningham, for unwavering friendship during the course of my studies.

All my family that survived the completion of this dissertation: Jill, Lauren, Eugene, my grandparents Ron and Daph José and especially my parents, for always believing in me.

The National Research Foundation and Harry Crossley Foundation for financial assistance.

Contents

1 Introduction

Towards a new biology

Since their inception, biology and biochemistry have been following an essentially modernist program which has been spectacularly successful in opening up the living cell and allowing its components and their relationships to be characterized and understood. Concomitant with the development of an evermore precise understanding of both the metabolic and genetic components that make up the living cell, as well as the processes that produce 'life', there has been a revolutionary development in computer and information technology. With the rapid pace of developments in both these fields—one generating ever larger amounts of data, the other solely in the 'business' of processing and manipulating digital information—it was inevitable that they would fuse and we would begin to see the development of a new *digital biology*.

Although it may seem to contain an inherent contradiction, the mechanization of the living organism is not a new concept. Since the time of Descartes and Newton a body was thought of as a mechanism which possessed some connection to a vital force or soul. Even earlier, the concept of a biological machine found a place in the work of Leonardo da Vinci, who investigated human physiology and described the workings of the limbs and body by way of forces which could be explained in terms of levers and pulleys; his diaries contain detailed plans for what could be described as one of the world's first androids or human machines Although today this may seem more like science fiction than science, we must consider that at the time the human body was regarded as the pinnacle of biological order and form. With today's detailed maps of the components that make up the more intricate cellular puzzle, would Leonardo, if alive, be building

digital cells? As technology has developed, so levers and pulleys have given way to ones and zeros; mechanization is now encapsulated by digitization.

In the broadest possible sense, this introduction sets the context of the work contained in this dissertation in terms of my view of biology and its position in modern science. This view has at its root that there is a rich intersection between the study of biological/cellular and mathematical/technological systems; an intersection that has possibly existed, in some form or another, since the beginnings of modern science. With this as a background, we now move forward a few hundred years to look at the latest (and arguably most exciting) incarnation of this intersection: computational systems biology.

Computational systems biology

The complete sequencing of the human genome marked a turning point in modern biology. For one, the development of highly automated laboratory techniques led to the generation of masses of data describing the various components present in the cellular hierarchy. Subsequently these have been broadly grouped as the 'omics data types' [1] which include, amongst many others, the genome, transcriptome, proteome, and metabolome. However, with the complete mapping/description of a number of 'omes and their storage, mostly, in publicly accessible databases, the question arose. What now?

What was proposed as an answer to the above question that an 'integrative systems approach' needed to be developed. This systems level approach could however be defined in different ways. Genomics with its strong *bioinformatics* component led the way by developing powerful techniques for analysing huge data sets and thereby developing the capability to search for and identify complex relationships contained in genomic and related networks [2]. Soon however, a new term[1], *systems biology* was being used to describe an integrative approach that could be applied to all levels of the cellular hierarchy. Wolkenhauer and Klingmüller capture the spirit of this approach as follows [4]:

[1]This is not to say that 'systems level' thinking in biology is new, having being associated with metabolic modelling since the 1950's. Historical perspectives to modern systems biology can be found in [1, 3, 4].

> In our view, systems biology is about methodologies, i.e., data-based mathematical modelling and simulation, that help an understanding of the dynamic interactions of cells and components within cells. For this vision to succeed, we require foremost experiments and technologies that generate quantitative, time-resolved data.

This definition brings out three strong themes that are associated with systems biology, namely: an integration of the cellular components using a quantitative theoretical framework, the mathematical modelling of such systems, and the verification of these models with an appropriate experimental approach. Implicit in this definition is that there exists (or should exist) the computational theory and tools needed to realize the theoretical framework and enable the mathematical modelling. Kitano includes this explicitly into his definition of *computational systems biology* [5]:

> To understand complex biological systems requires the integration of experimental and computational research — in other words a systems biology approach. Computational biology, through pragmatic modelling and theoretical exploration, provides a powerful foundation from which to address critical scientific questions head-on.

In this dissertation I will be dealing with three aspects of computational systems biology: a 'theoretical exploration' by way of 'pragmatic modelling' which is enabled by the development through 'computational research' of two new modelling applications: PySCeS and JWS Online.

Background and aims of this research

The project which led to this dissertation arose partly from extensive experience over two decades of the members of the Triple-J Group for Molecular Cell Physiology[2] of modelling the metabolic reaction networks in living cells. In fact, the first kinetic modelling and control analysis package to run on a microcomputer was written by a member of this group [6]. Since then, a number of such programs have become available, the most

[2]This research group is part of the department of Biochemistry at the University of Stellenbosch and is led by professors Jannie Hofmeyr, Johann Rohwer and Jacky Snoep.

important of which, for us at least, was Herbert Sauro's Scamp [7, 8] and Pedro Mendes's Gepasi [9, 10]. It is as part of the Triple-J Group that I became familiar with and extensively used these programs (in the case of Gepasi even acted as beta-tester).

It is because we have all been heavy users of these platforms that we realize both their worth and their limitations. Let it be said from the outset that they are excellent programs. Nevertheless, none of them fulfil all of our requirements, both in terms of functionality and philosophy: in general, we have come to realize that we require a simulation platform that not only has a simple design and is platform-independent, but most importantly, is completely under the control of the user. The user should be able to access all the source code, modify it, and extend it; there should be both a low-level and a high-level interface. It is this need for complete low-level control that we as primary users require most and what is lacking in other packages. This meant that we needed access to the source code of the program in order to be able to modify it to our needs, i.e., the program needed to be open-source. Furthermore, instead of a monolithic, compiled program we came to appreciate the power and ease of use of modern, high-level scripting languages such Python, Perl and Tcl. What is particularly useful, is that these languages can "glue" together algorithms from different sources and written in different languages.

This general wish for a new modelling platform would have remained a wish had it not been for a catalyst. In my M.Sc.-thesis [11] we designed, using supply-demand analysis [12] and rate characteristics, an end-product inhibited, linear metabolic pathway that was regulated according to a predefined set of functional criteria (see also, [13]). This pathway, affectionately known as the Stellenbosch Organism, is typical of the amino-acid biosynthetic pathways that have been extensively used as model pathways in the study of metabolic and genetic regulation [14–16]. What was new to our study was the use of the reversible Hill equation [17] to model the committing reaction of the biosynthetic sequence. This rate equation was developed specifically for modelling, and incorporates reversibility, cooperativity and allosteric modification. One important component of this study was to characterize the contributions to the overall regulatory behaviour of these systems of the kinetic parameters of the individual reactions.

During this study I discovered that altering the strength with which the Hill-enzyme

binds its immediate product caused what appeared to be discontinuities in the rate characteristic of the biosynthetic block; such discontinuities typically indicate the presence of multiple steady-state solutions [18]. However, with the simulation software available at the time it was only possible to confirm the presence of a double hysteresis (known as a mushroom) and isola (an isolated branch). Merely to confirm the existence of these hysteresis meant manually collating the data from two completely different simulation programs (a process that could take a few days to complete) and it was therefore impractical to continue with this investigation. However, as this hitherto undiscovered behaviour could be very important in our understanding of metabolic regulation, we decided that the time had come to create a new simulation platform that could be used to simulate such phenomena and at the same time avoid the deficiencies mentioned above that complicated our use of other programs. Therefore, our initial aim for this project was:

> To develop a new modelling application that not only incorporated numerical algorithms with which to characterize and explain discontinuous behaviour, but also realised our general ideals for a modelling program.

Our first quest was to find a suitable, open-source programming environment. We considered many alternatives and concluded that the programming language Python [19] was best suited to our purposes. Python is (i) simple and clear, but extremely powerful, allowing both procedural and object-oriented programming, (ii) easily extensible with wrapper generators such as Swig [20] and f2py [21], and (iii) has a very powerful numeric extension and an existing scientific computing add-on called SciPy [22] that already contains industrial strength ordinary differential and nonlinear equation solvers, in addition to a number of graphing packages. There are already two other simulation packages that use Python: the Systems Biology Workbench [23] allows scripting in Python, and ScrumPy [24] is Python-based. Chapter 3 describes the initial exploration of the suitability of the PySCeS/SciPy combination for cellular modelling (see the next section for description of the contents of all chapters).

The second aim of this study was formulated while PySCeS was being developed. We became aware that although there were publicly available databases that stored genomic and enzyme (generally pathway or structural) information, there was no similar

resource for the storage of the kinetic models typically used in computational systems biology (CSB). This was related to a more practical problem: it was extremely difficult to recreate and run a model based purely on the information typically published in the literature. As discussed earlier in this chapter, quantitative modelling and, by implication, the exchange and evaluation of models is a key methodology in CSB. In order to address these issues the second aim of this project was formulated as:

> Develop an online, interactive database of models that could act as a repository for the computational system's biology community.

It is worth noting, that at the time there was no example of an online, interactive database for biological models.

Chapter outline

This dissertation is divided up in the following manner:

Chapter 2 gives a brief introduction and overview of kinetic modelling, metabolic control analysis, and modelling applications.

Chapter 3 shows by way of example how the steady-state and time-dependent behaviour of a kinetic model can be investigated using only a programming language, Python, and its scientific libraries, SciPy. The original work from this chapter has been published as [25].

Chapter 4 looks in more detail at some of the current modelling applications and motivates why new modelling software needed to be developed from scratch.

Chapter 5 shows the theory behind, implementation and complete user interface of the Python Simulator for Cellular Systems, abbreviated as PySCeS. The original software and concepts developed in this chapter have been published as [26].

In Chapter 6, using the newly developed PySCeS software, we answer the original and new questions about hysteresis formation in a linear end-product inhibited pathway.

Chapter 7 switches focus to the Java Web Simulation Online (JWS Online) software, an interactive web-based modelling application. The original software and concepts developed in this chapter have been published as [27–29].

Chapters 9, 10, 11 and 12 are appendices to the preceding chapters, and contain supplementary information, model descriptions and working PySCeS code examples. The publications resulting from the work presented in this dissertation are included as Chapter 14.

2 A brief introduction to computational systems biology

In order to work with a computer representation of a biological system we need to use a formalism in which the cell's components and interactions can be quantified and described mathematically. In this chapter two such theoretical frameworks are introduced.

- The time-dependent and steady-state behaviour of a network of coupled reactions can be captured in a *kinetic model*.
- The control and regulatory properties of a cellular system can be quantified using a framework such as *Metabolic Control Analysis*.

Throughout this dissertation frequent use will be made of the term 'cellular system'. This concept should be thought of as a set of cellular components, linked by some sort of interaction, which is not necessarily situated in any single level of the cellular hierarchy. For example, a metabolic system is concerned only with interactions between enzymes and metabolites. On the other hand, a genetic system is made up of genes and genetic interactions. We regard a cellular system as either of the aforementioned systems, or as one that could contain both metabolic and genetic components.

2.1 The kinetic model

The construction and use of mathematical models in biology is an extensive topic which has been the subject of many thorough reviews [30–38], and several books and book

chapters [39–45]. The type of cellular system models introduced in this chapter will be restricted to open (i.e. non-equilibrium) enzyme catalysed reaction systems. Although it is possible to model enzymatic reaction networks using non-equilibrium thermodynamics [46], we will concentrate exclusively on kinetic models (see Eqn. 2.1) where the change in concentration of each variable component in the system ($\frac{ds}{dt}$) is a linear function of the rates of the reactions which either create or consume it (the product of the stoichiometric matrix, $\mathbf{N}$ and the vector of reaction rates, $\mathbf{v}$).

$$\frac{ds}{dt} = \mathbf{N}\mathbf{v} \tag{2.1}$$

The stoichiometric matrix captures the topology of the reaction network by describing for each metabolite the reactions in which it is either produced or consumed according to a fixed stoichiometry. This will be discussed in more detail in In Section 5.5. The enzyme rates $\mathbf{v}$ are commonly expressed as aggregate rate laws [31, 47, 48] which are generally non-linear. This means that the differential equations describing the kinetic model represented by Eqn. 2.1 are also non-linear and generally have no analytical solution. They can, however, be efficiently analysed numerically [49, 50].

In Section 5.5 we will look in more detail at the formulation of the kinetic model in terms of ordinary differential equations (ODEs). For now it is sufficient to note that ODE-based models are well suited for studying both the dynamic and steady-state behaviour of cellular systems. At the time when kinetic modelling was being pioneered by workers such as Garfinkel [51, 52], Chance [53–56] and others, another theoretical framework was developing that succeeded in quantifying the control and regulation of metabolic systems at steady state. This framework, called Metabolic Control Analysis (MCA) will be introduced next.

2.2 Metabolic Control Analysis

Kinetic (ODE based) models are usually studied in one of two ways: either by integrating the differential equations to determine their time-dependent behaviour, or by making the so-called steady-state assumption where

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

The steady state is the final state that the systems tends to asymptotically[1]. In steady state the kinetic model reduces to:

$$\mathbf{N}\mathbf{v} = 0 \tag{2.2}$$

which can be solved for the steady state metabolite concentrations and fluxes by using numerical non-linear solvers. With such a solution in hand it is now possible to ask how sensitive the steady-state is to perturbations in its parameters, thus making it possible to quantify the degree of control of, say, one reaction on a steady-state metabolite concentration or a particular system flux.

This quantification of control and its understanding in terms of the properties of the individual steps is the province of MCA [57–63]. More specifically, MCA allows one to relate the global (or systemic) responses of the system at steady state to parameter perturbations (quantified in terms of so-called response or control coefficients) to local properties of the individual steps of the network (the so-called elasticity coefficients, or variable kinetic orders). The definitions of these coefficients will be examined in more detail in Chapter 5. An extensive body of literature has been developed around MCA and the related framework of Biochemical Systems Theory [42, 64–66]; it includes books and collections such as [67, 68] and papers covering topics such as moiety conserved cycles [69–71], the relationship between the elasticities and control coefficients [72, 73], metabolic regulation [12, 74–76], multi-level or hierarchical control [77] and a variety of mathematical formulations of the central concepts of MCA [78–82]. This list is neither intended to be complete nor exhaustive, and does not do proper justice to many applications of MCA such as the top-down [83] and modular [84] approaches.

As stated above, the ODEs that describe a biological system's behaviour and the rate equations of the enzymes themselves are usually highly non-linear, making it practically impossible to solve these systems analytically. This, coupled with the fact that calculating the elasticities means partially differentiating the rate equations, means that specialist modelling software is needed to simulate the time-dependent and steady-state behaviour and the associated control profiles for all but the simplest systems.

[1] In dynamical systems the steady state is generally called a 'fixed point'. It is but one of number of possible attractors; other examples are cyclic attractors that oscillate or strange attractors that show chaotic behaviour.

2.3 Modelling applications

At the time when metabolic modelling was first pioneered, many of the first models were built on analog computers [55, 85]. However, digital computers soon became the standard technology and building a model most probably entailed being not only a biologist but also a specialist in numerical analysis and FORTRAN programming. Some of the first simulation packages such as FACSIMILE [86] and BIOSSIM [87] are examples of modelling packages developed for mainframe computers.

As computer technology developed and the personal computer (PC) became more widely available it was a natural development that modelling and control analysis software was developed for this new platform. METAMOD [6] consists of two sub-packages METADEF and METACAL. METAMOD was written in BASIC and developed for the BBC microcomputer. With the rapid development computer hardware and programming architectures new simulators and special purpose software such as SCoP [88], SIMFIT [89] and CONTROL [90] became available. It was, however, with the development PC's based on the Intel 386 architecture that some of the benchmark modelling applications were developed. These included MetaModel [91], Scamp [7, 8] and Gepasi [9, 10] (Gepasi and newer modelling software will be again be discussed in Chapter 4).

In general, as far as program operation and user interaction is concerned, Scamp and Gepasi represent two different design strategies. On the one hand Scamp was a *command line* simulator where the model was defined using a specific syntax as an input file which also contained the command that specified the type of analysis that was required (simulation, steady state etc.). This input file was then compiled and run in separate steps after which the results stored in data output files. On the other hand Gepasi, which was originally released for DOS and Windows 3.1 and then again for Windows 95 as Gepasi 3, used an *interactive* graphical frontend (GUI) where models could be defined and analysed interactively without any obvious intermediary steps.

One of the major strengths of a GUI based application is that it is easier to learn and has all major functionality, such as model definition, analyses selection and result plotting available in a single unified system of menus or panels. Although they have a slightly steeper learning curve, command line simulators have the advantage that model descriptions can be easily modified and information stored as comments in the input

files. This means that a modelling session or series of modelling experiments could be saved for future reference. Whether a GUI or command line simulator should be used often simply comes down to personal preference.

As we shall see in Chapters 4 some of the newer metabolic simulators such as Jarnac [92], ScrumPy [24] and PySCeS (introduced in Chapter 5) attempt to bridge this gap and are based on an *interactive command line* design. Later in Chapter 7 a new type of GUI-based application is developed: the JWS Online web-based interactive modelling environment and model repository.

Before looking at these new applications, let us first take a step back in Chapter 3 and examine the feasibility of building kinetic models directly using a modern object oriented programming language, Python.

3 Modelling cellular systems with Python and SciPy

This chapter[1] investigates the viability of modelling cellular systems directly using the programming language Python combined with the Scientific Libraries for Python (SciPy). It also introduces the concept of a Python-based modelling application which is further developed in Chapters 4 and 5. To begin with Python and SciPy are considered in more detail.

3.1 Overview: Python and SciPy

3.1.1 Python

Python is a modern computer programming language that is becoming widely used by both business and scientific institutions (examples[2] include IBM, Google, Nasa, the Lawrence Livermore National Laboratories and Los Alamos National Laboratory) to develop real world applications.

Many excellent resources, both on and off line, are available which can be used to learn [93–96] and develop [19, 97–101] applications using Python. Some of the advantages of using Python, which also make it well-suited as a programming language for computational systems biology, include:

- *Python is easy and fun to work with.* Although this claim is made by other lan-

[1] Parts of which (Sec. 3.2) have been published in [25]

[2] As listed by the Python Business Forum `http://pbf.strakt.com/success`

guages our (and others [93, 102]) experience has been that the Python syntax and coding style is easily learned and is accessible to 'non-programmers'.

- *Python is interpreted and interactive.* Much of Python's accessibility comes from the fact that it is run by an interpreter which can be used interactively. No compiling or linking steps are needed between program coding and execution (as is the case with Java, Fortran or C).

- *Python is portable.* The Python interpreter is available on a wide range of operating systems including Windows, Linux, MacOS, SunOS, IRIX and OpenVMS [103]. Python code developed on one operating system should, in general[3], run on any other operating system that supports the Python interpreter.

- *Python can act as a glue.* Using tools such as SWIG [20] and F2PY [21], which automatically generate Python interfaces to C and Fortran libraries, Python can act as a 'glue' and use high performance libraries written in both these languages [104].

- *Python has an extensive standard library* which eases application development by including (amongst other things) efficient modules for networking, operating system, data manipulation and testing [105].

It is, however, Python's high-level, interpreted nature that gives rise to some of its most powerful features. This includes a memory manager and garbage collection system that automatically allocates and reclaims system memory. Python also provides high-level data types, such as lists and dictionaries, which may simultaneously contain many different types of objects. For example, a single list can contain both integers and strings. It is also possible to extend Python's numerical capabilities by using powerful numerical extension libraries. In the next section an example of one such a library, SciPy, is looked at in more detail.

[3]Code portability is dependent on the use of external libraries and operating system specific code.

3.1.2 SciPy

SciPy is an Open Source library of high-level tools and algorithms that is geared towards developing science and engineering applications [22]. It is freely available under a BSD type licence and has an active community of developers and users from a wide variety of backgrounds. This ensures the development of a diverse collection of high quality tools that are continuously being debugged and improved.

Besides specialist data types and functions (based on the widely used Numeric Python[4]), SciPy provides a number of modules with a wide variety of useful functionality[5]:

- Linear algebra (`scipy.linalg`) and BLAS routines based on the ATLAS[6] implementation of LAPACK[7], including some sparse matrix support.
- Numeric routines for integration (`scipy.integrate`), constrained and unconstrained optimization methods and ODE solvers (`scipy.optimize`).
- Statistical (`scipy.stats`) and interpolation routines (`scipy.interpolate`) including an interface (via the `rpy` package) to the R statistical software.
- A fast fourier transform module (`scipy.fftpack`).
- Plotting modules based on wxPython (`scipy.plt`), GIST (`scipy.xplt`) and GnuPlot (`scipy.gplt`).

SciPy is being developed as a Python-based equivalent of a general data processing workbench such as Matlab, Octave or SciLab [106]. SciPy can either be downloaded[8] and installed as a binary distribution or compiled and built from scratch. Detailed instructions for building and installing SciPy on both Windows and Linux-type systems are available from the main SciPy website[9].

[4] `http://numpy.sourceforge.net/`
[5] More detail can be found in the SciPy FAQ, `http://www.scipy.org/About/FAQ.html`
[6] `http://math-atlas.sourceforge.net/`
[7] `http://www.netlib.org/`
[8] `http://www.scipy.org/download/`
[9] `http://www.scipy.org/documentation/`

Although the SciPy documentation is continually being developed (a tutorial is available online [106]), general resources on matrix and linear algebra, [107, 108], ordinary differential equations [109, 110] and numerical analysis [111–113] can effectively be used in conjunction with the source code documentation to develop new applications. The remainder of this chapter discusses one such application – building and analysing a metabolic model.

3.2 Modelling with Python and SciPy

The text and figures in this section are reproduced as originally published in [25]. Only the reference and figure numbers have been changed to be consistent with this chapter.

3.2.1 Introduction

With the rapid increase in the number of pieces in the puzzle that is the living cell, the need for tools that paste them back together again has become paramount. Computer software for modelling these complex cellular systems forms an indispensable part of this toolkit.

In this paper we show how it is possible to model a metabolic reaction network directly using a modern computer language. We use Python, an interpreted scripting language with an intuitive, high-level interface well suited to numerical programming. One of the most remarkable features of Python is its ability to interface with code written in other languages. This ability is particularly useful since many tried and tested numerical routines are written in Fortran or C. In the next section we shall, using only Python and SciPy (a library of routines for scientific computing), construct a simple metabolic model, perform a time-course simulation, solve for the steady-state concentrations and generate a parameter portrait. We also introduce a new modelling tool called PySCeS, short for Python Simulator of Cellular Systems, which is being developed by our group.

3.2.2 Modelling with Python and SciPy

The model [82] consists of a system with a branch and a two-member moiety conserved cycle with the reaction network (Fig. 3.1).

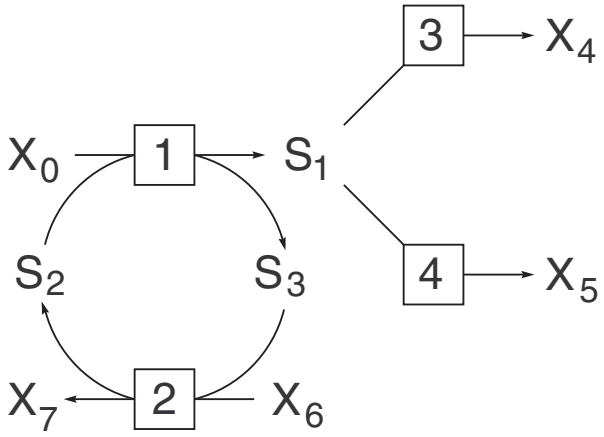


Fig. 3.1 A example reaction network.

The first step is to import the SciPy module and define the rate vector as a function of the concentrations and parameters (in the code samples that follow, tab spaces have been reduced; the fully formatted source code is given in Appendix 9).

```
import scipy

v = scipy.zeros((4),'d')    # create rate vector
def rate_eqs(S):
  # function with rate equations
  v[0]=V1*X0*S[1]/K1_X0*K1_S2/ \
       (1+X0/K1_X0+S[1]/K1_S2+X0*S[1]/K1_X0*K1_S2)
  v[1]=V2*S[2]*X6/K2_S3*K2_X6/ \
       (1+S[2]/K2_S3+X6/K2_X6+S[2]*X6/K2_S3*K2_X6)
  v[2]=V3*S[0]/(S[0]+K3_S1)
  v[3]=V4*S[0]/(S[0]+K4_S1)
  return v
```

Next the differential equations are defined and the dependent metabolites are calculated, returning a vector $\frac{dS}{dt}$. Note that the full ODE system is overdetermined because S2 and

S3 form a moiety-conserved cycle; S3 has been chosen as the dependent metabolite and its concentration is updated by the conservation equation. The only reason that the ODE for S3 appears is to obtain its concentration as part of the output vector:

```
def diff_eqs(S,t):
  Sdot = scipy.zeros((3),'d')  # create ODE vector
  S[2] = S_tot - S[1]          # calculate dependent S3 conc

  v = rate_eqs(S)              # calculate rates

  # Differential equations
  Sdot[0] = v[0] - v[2] - v[3] # dS1/dt
  Sdot[1] = v[1] - v[0]        # dS2/dt
  Sdot[2] = v[0] - v[1]        # dS3/dt (only for output)
  return Sdot
```

The concentration time course is generated with the SciPy LSODA ODE-solver. Although here we accept the default parameter values of this solver, the interface allows the user to set any of these parameters through optional arguments to the `odeint` function:

```
# create range of time points
t_start = 0.0; t_end = 10.0; t_inc = 0.1
t_range = scipy.arange(t_start, t_end+t_inc, t_inc)

t_course = scipy.integrate.odeint(diff_eqs,\
                                  [S1,S2,S3],t_range)
```

Similarly, the steady-state concentrations are calculated with the SciPy HYBRD nonlinear solver using the final value of the time course as starting value. As with `odeint`, any parameter of this solver can be set by the user:

```
fin_t_course = scipy.copy.copy(t_course[-1,:])
ss = scipy.optimize.fsolve(diff_eqs,fin_t_course,args=None)
```

The steady-state flux is calculated from the steady-state concentrations

```
J = rate_eqs(ss)
```

A parameter scan (here of V_4) repeatedly calls the steady-state solver over a parameter range. The output of each step is used to initialize the following step.

```
# set up scan range
S_start = 0.0; S_end = 20.0; S_inc = 0.1
S_range = scipy.arange(S_start,S_end+S_inc,S_inc)

ss_init = scipy.copy.copy(fin_t_course)
scan_table = scipy.zeros((len(S_range),5), 'd')

# scan V4
for i in range(len(S_range)):
  V4 = S_range[i]
  ss = scipy.optimize.fsolve(diff_eqs,ss_init,args=None)
  J = rate_eqs(ss) # calculate flux
  scan_table[i,0] = S_range[i]
  scan_table[i,1:] = J
  ss_init = ss
```

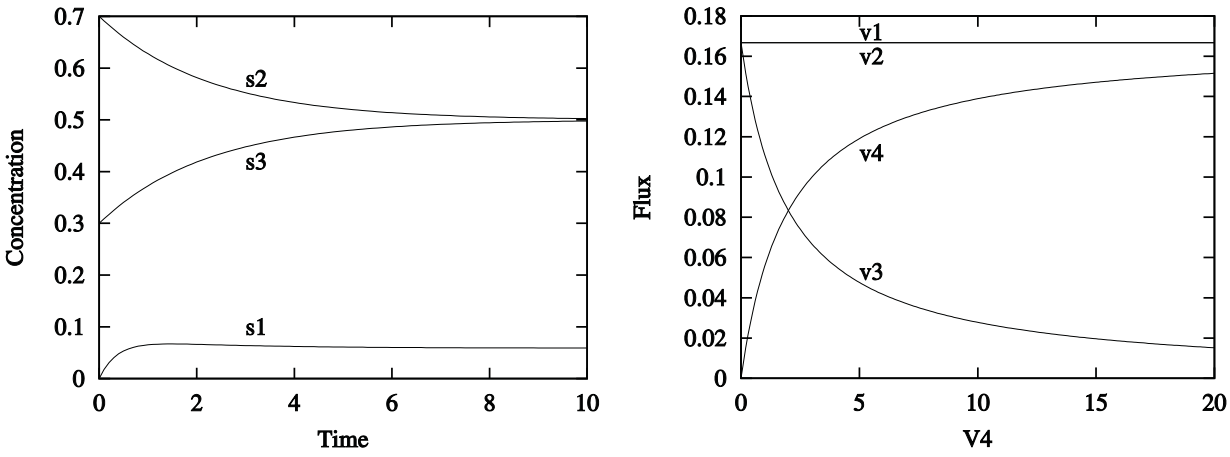
Results are visualized with the SciPy interface to the Gnuplot graphics program (Fig. 3.2). For example, the time course plot is generated by:

```
from scipy import gplt

gplt.plot(t_range,t_course[:,0],t_range,t_course[:,1],\
         t_range,t_course[:,2])
gplt.xtitle('Time'); gplt.ytitle('Concentration')
```

3.2.3 Discussion

From the above it is clear that Python/SciPy on its own already provides a simulation platform powerful enough to create models of arbitrary complexity. Python [94, 114]



a. Time simulation

b. V_4 scan of steady-state fluxes

Fig. 3.2 Examples of the graphical output of the Python/SciPy model described in the text.

is a simple and clear, but extremely powerful, scripting language, allowing for both procedural and object-oriented programming. Moreover, it is open-source and therefore freely available. It is also easily extensible with wrapper generators such as Swig [20] and f2py [21], it has a very powerful numeric extension, and an existing scientific computing add-on SciPy [22] that already contains industrial strength ODE and nonlinear solvers, in addition to a number of graphing packages. However, the procedure presented is inefficient and cumbersome for large models. The user has to derive and code the ODEs and conservation equations by hand, and concentration variables have to be referred to by indexed vector variables instead of ordinary chemical names. This may cause problems for the naive user. We are therefore simplifying and automating the process of modelling by providing high-level tools in a program suite[10] called PySCeS, which include packages that:

- parse a model definition in terms of reaction and rate equations from which the stoichiometric matrix that describes the network topology is constructed;
- analyse the stoichiometric matrix for relationships between steady-state fluxes and between the differential equations (from which conservation constraints are deduced);
- determine the elementary flux-modes of the system;

[10]Described in Chapter 5

- serve as front-ends to the ODE and nonlinear solvers;
- calculate elasticities by differentiation of the rate equations;
- calculate control coefficients;
- do a bifurcation analysis using existing continuation algorithms;
- serve as front-ends to optimization algorithms;
- interface to various graphing packages.

It could be argued that the creation of yet another simulation platform is unnecessary, since there are already a number available: for example, Gepasi [10] (or its latest reincarnation, Copasi[11]), Scamp [7] and its successor Jarnac [92] (part of the Systems Biology Workbench project [23]), Scrumpy [24], and DBSolve [115], to name the ones that we are most familiar with. It is because we have been heavy users of these programs that we realize both their worth and their limitations. Although these are all excellent programs, none of them fulfil all of our requirements, both in terms of functionality and philosophy: we require a simulation platform that not only has a simple design, runs under all the major operating systems, and is open source, but most importantly, is completely under the control of the user, both at a low and a high level[12]. It is the need for complete low-level control that we as primary users require most and what is lacking in other packages.

Having access to a simulation platform is, however, not enough. Modelling cellular processes can be very difficult; for the beginner there is very little available in the form of teaching materials. Another envisaged feature that makes this project unique is the provision of an extensive set of freely-accessible web-based tutorials that teach the art of modelling using PySCeS as platform, both in terms of the underlying theory and the construction of models and modelling tools.

[11] `http://www.copasi.org/`

[12] This is further explored in Chapter 4

4 Developing new modelling software

In this chapter we will consider the design principles and motivations that inspired the development of a new modelling application. While Chapter 3 can be seen as laying the foundations for our new software, here we explore these ideas and develop the blueprint for the Python Simulator for Cellular Systems: PySCeS.

4.1 Our design principles

Over the past few decades there has been a rapid increase in our knowledge of the components that make up the cell and new strategies must continuously be developed to try and make sense of how these components work together. Through the increasing use of computational models, the pursuit of a systems level understanding of the cell is not simply a pure academic exercise, but is beginning to find application in fields as diverse as agriculture and medicine [37, 116–119]. In this context, we should take into consideration that modelling investigators are increasingly likely to be found working in a diverse range of computing and socioeconomic environments. Any new application should try to take this into consideration and our first design principle is therefore simply:

> There should be as few constraints as possible to the use of and distribution of the software.

The next design principle is related to the first but deals with the structure and design of the program itself. As already mentioned, our expanding knowledge of cellular systems is leading to a concomitant increase in the theory (or theories) used to understand and

model them. As our theoretical knowledge develops, so to must the software needed to study it. This is summarized in the second design principle.

> The software should be built using an open architecture, facilitating adaptability and future extension.

Perhaps of most importance, the new software is meant to be a research tool. As such it should at least be able to analyse a kinetic, ODE based model's behaviour over time and solve for its steady-state solution. Any new software should be able to analyse the structural aspects of a system [78] allowing for non-integer stoichiometry, perform a Metabolic Control Analysis (MCA) [60, 61] and investigate systems with multiple steady states by the application of both numerical continuation techniques (for an overview of numerical continuation see [120]). This is summarized by the third design principle.

> The new application should provide us with an interactive environment and tools needed to perform our research.

In the following section the software applications discussed in different types of software with specific reference to our design principles. In no way is this intended to be an extensive, or detailed, comparison of modelling applications.

4.2 General mathematical workbenches

Matlab[1], Mathematica[2] or their open source cousins GNU Octave[3] and SciLab[4] all can be classified as general mathematical or scientific workbenches. They are all designed to provide a general purpose mathematical environment (Mathematica specializes in providing an environment capable of symbolic computation) with easy access to numerous mathematical/scientific algorithms and functions.

Mathematica and Matlab run on a variety of operating systems but both are commercial software packages and have strict (and expensive) licensing agreements covering their usage and redistribution. On the other hand Octave and SciLab are open source

[1] `http://www.mathworks.com/`
[2] `http://www.wolfram.com/`
[3] `http://www.octave.org/`
[4] `http://www.scilab.org/`

software that run on both Windows and Linux. All these mathematical workbenches are, however, designed primarily for numerical analysis and investigation. While well suited to the study of individually coded models they are not well suited for use as a programming environment in a general purpose modelling application.

4.3 Specialist modelling software

Two broad categories of stand-alone modelling applications currently exist: stand-alone software applications and web-based modelling resources (such as the JWS Online system described in Chapter 7). In this section we will consider only stand-alone software.

There is a strong tradition of excellent software capable of modelling cellular systems using the MCA framework. These include MetaMod written for the BBC Micro [6] and MetaModel [91]. The next generation of software includes Scamp [7], which runs on the Microsoft DOS operating system and uses a script based interface (subsequently updated for use with newer versions of Microsoft Windows as WinScamp[5]), Gepasi [9] (now in beta-release as Copasi[6]) which utilizes a graphical user interface and runs on Microsoft Windows and DBsolve [115]. Below we will look in more detail at some of the current, MCA capable, modelling software: Gepasi 3 [10], Scamp [7, 8], Jarnac [92] and ScrumPy [7].

In each case the modelling application will be considered in terms of the following: its user interface, the operating system on which it runs, the functionality it provides, and its design as related to our design principles.

4.3.1 Gepasi 3

User interface

Gepasi[8] is a widely used, Microsoft Windows compatible, modelling package [10]. It has a fully integrated graphical user interface (GUI) in which models are described, analysed

[5] http://www.cds.caltech.edu/~hsauro/Scamp/scamp.htm
[6] http://www.copasi.org/
[7] http://bms-mudshark.brookes.ac.uk/ScrumPy/
[8] http://www.gepasi.org

and results plotted. Gepasi has a tabbed appearance which groups functionality by analysis type (i.e. simulation, optimization etc.). This makes Gepasi well suited for both teaching and research purposes.

Functionality

Gepasi can perform time simulation, solve for steady states and perform control analysis (MCA). It also includes facilities for performing parameter scans as well as parameter fitting and optimization. Gepasi can import and export models as Systems Biology Markup Language (SBML) [121] level 1 files and can calculate elementary modes.

Design

Gepasi is freely available for use but designed to run only on Windows operating systems and is closed source and not user extendable, this limits its flexibility. As the user provides all the input interactively it is not possible to script a modelling sessions. We have used Gepasi extensively for modelling and only found it restrictive when we needed to do something with a model that the developer had not anticipated. Many of these issues are being addressed with the newest incarnation of Gepasi, namely Copasi[9].

4.3.2 Scamp

User interface

Scamp is a widely used, scripted modelling application. It uses a text file as input in which the model description, operations that needed to be performed and data output format are described. This file is compiled and executed and the results returned. Recently Scamp has been updated to WinScamp[10]), which includes a GUI based model editor and enhanced plotting facilities.

[9] http://www.copasi.org
[10] http://www.cds.caltech.edu/~hsauro/Scamp/scamp.htm

Functionality

Scamp can perform time simulations, solve for a steady state and perform control analysis and perform parameter scans. It includes one of the early continuation methods (the Kubicek algorithm [122]) for the analysis of systems that exhibit multiple steady states. Scamp does not have support for elementary modes or SBML.

Design

Scamp is only available for Windows systems and is closed source which limits both its accessibility and flexibility. However, the use of a 'script' style input file and the continuation capability made Scamp one of our favourite modelling applications.

4.3.3 Jarnac

User interface

Jarnac[11] is the successor to Scamp and extends the Scamp design by introducing the concept of a scripted modelling interface or modelling language. Model scripts can be run in Jarnac as well as manipulated interactively. Its GUI includes both a model editor, interactive console for real time analysis of models as well as a plotting interface. Jarnac is targeted towards the more advanced modeler.

Functionality

Jarnac has facilities to perform time simulations, calculate steady states and perform MCA. It also includes the Kubicek continuation algorithm [122], can calculate elementary modes and is SBML level 2 compatible. Jarnac is the only modelling software in this comparison that has a network capability and has been designed to interoperate with other software. Jarnac can act both as a server, via a direct TCP/IP socket connection to a user defined client, or as a component in the Systems Biology Workbench (SBW) [23].

[11] `http://www.cds.caltech.edu/~hsauro/Jarnac.htm`

Design

Jarnac is currently only available for Windows and is distributed under a non-commercial, open source licence. However, it is written in Pascal, using the commercial Delphi development environment. This unfortunately means that it is difficult for the user to modify and extend the software and places potential restrictions on its flexibility.

4.3.4 ScrumPy

User interface

ScrumPy[12] is an interactive modelling program which uses Python as a working environment. It uses an enhanced Idle console for real time model analysis, editing and plotting. Idle is the default integrated development environment that is distributed with Python.

Functionality

ScrumPy provides facilities for time integration, solving for steady states and MCA. It can calculate and manipulate sets of elementary modes. ScrumPy provides a facility for model fitting and optimization and does not provide support for SBML.

Design

ScrumPy is open source and distributed under the GNU general public license, it currently runs on RedHat Linux and Solaris operating systems which limits its wider accessibility. Being based on Python, ScrumPy provides the user with a flexible working environment and allows it to be user extendable.

4.3.5 Conclusion

> "every good work of software starts by scratching a developer's itch" – Eric Raymond [123]

As we have seen in the previous sections, excellent software exists for the modelling and analysis of cellular systems. However, no single application properly satisfies all of our

[12] `http://bms-mudshark.brookes.ac.uk/ScrumPy/`

particular design principles. Paraphrasing (badly) Eric Raymond, none of the software that we investigated managed to sooth the itchiness of our modelling needs.

4.4 Introducing PySCeS

Realistically, the only option left to us was to develop a new modelling application from scratch. As there were a bewildering array of possible programming environments, languages and architectures, we decided to use an environment that we had found imminently suited for scientific programming, Python [114] with the SciPy scientific libraries (see Chapter 3).

Both Python and SciPy run on a variety of operating systems and are freely available and so satisfy our design principle of software accessibility. The scripted, object oriented nature of Python allows for the design of user extensible software without the prerequisite of having to know a language such as C or C++. Of course this does not apply to the low level numerical algorithms (such as LAPACK) used by SciPy which are implemented as either wrapped C or Fortan libraries for performance reasons.

> "Good programmers know what to write. Great programmers know what to rewrite (and reuse)." – Eric Raymond [123]

This is one of the principal reasons we decided to use SciPy as a base for our new application. By leveraging the numerical capabilities provided by SciPy we could start developing our own application without first having to wrap external libraries and debug interfaces to low level algorithms. In Chapter 5 we will see how SciPy also provides a operating system independent way of building and distributing customized Fortran extension libraries (a capability not provided by Python).

Using Python/SciPy as a development platform, our new application was born:

PySCeS – the **Py**thon **S**imulator for **Ce**llular **S**ystems[13]

By choosing Python/SciPy as a basis for PySCeS an environment was created that could satisfy two of our design principles, namely those of accessibility and flexibility.

[13]Name and acronym courtesy of Jannie Hofmeyr.

4.4.1 The Open Source development model

With the success of Linux and the Open Source development model and the fact that Python based programs are not compiled and effectively distributed as source code, it seemed logical to also use this strategy for PySCeS. This is especially true where software testing and debugging is concerned.

> "Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging" – Eric Raymond [123]

With a diverse potential user community, working on many different applications of computational systems biology, leveraging the testing, debugging and development potential of the open source process seemed to be a win–win situation.

In order to distribute PySCeS while still maintaining authorship we have licensed PySCeS using the GNU General Public Licence[14] (GPL) which is designed to effectively keep software free from restrictions on its use. Using the GPL ensures that the software source code will always be available for inspection, modification or derivation.

- Anyone is free to distribute the software providing that the software itself is not charged for and the source code is distributed with any binary distribution.
- Copyright is maintained by the author(s) and any derivative works must be distributed under the same licence with any changes from the original work clearly indicated.
- The source code should be made publicly available for at least three years.

With the exception of specific external libraries dealt with later on in this chapter, all software used with PySCeS is distributed under an Open Source compatible licence.

There is one perceived disadvantage to using the GPL and this is its so-called 'viral' property. In essence this means that any application using code derived from PySCeS must itself be licensed using the GPL, this might make developers wary of extending PySCeS or integrating it into other applications. An alternative to the GPL is the Lesser GPL or BSD style licence which while providing an Open Source copyright does not have

[14] `http://www.gnu.org/licenses/gpl.html`

this 'viral' distribution property. The first major advantage of using the GPL was that we were then free to use GPL'd code in PySCeS. Additionally, any enhancements that might be made by derivative software could be fed back into the PySCeS code-base. Of course, if using the GPL becomes an obstacle to the usage or extension of PySCeS by the systems biology community, a change to another form of open source licence might be considered.

Eric Raymond's collection of essays *The Cathedral and the Bazaar* [123], also available from his web site[15], make excellent reading and cover many aspects of the Open Source development model.

An important aspect of the GPL is the source code availability; in order to ensure that PySCeS is as widely accessible as possible it has been registered with SourceForge[16], one of the largest open source software repositories based in the USA. Projects hosted by SourceForge get access to:

- space for project web site: `http://pysces.sourceforge.net`,
- a file release system[17] which archives all file releases indefinitely,
- mailing lists and CVS code repository.

In the next chapter we will look at how PySCeS is implemented, as open source software, using Python and SciPy.

[15] `http://www.catb.org/~esr/writings/cathedral-bazaar/`
[16] `http://sourceforge.net`
[17] `http://pysces.sourceforge.net`

5 PySCeS: the Python Simulator for Cellular Systems

In this chapter 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.

5.1 Installing PySCeS

PySCeS is implemented as a Python package and is installed using the Python Distributions Utilities (Distutils) [124]. Distutils allows for the installation of Python scripts

[1] PySCeS has been co-developed with Jannie Hofmeyr and Johann Rohwer. Parts of this chapter have been published in [26]

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 3 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`

5.1.1 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.

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 10.1 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.

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

- `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.

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.

Building MetaTool binaries

MetaTool [125] 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 10.2) to enable

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

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 10.1).

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.

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 10.3 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 chapter.

Compiling extension modules

Finally, setup attempts to build the two Fortran libraries using the scipy_extension mechanism and F2PY [21]. 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

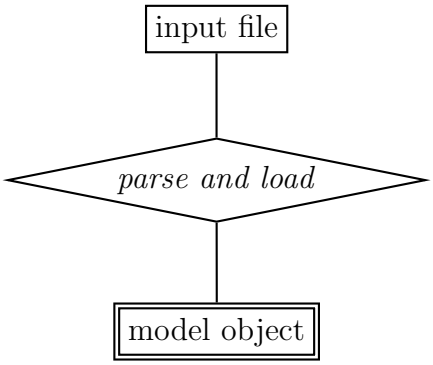


Fig. 5.1 *Loading a PySCeS model.* The input file is parsed and converted into a Python object

is stored first [126]. 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 manually install the file `nleq2.f` which is the source file used by Distutils to build the NLEQ2 extension module.

5.2 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. 5.1 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.

5.3 The PySCeS model input file

The PySCeS *input file* is an ASCII text file, similar to those used by Scamp [8], Jarnac [92] and ScrumPy [24], 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

[4] More information can be found in the NLEQ2 source code `nleq2.f` distributed with PySCeS or from ZIB (`http://www.zib.de/SciSoft/ANT/nleq2.en.html`)

portability between Windows, Macintosh and Linux systems, filenames must end with the extension (i.e. final four characters): `.psc`

5.3.1 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:

```
# 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
```

5.3.2 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.

5.3.3 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 10.4.

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:
```

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
```

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 5.1.

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

Table 5.1 Operator precedence increases from top to bottom and left to right, adapted from the *Python Language Reference* [19].

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)
```

5.3.4 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.

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
```

Initializing variable metabolites

The initial concentrations of the models variable metabolites are used to calculate the moiety conserved sums [127] (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.

5.4 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 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

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

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 5.1.1 of this chapter described

how the PySCeS configuration file (`pyscfg.ini` see Appendix 10.3) 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 10.6.

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 5.19.

5.4.1 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.

5.4.2 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
```

5.4.3 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.

`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. 5.4.4.

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.

```
>>> mod.eModeExe_dbl
```

[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.

```
'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.

`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 5.13.

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

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

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 5.6.3) 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.

5.4.4 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.
- 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.

5.5 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 [78].

$$\frac{ds}{dt} = \mathbf{N}\mathbf{v}[s, p] \tag{5.1}$$

Using the newer formalism and notation as given by Hofmeyr in [82] the kinetic model, shown in Eqn. 5.1, 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. 5.1, consider a metabolic network as earlier described in Fig. 3.1. In this system external (or boundary metabolites) are indicated as X_0 ... X_7, internal (variable) metabolites are shown as S_1 ... S_3 and the enzyme catalysed reactions as E_1 ... E_4. The system in Fig. 3.1 can be rewritten as a stoichiometric matrix ($\mathbf{N}$, whose construction is described in more detail

in the next section):

$$\mathbf{N} = \begin{array}{cccc} \mathrm{E}_1 & \mathrm{E}_2 & \mathrm{E}_3 & \mathrm{E}_4 \\ \hline 1 & 0 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{array} \begin{array}{l} \\ \mathrm{S}_1 \\ \mathrm{S}_2 \\ \mathrm{S}_3 \end{array}$$

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. 5.1 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:

$$\begin{aligned} \frac{d\mathrm{S}_1}{dt} &= v_1 - v_3 - v_4 \\ \frac{d\mathrm{S}_2}{dt} &= v_2 - v_1 \\ \frac{d\mathrm{S}_3}{dt} &= v_1 - v_2 \end{aligned}$$

5.5.1 The stoichiometric matrix

In this section we investigate the first component of the kinetic model as shown in Eqn. 5.1, 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:

$$N = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & & \vdots \\ c_{m1} & \cdots & c_{mn} \end{bmatrix} \tag{5.2}$$

where as described in [82]:

- $c_{ij} < 0$ 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.

5.6 Stoichiometric analysis

There are two structural properties that are contained in the stoichiometric matrix ($\mathbf{N}$), both of which deal with the relationship between either the differential equations describing the metabolites or fluxes at steady state [78, 82]. 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 [128]. Let us begin by investigating how we can determine the moiety conserved relationships of a system from $\mathbf{N}$.

5.6.1 Moiety conservation – calculating L

The detection of moiety conservation [41] is critical in many aspects of metabolic analysis and modelling, for example,

- in the computation of the steady state [70],
- calculation of control coefficients using the control matrix equation [76, 82, 129]
- and the understanding and reduction of complex reaction networks [130].

Central to the idea of moiety conservation is the *Link* or **L** matrix [131]. If conservation exists the **L** matrix can be partitioned into an identity and zero link matrix, $\mathbf{L_o}$:

$$\mathbf{L} = \begin{bmatrix} \mathbf{I} \\ \mathbf{L_o} \end{bmatrix} \tag{5.3}$$

There are various ways of finding the moiety conservations present in **N** including Gaussian elimination, Gauss-Jordan elimination and Singular Value Decomposition [132] (see [133] for a comparison of various computational techniques). PySCeS uses Gauss-Jordan elimination [128] to determine the conserved relationships between the differential equations. This is a variation of the technique that uses an augmented **N** matrix and Gaussian elimination to determine the conservation matrix [134] (for an example see [130]).

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 **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 $\mathbf{L_o}$ directly from $\mathbf{N}^T$ using the following procedure [133]:

- 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 $\mathbf{L_o}$ from the LU factorization result,
- build $\mathbf{L}$ from $\mathbf{L_o}$,
- using $\mathbf{L}$ and $\mathbf{L_o}$, compute the conservation matrix and form the reduced stoichiometric matrix.

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:

$$\mathbf{N}^T = \mathbf{PLU} \tag{5.4}$$

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 [135]. 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.

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 [136, 137], 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

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

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.

$$echelon(\mathbf{N}^T) = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ & \cdot & \cdot & \cdot & \cdot \\ & & \cdot & \cdot & \cdot \end{bmatrix} \tag{5.5}$$

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 $\mathbf{N}$:

$$reduced(\mathbf{N}^T) = \begin{bmatrix} 1 & & & \cdot & \cdot \\ & 1 & & \cdot & \cdot \\ & & 1 & \cdot & \cdot \end{bmatrix} \tag{5.6}$$

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

As shown in [133] the reduced form matrix shown in Eqn. 5.6 can generally be written as

$$reduced(\mathbf{N}^T) = \begin{bmatrix} \mathbf{I} & \mathbf{F} \\ 0 & 0 \end{bmatrix} \tag{5.7}$$

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. 5.3) as well as construct the conservation matrix γ which is also defined in [133].

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

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 $\mathbf{L}$ we can tell which of $\mathbf{N}$'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.

5.6.2 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 [82] and are expressed as the *Kernel* matrix, $\mathbf{K}$, where:

$$\mathbf{K} = \begin{bmatrix} \mathbf{I} \\ \mathbf{K_0} \end{bmatrix} \tag{5.8}$$

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. 5.1 as shown in [82]:

$$\mathbf{N_R v}[s_i, s_d, p] = 0 \tag{5.9}$$

or for the situation where there is no conservation

$$\mathbf{N v}[s, p] = 0 \tag{5.10}$$

The theory of linear algebra (see [128]) shows us that for any set of equations that have the form:

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

the nullspace of $\mathbf{A}$ (if it exists) shows us the linear dependencies amongst the columns of $\mathbf{A}$ (in our case the fluxes). Fortunately, calculating the null space of $\mathbf{N}$ is almost identical to the procedure used to calculate $\mathbf{L_0}$ (as described in Section. 5.6.1).

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 $\mathbf{N}$ and $\mathbf{K_0}$ is extracted. After the reduction of $\mathbf{N}$ to its row reduced form:

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

The basis for the nullspace ($\mathbf{F}$) 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 $\mathbf{K}$ matrix (Eqn. 5.8) 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.

5.6.3 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.

N and $\mathbf{N_R}$

The following attributes represent $\mathbf{N}$ 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 $\mathbf{N}$ to screen including row and column labels
- `mod.nmatrix` – a SciPy array containing the stoichiometric matrix, $\mathbf{N}$.
- `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 $\mathbf{N_R}$.

L, L_0 and conservation matrices

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

- `mod.showL(File=None)` print $\mathbf{L}$ to screen with rows and columns, if no conservation is present a warning message will be printed instead.
- `mod.lmatrix` – link matrix, $\mathbf{L}$. If there is no moiety conservation in the system this is an identity matrix.
- `mod.lmatrix_row` – the $\mathbf{L}$ row vector contains metabolites partitioned into independent and dependent.
- `mod.lmatrix_col` – the $\mathbf{L}$ column vector contains the independent metabolites.
- `mod.lzeromatrix` – the link zero matrix, $\mathbf{L_0}$. This matrix is an identity matrix if there is no moiety conservation present in the model.
- `mod.lzeromatrix_row` – the $\mathbf{L_0}$ rows contain the dependent metabolites as linear combinations of the independent ones.
- `mod.lzeromatrix_col` – the $\mathbf{L_0}$ columns contain the independent metabolites.

As we have seen, once the $\mathbf{L}$ 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, $\mathbf{T}$ 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.

K and $\mathbf{K_0}$

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

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

5.7 The Tao of PySCeS: Part 2

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

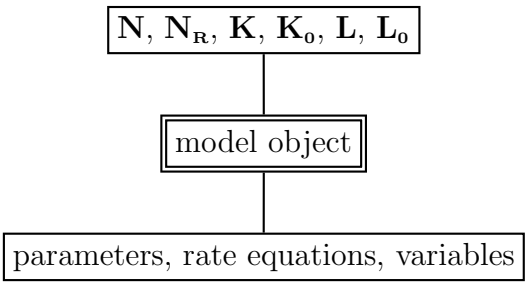


Fig. 5.2 *A PySCeS model object.* When a model object is initially created using the `doLoad()` method it has attributes representing both the basic and structural properties of the model

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. 5.2). 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

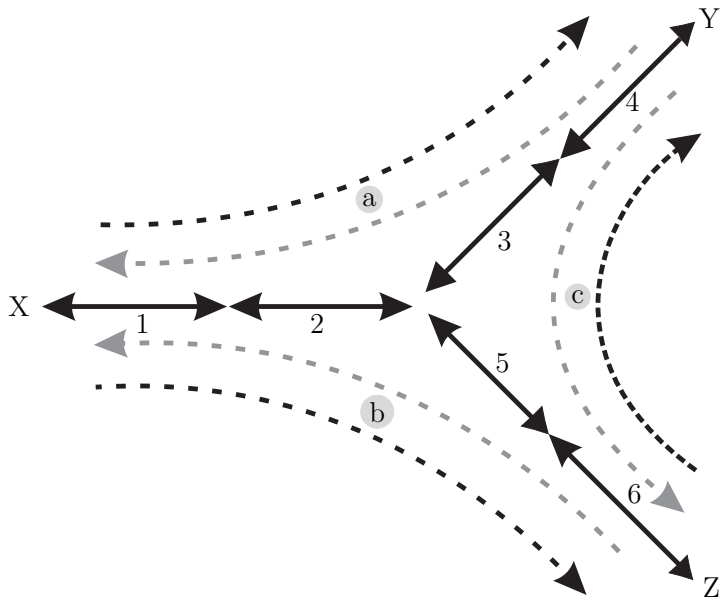


Fig. 5.3 *Elementary flux modes.* The solid lines represent six reversible enzyme catalysed reactions that form a branched metabolic pathway. In this system there are 6 elementary modes shown as three pairs a–c of black and grey dashed lines. This system is based on a similar one described in [140].

that follow we will describe various analyses which can be performed on the basic model object.

5.8 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 [138, 139]. In Fig. 5.3 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. 5.3, 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 [140]. 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 de-

composed 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 [138, 141–143]. An analogous procedure has also been proposed for the determination of 'metabolically meaningful pools' [144].

5.8.1 Using MetaTool with PySCeS

In order to calculate the elementary flux modes PySCeS incorporates an interface to a stand-alone application: MetaTool [125]. 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.

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.

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

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

5.9 Evaluating the differential equations

Analysing the kinetic model whether dynamically or at steady steady state involves solving the differential equations as expressed in Eqn. 5.9. 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. 5.9

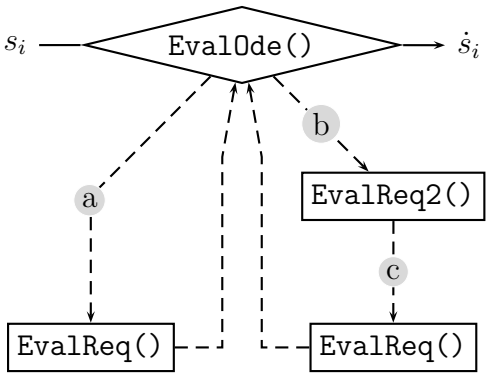


Fig. 5.4 *Evaluating the differential equations.* If there is no moiety conservation `EvalOde()` uses only `EvalReq()`, (a) while when there is moiety conservation `EvalReq2()`, (b) and `EvalReq()`, (c) are used to determine $\dot{s}_i$ from a concentrations vector s_i

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. 5.9 namely $\mathbf{Nv}$ can be used directly as $\mathbf{s} = \mathbf{s}_i$, as is shown in Fig. 5.4 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. 5.4 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()
```

5.10 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 [145, 146]. 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)

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

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

- `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.

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

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:

```
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.

5.11 Solving for the steady state

Conceptually, calculation of the steady state is easy: for any kinetic model as shown in Eqn. 5.1 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. 5.9. 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.

5.11.1 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. 5.5 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. 5.5 a to FINTSLV, Fig. 5.5 c.

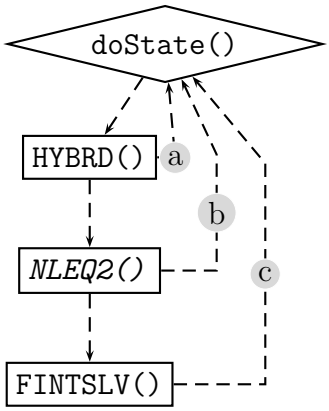


Fig. 5.5 *Steady-state solvers in a fallback configuration.* Three different algorithms are available to solve for a steady state where the dashed lines indicate optional routes to a solution. a HYBRD, b NLEQ2 and c FINTSLV. As NLEQ2 is an external solver it can be disabled and is therefore shown in 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.

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 [147] .

HYBRD uses a modified Hybrid Powell method [148] and generally is able to converge quickly even when the initial estimation of the solution is far from the actual solution [149]. 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.

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

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

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 10.1 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 [21]. Once an F2PY wrapper (included in Appendix 10.7) 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 5.12.

NLEQ2 is one of a family of algorithms including NLEQ1 and NLEQ1S which are based on Deuflhard's affine invariant damped Newton techniques [150, 151]. In addition to the damping strategies employed by NLEQ1, NLEQ2 incorporates algorithms which serve to extend the convergence domain of the algorithm [152]. 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).

[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_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).

- `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.

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.

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

- `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.

5.11.2 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.

5.12 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. 5.6 (the PySCeS code used to generate this figure is given in Appendix 10.8). This type of behaviour will be looked at in more detail in Chapter 6.

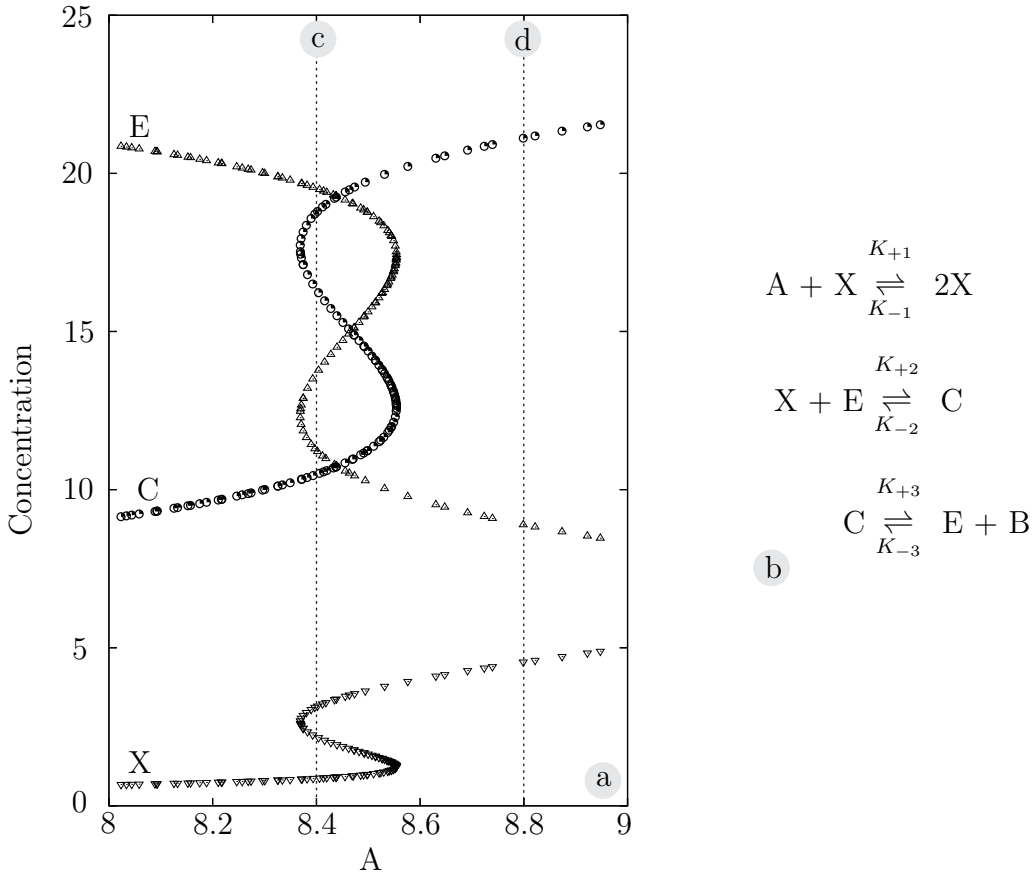


Fig. 5.6 *Parameter scan generated using PySCeS continuation.* The Edelstein model (whose reaction steps are shown in b) [153] exhibits multiple steady-state solutions (represented by the steady-state metabolite concentrations **X**, **E** and **C**) at different values of the parameter, **A**. In a 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.

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 [120, 131]). Two widely used techniques for investigating systems that exhibit multiple solution are homotopy and continuation methods [120]. 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) [154–156]. 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 website[18].

5.12.1 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 [21]. 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.

PITCON proved to be more of a challenge, as its Fortran function definition had a

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

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 10.9) 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.

5.12.2 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

[20] `http://cens.ioc.ee/pipermail/f2py-users/2003-December/000611.html`

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

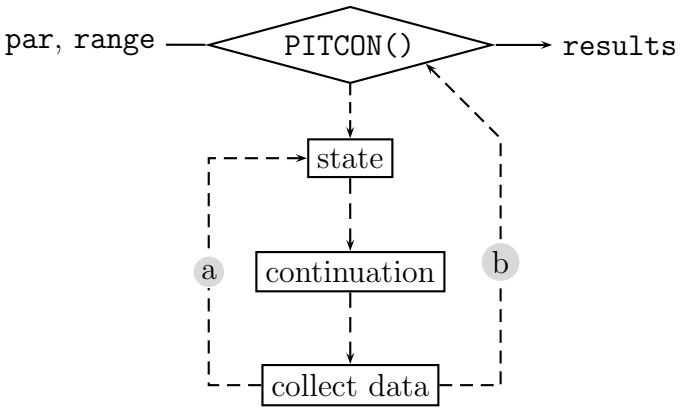


Fig. 5.7 *Continuation algorithm using PITCON.* For every point in the supplied `range` a steady state is calculated and used to initiate a continuation for the parameter (`par`), a. Once completed the complete data set is returned to the calling method, b and output as the continuation `results`.

present a problem as we can provide the continuation algorithm an exact solution by 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

[21] Available online at `http://www.netlib.org/contin/dpcon61.f`

was extended by appending a parameter (λ) thus forming an extended concentration vector, $\mathbf{s}_{\mathbf{I},\lambda}$

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

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 $\mathbf{s}_\mathbf{I}$ and dynamically assign λ a value before evaluating the rate equations using $\mathbf{s}$ which could be generated from $\mathbf{s}_\mathbf{I}$. 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. 5.7) inside a parameter scan defined by the user (shown in Fig. 5.7 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. 5.6). 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. 5.7 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. 5.6 was generated in this way.

5.12.3 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).

[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_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 $\mathbf{s}_{\mathbf{I},\lambda}$ 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 $\mathbf{s}_{\mathbf{I},\lambda}$ 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.

- `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 6 contains worked out examples which further demonstrate how `mod.PITCON()` can be used to study the stability and control of such a system.

5.13 Metabolic Control Analysis

So far in this chapter 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 [60, 61]. 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 ($\mathbf{s}$), external metabolites ($\mathbf{x}$) and parameters ($\mathbf{p}$) are constant quantities [82].

$$\varepsilon_{s_j}^{v_\mathrm{i}} = \left(\frac{\partial v_\mathrm{i}/v_\mathrm{i}}{\partial s_j/s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} = \left(\frac{s_j}{v_\mathrm{i}}\right)\left(\frac{\partial v_\mathrm{i}}{\partial s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} = \left(\frac{\partial \ln v_\mathrm{i}}{\partial \ln s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} \tag{5.11}$$

Eqn. 5.11 shows three equivalent versions of the scaled (dimensionless) elasticity ($\varepsilon_{s_j}^{v_\mathrm{i}}$). The corresponding unscaled form of this elasticity ($\tilde{\varepsilon}_{s_j}^{v_\mathrm{i}}$) would be:

$$\tilde{\varepsilon}_{s_j}^{v_\mathrm{i}} = \left(\frac{\partial v_\mathrm{i}}{\partial s_j}\right)_{[\mathbf{s},\mathbf{x},\mathbf{p}]} \tag{5.12}$$

The definitions of the elasticity coefficient as shown in Eqn. 5.11 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 5.13.1, 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 ($\mathbf{z}$) of a change in a system parameter ($\mathbf{p}$).

$$R_p^z = \frac{\partial z/z}{\partial p/p} = \frac{\partial z}{\partial p} \cdot \frac{p}{z} = \frac{\partial \ln z}{\partial \ln p} \tag{5.13}$$

Using the partitioned response property:

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

a steady-state flux ($\mathbf{J}$) 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] [76, 82, 129].

$$\begin{bmatrix} \tilde{\mathbf{C}}^J \\ \tilde{\mathbf{C}}^s \end{bmatrix} [\mathbf{K} \;\; -\tilde{\boldsymbol{\varepsilon}}_\mathbf{s}\mathbf{L}] = \begin{bmatrix} \mathbf{K} & 0 \\ 0 & \mathbf{L} \end{bmatrix} \tag{5.14}$$

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 [45, 129]:

$$\tilde{\mathbf{C}}^i = \tilde{\mathbf{E}}^{-1} \tag{5.15}$$

In Section 5.13.2 it will be seen how PySCeS uses the relationship shown in Eqn. 5.15 to calculate the matrix of independent control coefficient by inverting the elasticity matrix.

[23] Various alternatives to this equation have been proposed, see for example [78, 80, 157, 158])

5.13.1 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. 5.8 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, $\tilde{\varepsilon}_{\mathbf{v}}$, as well as the system parameters, $\tilde{\varepsilon}_{\mathbf{p}}$. The actual code used to calculate the derivatives is generated and compiled

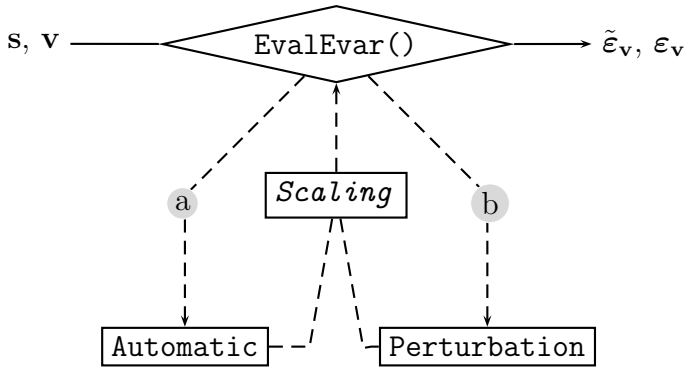


Fig. 5.8 *Differentiating the rate equations* Given a concentration (**s**) and rate vector (**v**) `EvalEvar()` generates the elasticities with respect to the variable metabolites ($\tilde{\varepsilon}_{\mathbf{v}}$) by either automatic, a, or numeric differentiation, b. Once generated $\tilde{\varepsilon}_{\mathbf{v}}$ can optionally be scaled to $\varepsilon_{\mathbf{v}}$.

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 differenti-

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

ated automatically. In order to cater for this situation PySCeS includes methods for the numerical approximation of the elasticities (as shown in Fig. 5.8 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 `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.

[25] `http://www.scipy.org`

[26] This solution was inspired by Herbert Sauro's Jarnac [92]

- `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.

5.13.2 Calculating the control coefficients

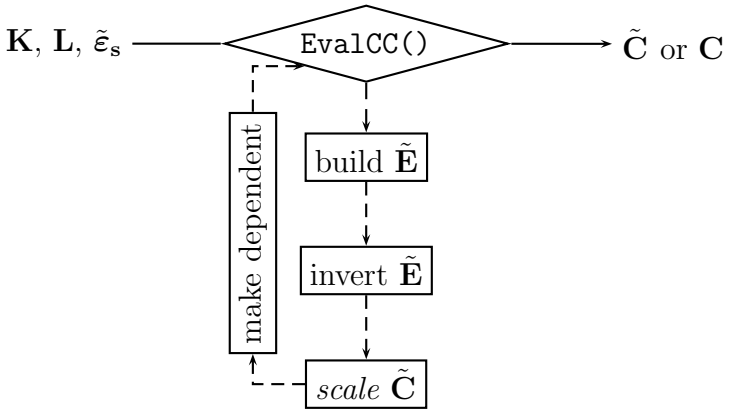


Fig. 5.9 *Calculating the control coefficients.* Using the **K**, **L** and elasticity matrix $\tilde{\varepsilon}$, `EvalCC()` constructs $\tilde{\mathbf{E}}$ using $[\mathbf{K} \; -\tilde{\varepsilon}_{\mathbf{s}}\mathbf{L}]$. The control-matrix equation, $\tilde{\mathbf{E}}$ is inverted to form $\tilde{\mathbf{C}}^i$, a matrix of independent control coefficients. This matrix is then optionally scaled and used to regenerate the dependent coefficients.

Once a system's elasticities have been obtained the control coefficients can be calculated by using the control-matrix equation. Fig. 5.9 gives an overview of the method which is used to implement Eqn. 5.15 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 $\boldsymbol{\varepsilon}$ to form (using Eqn. 5.14) 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

[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`.

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 $\mathbf{K}$ and $\mathbf{L}$, to form a scaled $\mathbf{E}$. By inverting a scaled $\mathbf{E}$ 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 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 $\mathbf{E}$, 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 $\mathbf{E}$, 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 [82]:

$$\begin{aligned} \mathbf{C^{s_d}} &= \mathcal{L}_0 \mathbf{C^{s_i}} \\ \mathbf{C^{J_d}} &= \mathcal{K}_0 \mathbf{C^{J_i}} \end{aligned} \tag{5.16}$$

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.

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

- `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. 5.9, 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.
- `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 chapter, 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.

5.14 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 [82] it is possible to formulate the Jacobian matrix ($\mathbf{M}$) for the kinetic model as:

$$\mathbf{M} = \mathbf{N_R}\tilde{\varepsilon}_\mathbf{s}\mathbf{L} \tag{5.17}$$

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 10.10 which contains an extract from the GEEV source code[29].

- `mod.doEigen()`: a high level method that calculates a steady state, elasticities and eigenvalues.

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

- `mod.EvalEigen()`: the eigenvalue analysis method assumes the the elasticity matrix, $\tilde{\varepsilon}_{\mathbf{s}}$, exists.
- `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 [159]:

- 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
```

5.15 The Tao of PySCeS: part 3

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

So far in this chapter 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.

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

5.16 Single dimension parameter scan

Earlier in this chapter 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 [12]. A slightly more esoteric example of a parameter scan has already been seen in Fig. 5.6 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.
- `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 10.11.

5.17 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.

5.17.1 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
```

5.17.2 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.

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
```

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

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. 5.10. 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
- `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'

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. 5.10 *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 [160, 161].

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.

5.18 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.

5.19 PySCeS module functions

So far this chapter 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.*`.

5.19.1 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`.

Drawing graphs

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

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

- `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.

```
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')
```

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.

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='')`

5.19.2 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.

5.19.3 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 10.12.

5.19.4 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.

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 (but 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'
```

5.20 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 [123]

In Chapter 4 three principles were used as the basis for the design of a new modelling application. Hopefully, this chapter has shown how these principles have been implemented in the form of a new modelling application, PySCeS.

One argument that is often used against script-based modelling software is that 'there is so much typing' or 'there are so many commands to remember'. In a sense this may be true, but it ultimately depends on your modelling purpose. If ease of use is important then the JWS Online software, discussed in Chapter 7, or a graphical package such as Gepasi should rather be used. If however, flexibility of use is required then a scripted application such as PySCeS or Jarnac is a logical solution. Many of today's successful scientific software applications (e.g. Autocad, Matlab and Mathematica) have shown how powerful the integration of a scripted interface with a GUI menu/notebook system can be. In a similar way PySCeS has been designed so that the scripted interface forms a base which can be built on to fulfil a variety of potential modelling roles.

PySCeS is currently the only freely available, open source, interactive computational systems biology modelling package that is capable of doing structural, kinetic, continuation and control analysis, using exactly the same interface, on both Linux and Windows

systems. Of course in principle nothing should prevent it being ported with minor configuration changes to run on any operating system that supports SciPy[32]. PySCeS has been designed to be flexible and user extendable, I like the idea of a todo list with, at least, one perpetual item:

- anything that you can need or can imagine ...
-

On the other hand, script-based (or command line interface) modelling software is often associated with sentiments such as, 'there is so much typing' or 'there are to many commands to remember'. In a sense this may be true, but it ultimately depends on your modelling purpose. If ease of use is important then the JWS Online software, discussed in Chapter 7, or a graphical package such as Gepasi should rather be used. If however, flexibility of use is required then an interactive application such as PySCeS or Jarnac is a more attractive solution. Many of today's successful scientific software applications (e.g. Autocad, Matlab and Mathematica) have shown how powerful the integration of a scripted interface with a GUI menu/notebook system can be. In a similar way PySCeS has been designed so that the scripted interface forms a base which can be built on to fulfil a variety of potential modelling roles. For example, PySCeS has been successfully used for postgraduate teaching and research purposes and is being further developed in order to act as a base for a comprehensive kinetic modelling and computation systems biology resource to be known as Edu*PySCeS*.

It is interesting to note that soon after releasing PySCeS and presenting it at the 11th Workshop of the BioThermoKinetics Study Group[33] the first message on the PySCeS help forum[34] was posted in connection with extending the scanning function and automating (i.e. scripting) a model's analysis.
Perhaps it is fitting that Eric Raymond be allowed the final word on the matter:

> "The next best thing to having good ideas is to recognize good ideas from your users. Sometimes the latter is better." – Eric Raymond [123]

[32]This includes Apple Macintosh, FreeBSD and others
[33]`http://btk2004.brookes.ac.uk/`
[34]`http://sourceforge.net/forum/forum.php?thread_id=1143986&forum_id=43965`

6 Multiple hysteresis in a linear metabolic pathway with end-product inhibition

6.1 Introduction

6.1.1 Background

In a previous study [11, 13] we used a process of forward engineering (which we called metabolic design) to develop a deeper understanding of the behaviour, control and regulation of a typical metabolic 4-way junction that consisted of (i) biosynthesis of an end-product that serves as a macromolecular building block, (ii) external supply of this end-product, (iii) its catabolic breakdown, and (iv) macromolecular synthesis that uses the end-product. We made extensive use of supply-demand analysis with log-log rate characteristics [12] to design a system that has certain predefined regulatory properties, in particular, effective homeostasis of the steady-state concentration of the end-product in the face of flux control by demand. Fig. 6.1 shows a typical log-log supply-demand rate characteristic. The steady-state flux, J, and steady-state concentration $\bar{p}$ can be read from the graphs at the point where the two curves intersect. In addition, the slopes of the curves are equal to the supply and demand block elasticities with respect to P. An important aim of this study was to quantify the contributions of the kinetic properties of the individual enzymes to the overall regulatory behaviour of the system.

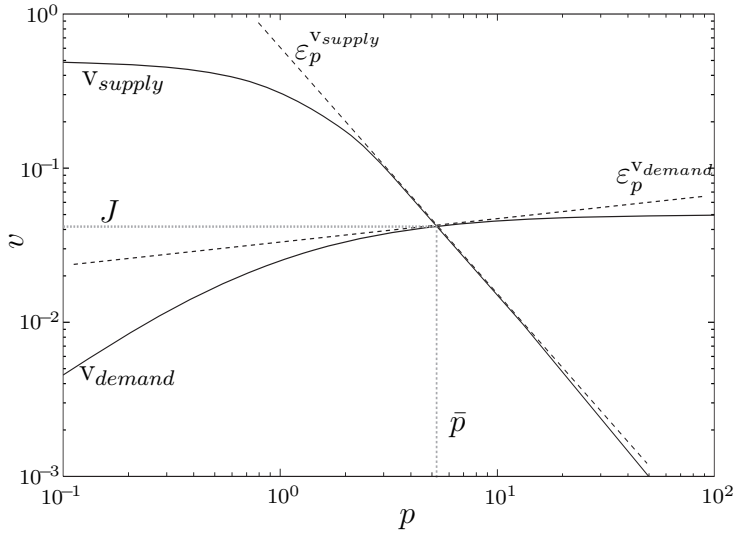


Fig. 6.1 *The rate characteristics of a supply-demand system plotted in log-log space.* Steady state obtains at the intersection point. At any p the slopes of the tangents of the rate curves are the elasticity coefficients of supply and demand for a particular value of p. Shown on the graph are the elasticities of supply and demand with respect to the linking metabolite P at steady state.

The core of our system is the biosynthetic block, which we modelled as a 4-enzyme linear sequence of reactions, with and without end-product inhibition of the committing enzyme, i.e., the first enzyme in the sequence (this pathway is known affectionately as the Stellenbosch organism). This system (shown in Fig. 6.2) is typical of a biosynthetic pathway leading to, for example, and amino acid or a nucleotide, and has been extensively studied in a wide range of contexts [16, 42, 162–164]. Of particular interest was our use of the reversible Hill rate equation, developed by Hofmeyr and Cornish-Bowden [17], to model cooperative binding and allosteric inhibition of the committing enzyme (shown as E_1 in Fig. 6.2). We have argued [165] that this rate equation provides a better description of regulatory properties than the often-used Monod-Wyman-Changeux rate equation.

For the purposes of the investigation described here, the system described above is simplified to a biosynthetic supply block (E_1, E_2, E_3 in Fig. 6.2) linked by P to a single demand block. Once defined in this way we can use log-log rate characteristics [12]

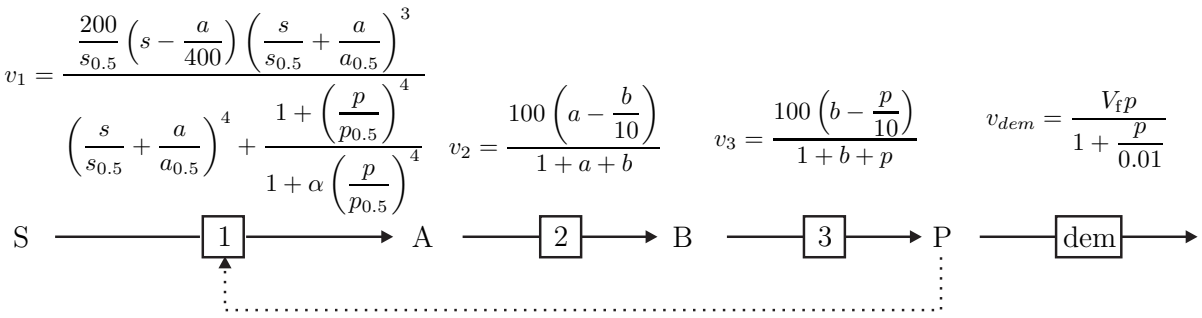


Fig. 6.2 *The Stellenbosch organism.* A linear 4 enzyme pathway that contains a supply block of three enzymes and a demand block consisting of a single reaction. E_1 is described by the reversible Hill equation [17] where S, $s_{0.5}$ and $p_{0.5}$ are given a value of 1.0 concentration units and the inhibition factor α is set to 0.01. Reactions E_2 and E_3 are modelled with reversible Michaelis-Menten kinetics, $v = \frac{V_f}{K_s}\left(s - \frac{p}{K_{eq}}\right) \Big/ \left(1 + \frac{s}{K_s} + \frac{p}{K_p}\right)$ where for both reactions K_s and K_p are set to 1.0 concentration unit. This system is connected to a demand reaction (*dem*) which is modelled with an irreversible Michaelis-Menten rate equation. Adapted from [18].

towards P to simultaneously study the response in the steady-state fluxes local to both of the supply and demand blocks to changes in the concentration of P. In other words, for the purposes of modelling this system, P is regarded as a fixed parameter. At any fixed concentration of the pathway substrate S the maximum value of P is determined by the overall equilibrium constant of the biosynthetic blocks, which can be calculated as the product of the equilibrium constants of the individual reactions:

$$K_{tot} = K_{e_1} \times K_{e_2} \times K_{e_3}$$

For an S-concentration of 1 K_{tot} in our system would be 40000 ($400 \times 10 \times 10$). These are the values used throughout.

6.1.2 Unexpected behaviour: hysteresis

As part of the study described above we were interested in the effect of changing the strength with which the committing enzyme bound its immediate product, A. This was accomplished by varying the half-saturation constant $a_{0.5}$ of E_1. With increasing strength we started to see with Gepasi what at the time seemed like strange discontinuities in the supply rate characteristic (Fig. 6.4), but what turned out to be an artifact of Gepasi inability to cope with hysteretic behaviour. Using the rather limited continuation abilities of Scamp we were then able to show that the system actually was capable of

multi-stationary behaviour due to hysteresis. This surprised us because such behaviour is typically associated with kinetic phenomena such as positive allosteric feedback [166–168]. We were, however, unable to study this interesting phenomenon in depth, due to the fact that none of the software packages available could simulate this behaviour fully. We did manage to cobble together a picture of this behaviour of a range of $a_{0.5}$-values but, frustratingly, could go no further; to generate even a single two-dimensional rate characteristic took an inordinate amount of time. At the time we described these problems as follows:

> "At this stage we felt like Plato's cave-dwellers that catch only glimpses of reality by watching shadows on the rock face. However, to climb out the cave and view the full reality turned out to be more complicated than expected. To get a clearer picture of the transition from low to high affinity of E_1 for A we wanted to construct a 3-dimensional plot by adding an axis for $a_{0.5}$. We could not produce the regularly gridded data needed for generating the 3-D plot with one simulation program: Gepasi can produce gridded data, but only for the stable parts of the curves; Scamp in continuation mode on the other hand can produce the full picture but, because p is now a variable, its values are not gridded. At any specific $a_{0.5}$ we could, however, generate a curve on a planar cross-section of the 3-D space from separately simulated sections of that curve. Roughly 20 megabytes and 48 data files later, we finally climbed out of our cave and saw the weird landscape in the bright light of day. And what a picture it was (Fig. 6.3)." – [11]

As described in the introduction, it was to a large degree the inability to continue investigating this phenomenon that led us to begin developing our own modelling software package, which became PySCeS. The rest of this chapter describes and discusses the results that we were finally able to obtain with PySCeS and the powerful continuation algorithm PITCON (see Section 5.12 for details of its implementation in PySCeS). Appendix 11 contains both the models and example scripts that were used to generate most of the results. It also contains URL's where these scripts can be downloaded and used to recreate the results presented in the remainder of this chapter.

6.2 Results

6.2.1 The effect of changing $a_{0.5}$

Using PySCeS we now pick up the strands left dangling in the previous study by looking at the effect on the supply block of increasing the strength with which E_1 binds its immediate product A (see Fig. 6.2 for details of the enzyme kinetics)

Fig. 6.4 a shows a supply block rate characteristic where discontinuities are highlighted with a grey block. Instead of the monotonic competitive product-inhibition response curves that one would expect when E_1 strongly binds its product (i.e. $a_{0.5} < 1$; front part of graph) [18], we find instead abrupt jumps to and from what appears to be another stable branch of steady-state solutions. Discontinuities such as these are generally associated with some sort of hysteretic behaviour, so it is distinctly possible that there are regions where the system has multiple steady-state solutions. In our previous study this is where we would have to begin the time consuming process of using different applications to try and piece together the data needed to characterize these regions. Fortunately, this can now easily be done using the PySCeS `PITCON` interface.

As can be seen in Fig. 6.4 b , the PITCON algorithm can solve for multiple steady-state solutions irrespective of whether that solution is stable or or not. This is shown in Fig. 6.5 where it can be seen that in the region of the hysteresis curves (Fig. 6.5 b) there are three possible steady states for each value of P. So far we have been investigating a single two dimensional rate characteristic where the $a_{0.5}$ has been fixed at a value of 3. The next logical question is, how does this system react for different values of $a_{0.5}$? To answer this question we need to be able to draw the 3D rate characteristic that was described earlier in this chapter. Fortunately, the scriptable nature of PySCeS allows us to easily create a PySCeS program that runs a series of PITCON scans and collects the data in a format suitable for plotting. Fig. 6.6 shows how by simply changing the binding strength of E_1 for its direct product A ($a_{0.5}$), the supply rate characteristic can be transformed from a typical sigmoidal allosteric response curve (weak binding) to ones containing dual hysteresis and isola's (strong binding) to a typical competitive binding curve (very strong binding). Using PySCeS, we are now able to quickly generate two and three dimensional rate characteristics of the supply block even when they contain

unstable steady-state solutions.

The question arises of the nature of the mechanism that causes the hysteric behaviour. Clearly the enzyme that is responsible for this must be E_1, as the simple reversible Michaelis-Menten kinetics of E_2 and E_3 cannot cause this behaviour. There must be some form of positive feedback involved, and the only candidate is A, the immediate product of E_1. Inspection of the E_1 equation shows that the place where A could have a positive effect on the rate is in the numerator term $\left(\frac{s}{s_{0.5}} + \frac{a}{a_{0.5}}\right)^3$. A also occurs in the thermodynamic numerator term $\left(s - \frac{a}{K_{\text{eq}}}\right)$ and the denominator, but here it could only have an inhibitory effect. What we are therefore interested in, is the behaviour of the elasticity profile of v_1 with respect to a, particularly whether it is positive in the hysteretic regions. This is investigated in the next section.

6.2.2 Elasticity profile of a multi-stable system

The continuation capabilities of PySCeS allow us to investigate both stable and unstable steady states.

Fig. 6.7 shows the elasticities ε_s^1, ε_p^1 and ε_a^1 plotted against P of a system where $a_{0.5}$ = 3. Unlike the log-log rate characteristics previously presented in this chapter this is a semi-log plot, because elasticities can have negative values. That the elasticity curves are continuous through the hysteresis regions (even though they exhibit some rather 'spectacular' looping patterns), emphasizes the fact that they are local properties of the enzymes and exist independent of the stability of a particular system state.

Of interest is that at in both hysteresis regions ε_a^1 is positive, whereas outside these regions it is negative. This indicates that in the hysteresis regions E_1 is being activated by its direct product, A, whereas normally it would be inhibited by A.

6.2.3 Tracking stability using eigenvalues

Although anomalous behaviour of elasticities discussed in the previous section provides some clues about possible regions of instability of the supply, only a study of the eigenvalues of the Jacobian matrix of the system can provide definitive answers. For example, if the real parts of the eigenvalues are all negative the steady state is stable; conversely,

if any eigenvalue has a positive real part, the steady state is unstable. Seydel [159] defines a test function that is useful in this regard. It simply suggests that one needs to trace the behaviour of the maximum real part of the eigenvalues against a change in a parameter. In our case we plot the maximum real part of the eigenvalue against the parameter p; in the regions where this value is negative the system is stable. When positive, it implies that one or more of the real parts of the eigenvalues are positive and the steady state is unstable.

Fig. 6.8 shows the result of plotting the maximum and minimum real parts of the eigenvalues for the supply block rate characteristic while Fig. 6.9 shows the original supply block rate characteristic where the unstable steady states have been highlighted. By following the maximum eigenvalue curve it is evident that in the regions where hysteresis occurs the values becomes positive and the system unstable. These correspond to the regions where product activation is shown to occur in Fig. 6.7.

6.2.4 The effect of changing $s_{0.5}$

Having identified $a_{0.5}$ as a key component of hysteresis formation in this system, we were interested in how substrate binding affects the supply block rate characteristic? Initially, we simply doubled the substrate half saturation binding constant ($s_{0.5}$) from its original value of one to a value of two and looked at how the 3D rate characteristic (Fig. 6.6) was affected. This meant that the binding strength of E_1 for S was decreased without the overall equilibrium of the pathway being affected. We found that a new discontinuity appeared at an $a_{0.5}$ value ≈ 4.

In Fig. 6.10 a – c we show a magnified section of the 3D rate characteristic at different values of $s_{0.5}$. In the case of a $s_{0.5}$ was at its default value of one and the 'normal' rate characteristics for strong product binding obtains. Note that due to the magnification, the P-axis now ends at a maximum of 10^3; we chose a linear $a_{0.5}$–axis because the effect was clearer in this view. In Fig. 6.10 c $s_{0.5} = 3$, and, as expected, the maximum supply at low concentrations of P is reduced.

However, in the case of Fig. 6.10 b, where $s_{0.5} = 2.0$, another hysteresis is formed which leads to an extended region at lower p where multiple steady-state solutions are possible. Having pinpointed this region of $s_{0.5}$ where the new hysteresis occurs, the next

step was to fix $a_{0.5} = 3.0$ and scan $s_{0.5}$ around the value of 2.0. The results are shown in Fig. 6.10 d. What we find is that altering the strength with which E_1 binds its substrate S produces a double hysteric region in the supply rate characteristic similar to the one produced by an alteration in $a_{0.5}$. This is to be expected as S and A not only compete for the binding sites on the enzyme, but they also enhance each other binding cooperatively (i.e., if, for example an S is bound to one binding site, it on the one hand blocks the binding of A to that site, but on the other hand makes it easier for an S or an A to bind to the next open site).

As with our investigation of the effects of changing $a_{0.5}$, we also investigated the elasticity profile and the eigenvalues of the supply system at values of $s_{0.5}$ and $a_{0.5}$ where the new hysteresis occurs. The elasticities are shown in Fig. 6.11 for the case when $s_{0.5} = 2.02$ and $a_{0.5} = 3$). In the region where the new hysteresis occurs (at very low P) ε_a^1 becomes positive and creates the positive feedback needed for the formation of a hysteresis. This suggests that the cause of all the hysteresis in this system is the activation caused by strong product binding; substrate binding has a modulatory effect on the product activation. Finally, in Fig. 6.12 the maximum real component of the eigenvalue is plotted, clearly showing the appearance of unstable steady states in the new hysteresis region.

In the following section we use supply-demand analysis to increase our understanding of the causes of the hysteretic behaviour of the supply rate characteristic.

6.2.5 Analysing hysteretic behaviour with supply-demand surfaces

In Fig. 6.6 we saw how, by changing the value of $a_{0.5}$, it was possible to define three general patterns in the 2D rate characteristics namely, a sigmoidal allosteric binding curve, a mushroom consisting of a double hysteresis, and an isolated branch or isola that lies below a competitive binding curve. One question which arises from this behaviour is whether these patterns are decomposable into interactions between the underlying system components.

One way of answering this question is to apply to the supply block the same methodology that was applied to the original supply-demand system, namely to break it into a smaller supply-demand system. As it is the binding of A to E_1 that generates the

hysteretic behaviour we create new supply and demand block around it. The PySCeS model is now modified by fixing A in addition to P (see Appendix 11.4). Using this new model we now simultaneously plotted the three dimensional supply-demand rate characteristics of the supply block produced when both A and P are scanned together, and compared it to the results of a parameter continuation of the original system where A is free. Additionally, we compared these at three different values of $a_{0.5}$ that are typical of the three distinctive curves discussed earlier.

Fig. 6.13 shows the results of this experiment. The graphs on the left hand side Fig. 6.13 a – c are the 3D rate characteristics of the mini supply-demand system and the resulting supply (E_1) and demand flux (E_2, E_3) are plotted. The right-hand graphs Fig. 6.13 d – f are 2D rate characteristics of the original supply block and represent three different cross-sections of Fig. 6.6.

At a first approximation, the intersection between the supply and demand surfaces in the left-hand graphs have the same form as the supply curve in the right hand graphs. Conceptually, if one imagines that by unfixing A in graphs Fig. 6.13 d – f we effectively remove the A parameter axis from the graph and 'squash' or project the intersection curves of graphs Fig. 6.13 a – c onto a 2D surface, which would then match graphs Fig. 6.13 d – f.

This type of nested supply-demand analysis clearly helps in interpreting the steady-state responses of metabolic systems and may prove useful in general.

6.3 Discussion

Using a combination of supply-demand rate characteristics, and a subsequent analyses of the elasticity and stability profiles of the supply block, we have been able to show that the primary trigger for hysteresis in this system is direct product activation of a cooperative enzyme; this is directly related to the strength with which E_1 binds its product, A, and can be modulated by changing the the strength with which E_1 binds its substrate, S. Further analyses, using three dimensional rate characteristics, of the product activation effect showed how the hysteretic behaviour is generated by the interaction of the underlying system components (Fig. 6.13).

Although we could show the power of simulation with PySCeS in the investigation described here, it may be that our understanding may be enhanced even further by using additional mathematical frameworks such as, for example, Catastrophe Theory [169, 170]. By inspection, the hysteresis folds seen in Fig. 6.6 could be described as two elementary cusp catastrophes coalescing into an isola and eventually terminating in a singularity. Examples of the elementary catastrophes have been proposed to occur in both chemical and biological systems [170, 171]. It would be interesting to see if some of the conclusions arrived at by catastrophe theory are applicable to the *Stellenbosch Organism*.

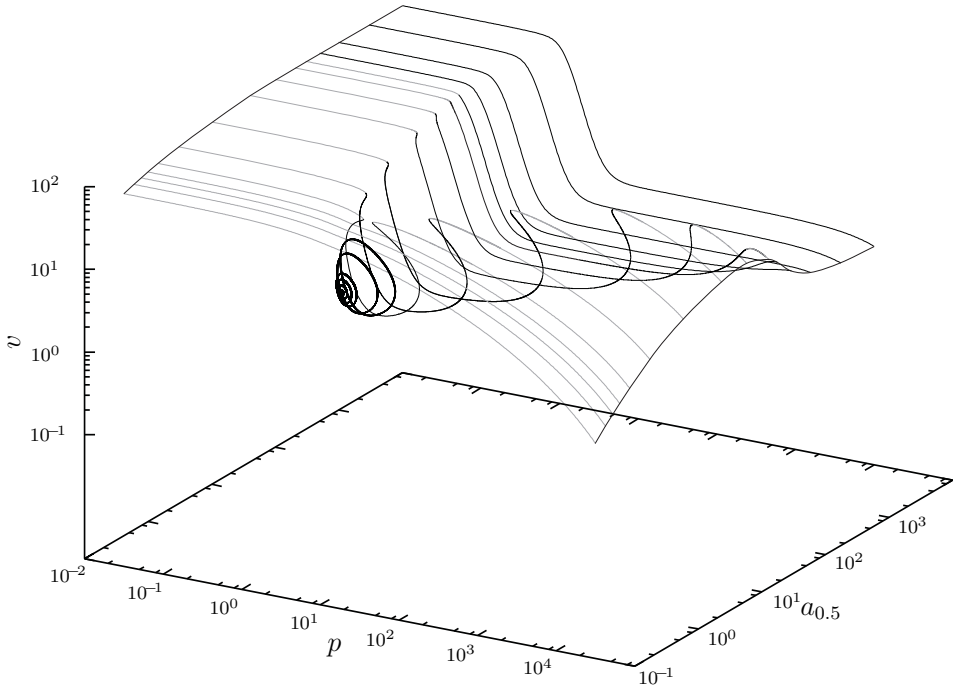


Fig. 6.3 *A 3-D picture of the mushroom transforming into a vanishing isola, with flux plotted against p at different values of $a_{0.5}$.* Similar graphs can be constructed for the concentrations of the intermediates A or B: the shapes of the mushrooms and the isolas are different, but on the whole the same picture is found. This figure is reproduced from [18].

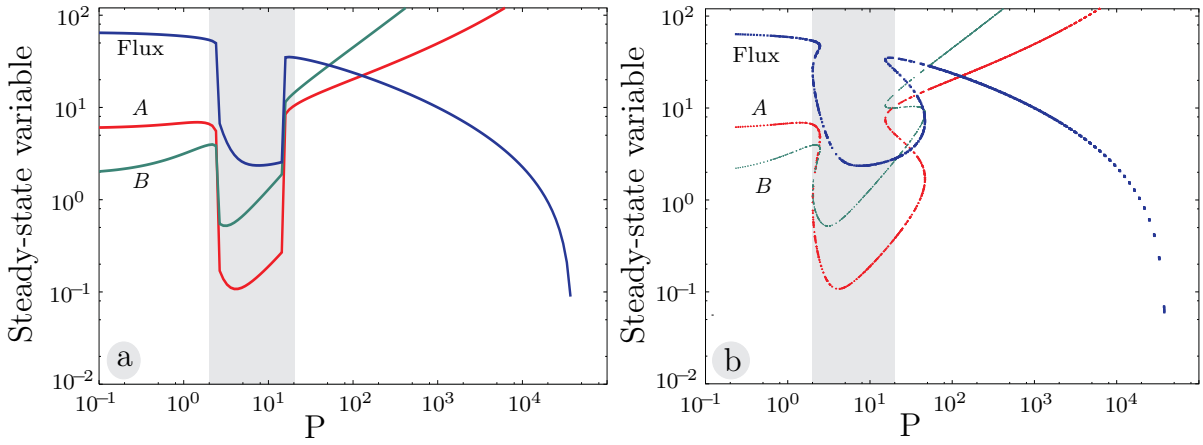


Fig. 6.4 *Parameter scans generated in two different ways.* Both these graphs show how the steady-state variables change with P when E_1 strongly binds its product A (achieved by setting $a_{0.5} = 3.0$). All other parameters are as described in Fig. 6.2. The graph in (a) is generated using the HYBRD and NLEQ2 non-linear solvers while (b) is produced using the PITCON continuation algorithm. In both graphs the grey shaded area highlights the seemingly discontinuous behaviour.

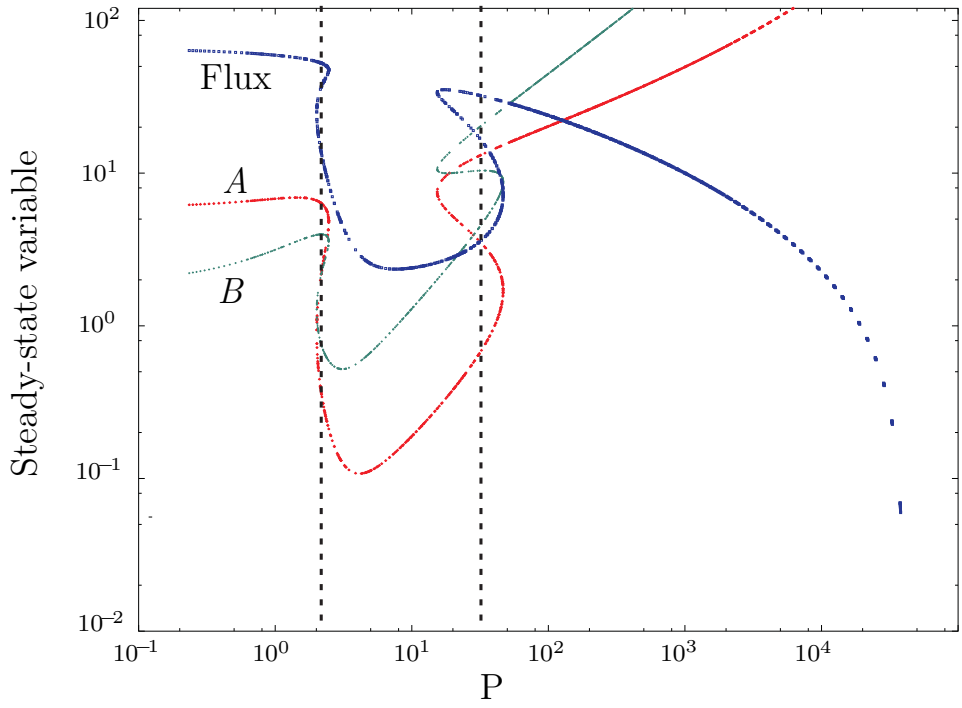


Fig. 6.5 *Multiple steady-state solutions are possible for a single value of* P. For each of the steady-state flux and concentration (A and B) curves there are regions where three steady-state solutions are possible (dotted lines).

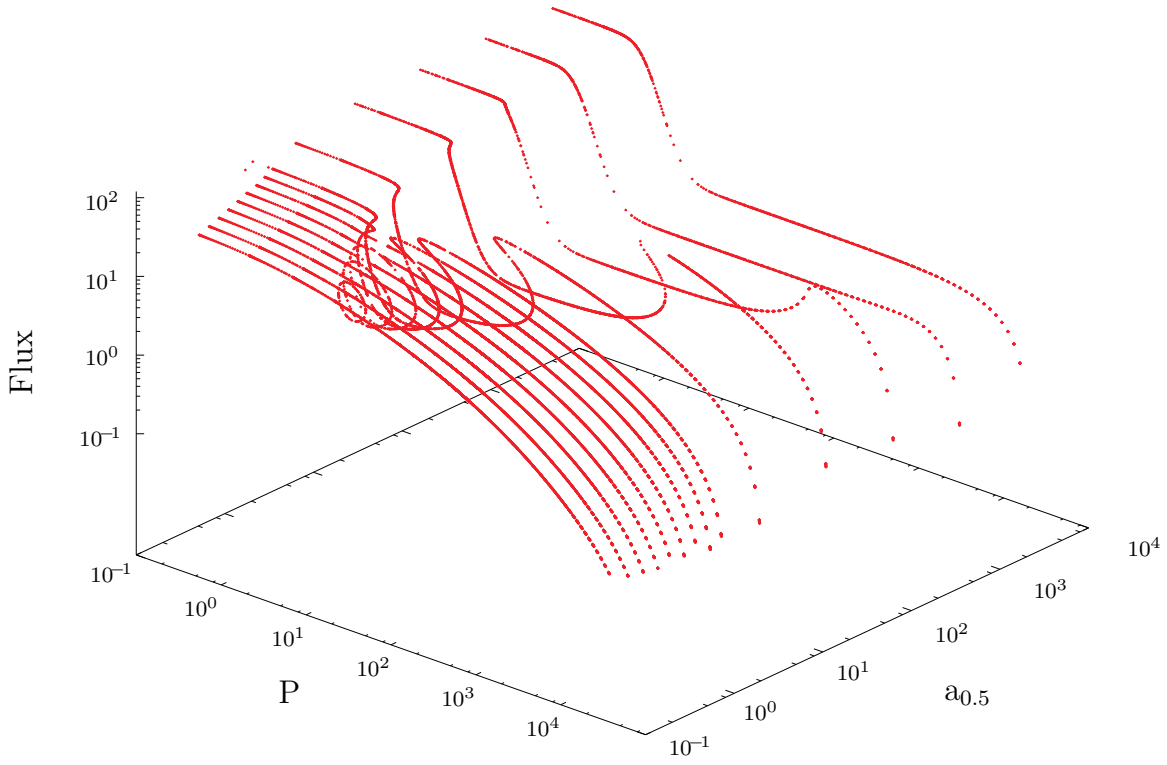


Fig. 6.6 *The effect of changing* $a_{0.5}$. A 3D rate characteristic showing how increasing the strength with which E_1 binds A ($a_{0.5}$) the sigmoidal allosteric supply rate characteristic (weak binding) transforms via double hysteresis (strong binding) to an isola and finally a competitive inhibition curve (very strong binding).

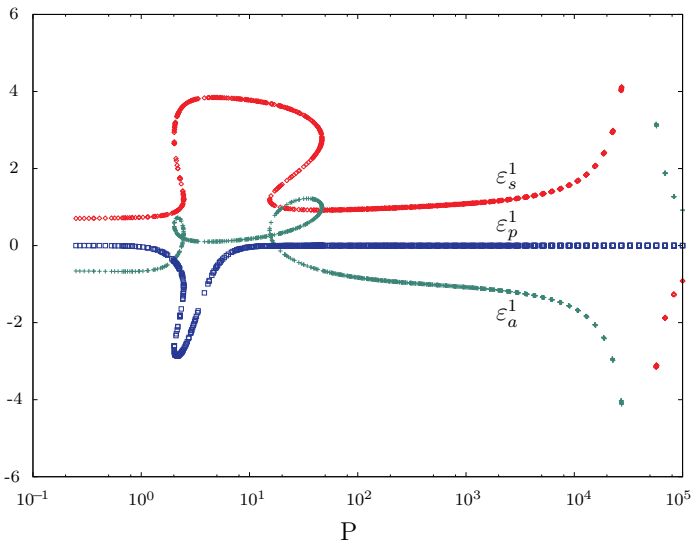


Fig. 6.7 *Elasticity profile of the supply block.* This graph shows the elasticities of E_1 to its substrate, ε_s^1 product, ε_a^1 and effector, ε_p^1 plotted agains the concentration of P when E_1 has a strong affinity for its product ($a_{0.5} = 3.0$). Notice the loops and switchbacks in the hysteresis regions.

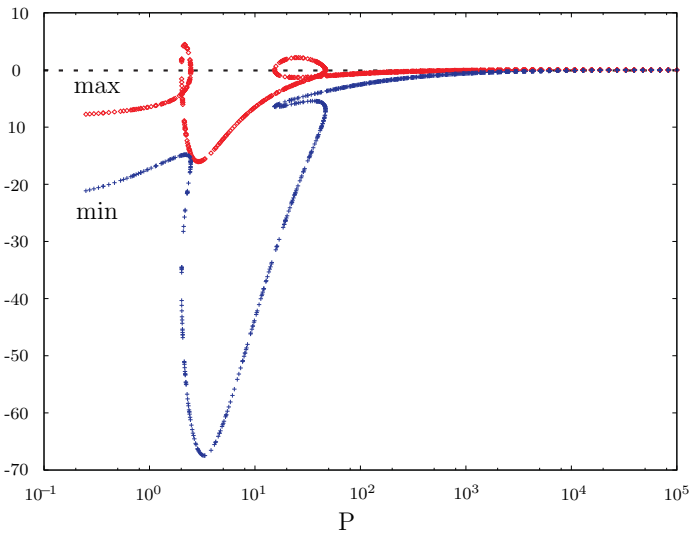


Fig. 6.8 *Eigen value profile of the supply block.* This graph plots the maximum (`max`) and minimum (`min`) real parts of the Eigen values as a function of P when E_1 has a strong affinity for its product ($a_{0.5} = 3.0$). It clearly shows the regions when an Eigen value's real part becomes positive and the system is unstable.

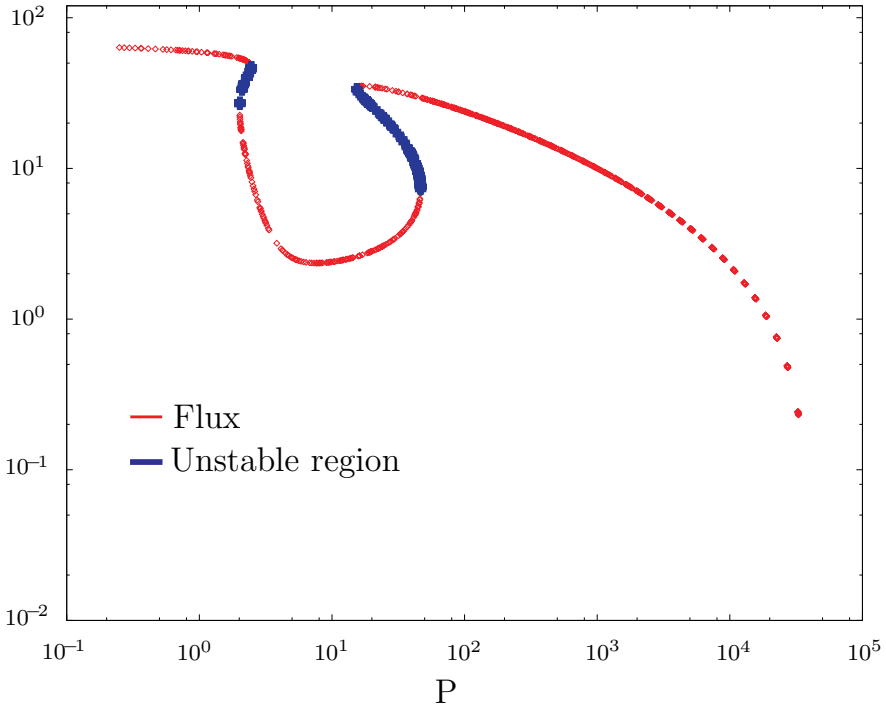


Fig. 6.9 *Flux profile of the supply block.* This figure shows the flux response of the supply block to a change in P when E_1 has a strong affinity for its product ($a_{0.5} = 3.0$). Unstable steady-state solutions are shown in blue.

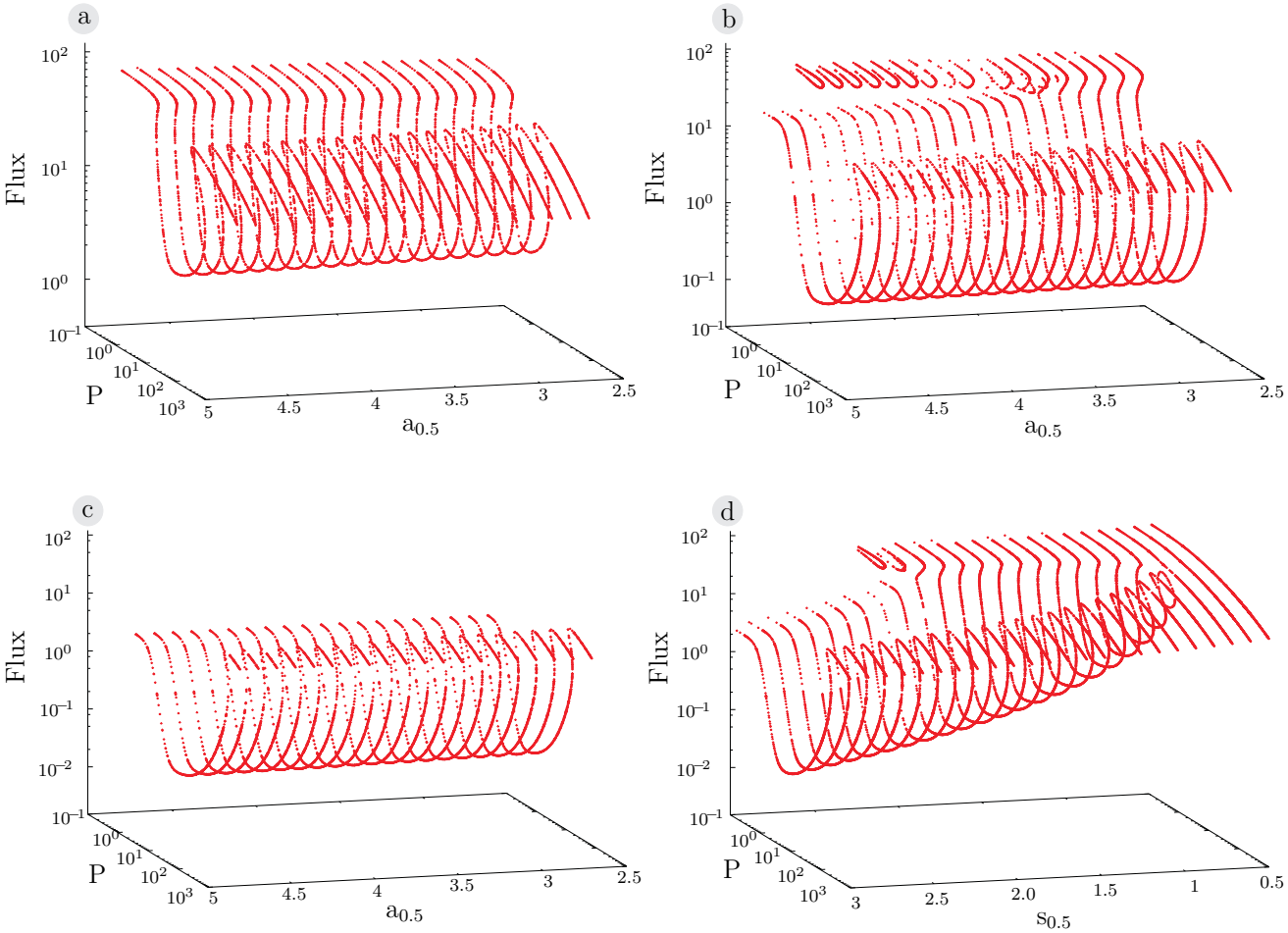


Fig. 6.10 *The effect of changing* $s_{0.5}$. (a) a magnification of part of the 3D rate characteristic shown in Fig. 6.6 where $s_{0.5}$ has a value of 1 (the standard value used in these experiments). Note, the P axis has a maximum of 1000 and that the $a_{0.5}$ axis has a linear scale. (b) the same section is shown except that the $s_{0.5} = 2$ (weaker binding). (c) is the same section again except that $s_{0.5}$ now has a value of 3 (even weaker binding). In (d) $a_{0.5}$ is fixed at 3 and $s_{0.5}$ is scanned.

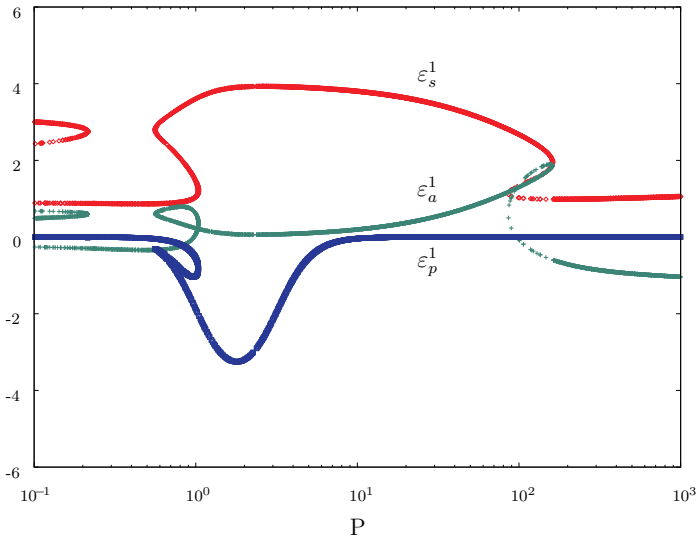


Fig. 6.11 *Elasticity profile of the supply block.* This graph shows the elasticities of E_1 to its substrate, ε_s^1 product, ε_a^1 and effector, ε_p^1 plotted againts the concentration of P when $s_{0.5} = 2.02$ and E_1 has a strong affinity for its product ($a_{0.5} = 3.0$). Notice the loops and switchbacks in the hysteresis regions.

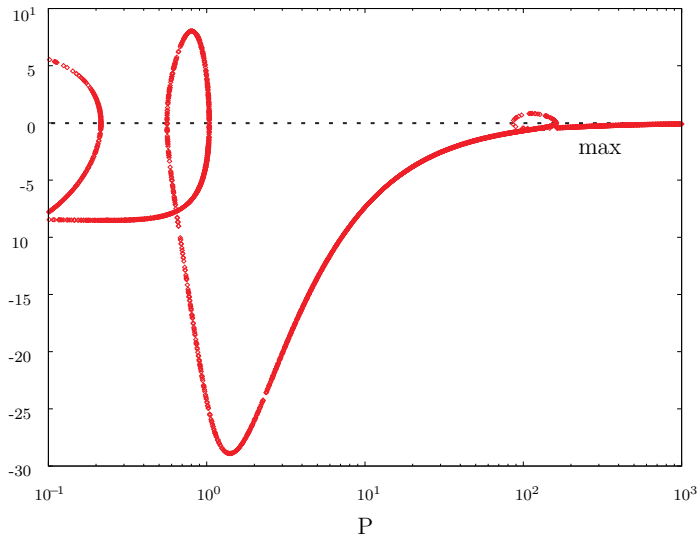


Fig. 6.12 *Eigen value profile of the supply block.* This graph plots the maximum (`max`) real part of the Eigen values as a function of P when $s_{0.5} = 2.02$ and E_1 has a strong affinity for its product ($a_{0.5} = 3.0$). It clearly shows the regions when an Eigen value's real part becomes positive and the system is unstable.

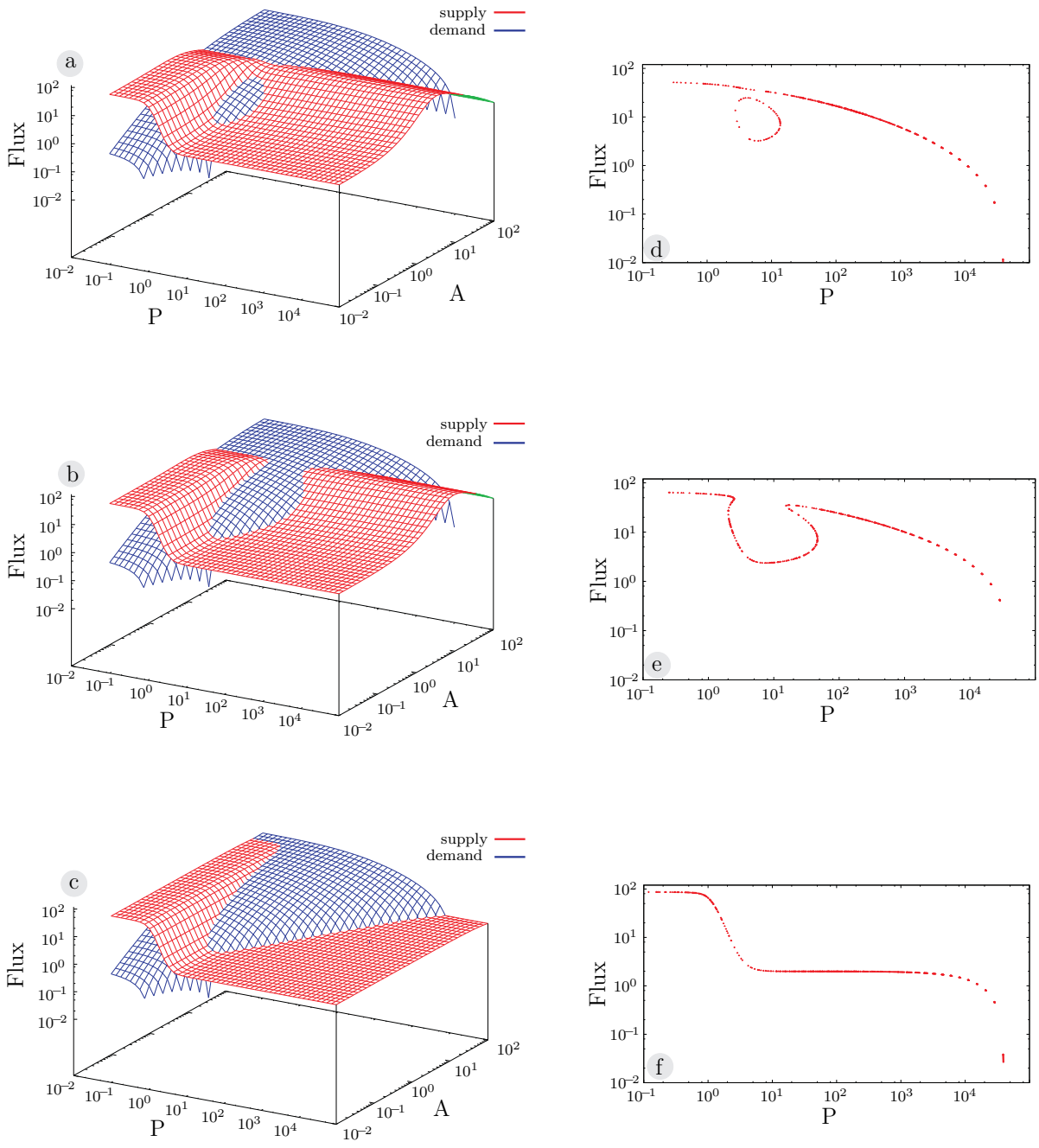


Fig. 6.13 *Supply block rate characteristics.* Here we compare the now familiar supply block rate characteristics generated with PITCON (d)–(f) with the 3D supply–demand rate characteristics generated for the same system (a)–(c) except that A is fixed as the linking metabolite. In graphs (a) and (d) $a_{0.5}$ has been set to one (very strong binding). Graphs (b) and (e) have $a_{0.5}$ is set to three (strong binding) while graphs (c) and (f) have $a_{0.5}$ set to 10000 (weak binding). For each of the aforementioned pairs the behaviour shown in the right hand graph can be decomposed into the interaction between the supply–demand surfaces shown in the left hand one.

7 JWS Online: a web-based resource for kinetic modelling

The JWS Online project is a collaborative effort between Prof. Jacky Snoep and this author. While all publications arising from this collaboration have been jointly published the software is developed as two separate interacting units. Jacky Snoep is responsible for administrating and curating the model database as well as encoding models into stand-alone Java applications using Mathematica and the J/Link language. The development and maintenance of the Java client/server and Python based management software as well as the design, management and implementation of the JWS Online websites has been undertaken by this author.

7.1 Introduction

Kinetic models are powerful tools for describing and understanding the behaviour of complex cellular systems. Since the 1960s and especially in the last decade, there has been a steady increase in the number of models being used to study cellular systems. Today, such models even being considered as the basis for larger projects such as the Silicon Cell whose aim is the modelling of, potentially, an entire cell's metabolism [172].

However, biologists who would like to study these models or use them in education often encounter certain basic problems. For example, no central, publicly available database exists for kinetic models as it does for genomic data. Even when models

are publicly available, they often require a specific software environment to run in. A need, therefore, exists for both a central repository of kinetic models as well as a freely accessible, platform-independent, user-friendly modelling interface. In an attempt to address these problems we[1] have developed the JWS Online modelling system [27].

7.1.1 Do we need another database?

Global genomics initiatives such as the Human Genome Project have led to a rapid increase in the amount of data that needs to be stored in publicly accessible data repositories. Genbank [173] is the classic example of such a database. Other popular databases exist for nucleotide sequences – EMBL [174], protein sequences – SWISS-PROT [175] and enzyme properties – BRENDA [176] and have become essential tools for the modern day biologist. All contain vast amounts of information that can be accessed, searched and extracted. However there is, as yet no similar, publicly-accessible database for kinetic models.

This lack of such a central model repository may restrict the widespread use of kinetic models in a number of ways. One of the biggest problems is that it is often difficult to find all the relevant information needed to reconstruct a working model. It is possible that even if the model is published in the literature and so made publicly available, it can be incomplete and not fully described. Due to a lack of a standard model description, a reconstructed model can be difficult to verify if no reference model description exists. New initiatives, such as the one by the System Biology Markup Language (SBML) Working Group[2] are beginning to address the problems of model interchange and storage in Systems Biology by creating a standard language for describing biochemical reaction networks [121]. At present only a limited number of models are available in SBML format. There are, however, a growing number of SBML compatible applications, examples of which include modelling software such as Jarnac [92], Gepasi [10], Copasi[3] and PySCeS [26] as well as whole cell simulators such as E-Cell [177] or Virtual Cell [178].

One of the problems facing any potential SBML model database is that it would only

[1] This work has been jointly published in [27–29].

[2] `http://www.sbml.org/`

[3] `http://www.copasi.org`

contain the basic model description. This implies that the end user is responsible for installing the software needed to run the model. Even when using a standard description such as SBML, the need to install, configure and learn an application, which might be operating system dependent or expensive to obtain, simply to run a model from the database would limit its accessibility and usefulness. In order to remove this barrier to working with models in such a database, operating system/hardware independent software could be used which would allow the running of models in a wide variety of computing environments. The combination of a central database of models with platform independent software would directly address the problems of the storage and distribution of kinetic models.

7.1.2 Introducing the Java Web Simulation Online system

The Java Web Simulation Online (JWS Online) model repository is a collection of online models that are accessible from a central server that can be run using a web-browser. As the software is browser-based, it is practically independent of the client operating system that it is being run on. This means that the end user is not required to install any software package (web-browsers are ubiquitous to most modern operating systems) in order to run a model contained in the database. Another advantage of browser based software, which stems from its software independence, is that it is useful for teaching purposes, either in the computer laboratory or distance education environments.

The models in the repository are collected using a dual strategy. Firstly, models are transcribed directly from published literature. This strategy is relatively time consuming as published models are sometimes incompletely described and the original authors might no longer be working with them. The second strategy involves collaborations with two journals, the *European Journal of Biochemistry* (from 2005, the *FEBS Journal*) and *Microbiology*. Authors are invited to submit their models to JWS Online where they are made available on an access controlled web site. The online model is then made available to the paper's reviewers. This strategy has the advantage that the database curator is directly in contact with the model's author(s) if a model's complete description is not evident from the paper. Once the paper is accepted the model is moved to the main database and the model is publicly available as soon as the paper is published. A

comment in *Microbiology* [28] has details of this collaboration. Examples of models that have been added to the repository, as a result of this strategy, have been published as [179] and [180].

It can therefore be argued that the development of the web-based, interactive software, and the successful co-operation with leading journals supports the idea that an interactive, web-based repository of kinetic models can lower some of the obstacles to the general adoption and teaching of kinetic models in biology. Using this motivation the following sections in this chapter describe the use and development principles of the JWS Online system.

7.2 JWS Online user's guide

This guide describes the JWS Online system that is freely available over the internet from the main site located at the Stellenbosch University[4] (South Africa) and international mirrors at the Free University of Amsterdam[5] (Netherlands) and the Virginia Bioinformatics Institute[6] (U.S.A).

7.2.1 Navigating the JWS Online web site

All JWS Online web sites have the same basic design. When accessing the site root, the home page displays the site logo and latest news. The site information page contains information about the site, software, contact details and licence. Throughout the site forum links refer to online bulletin boards hosted on the main server. Forum topics include discussions about models, JWS Online software and developer news.

The database page

The core of any JWS Online site is the database page, Fig. 7.1, which consists of three tables containing all the models available on the site (Fig. 7.1 b – e). The largest group of models are the 'silicon cell' or detailed models which are based on experimentally

[4] `http://jjj.biochem.sun.ac.za`
[5] `http://www.jjj.bio.vu.nl`
[6] `http://jjj.vbi.vt.edu`

observed data. Alternatively, the 'core' models are representations of a specific pathway and can illustrate a certain type of behaviour (e.g. [181]), while 'demo' models illustrate a generic metabolic motif, for example a branched pathway. JWS Online already contains a variety of detailed kinetic models, describing a number of metabolic pathways, from a variety of organisms.

- *Saccharomyces cerevisiae*: glycolysis [182] and glycerol synthesis [183].
- Enteric bacteria: phosphotransferase system [184] and tryptophan operon [185].
- Cell cycle models [186].
- Signal transduction models [187].

Aside from the basic model information such as name, author and year published, the database table has links to pages (the info links) containing extended bibliographic information such as the abstract and reference. We have started a collaboration with the Systems Biology Workbench (SBW) group [121] and it is possible to download certain models in SBML format[7] which allows them to be run using an SBML compatible application. Following the model link loads the page containing the online model.

A model page

Besides the standard navigation bar on top of the page a JWS Online model page is divided into three main parts as shown in Fig. 7.2. These three sections contain the basic information needed to study the requested model, namely, Fig. 7.2 a, the web based Java interface to the model repository, Fig. 7.2 b, an HTML image map that describes the model reaction network and c a rate equation panel. Panel Fig. 7.2 c dynamically displays the rate equation of the reaction step selected in panel Fig. 7.2 b. A reaction step is selected by positioning the mouse pointer over the oval representing an enzyme.

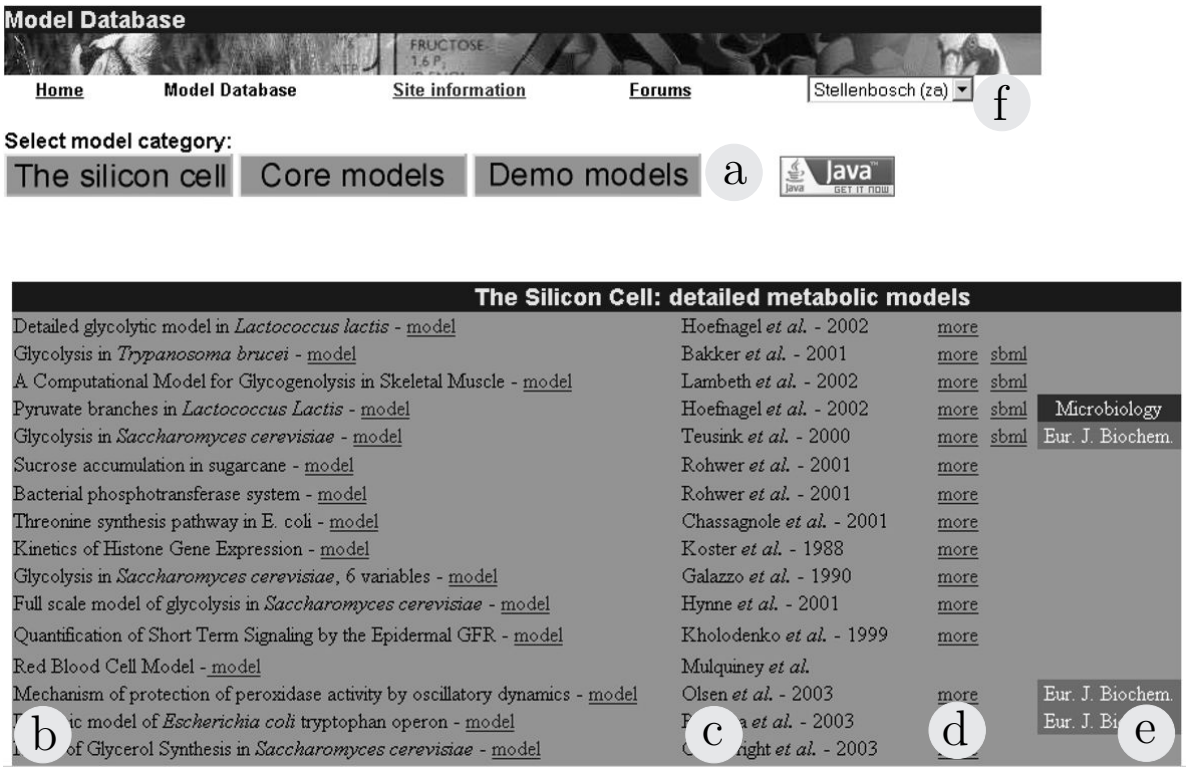


Fig. 7.1 *The JWS Online database page.* a model category selection, b column containing published article's title and link to the online model, c author and year of publication, d link to a page containing bibliographic information and a downloadable SBML file (if available), e if the model was added as part of a journal collaboration, the collaborating journal name is shown here and f is a menu listing the available JWS Online servers.

7.2.2 Using the JWS Online applets

In order to run the Java applet, which acts as a graphical interface to the JWS Online system, an internet connection, web browser and compatible Java Runtime Environment[8] (JRE version 1.3 or newer) needs to be installed. The client has been successfully used on a variety of operating systems[9] and browsers.

- Microsoft Windows 98, 2000, XP using Mozilla 1.4+ and Internet Explorer 5.0 and 6.0

- Mandrake Linux 9.2 and 10.0, RedHat Linux 9.0 using either Mozilla or Galeon

[7] `http://www.sbw-sbml.org/ModelsWebPages/ModelRepository.htm`

[8] Available from `http://www.java.com`

[9] All trademarks and registered trademarks are the property of their respective owners.

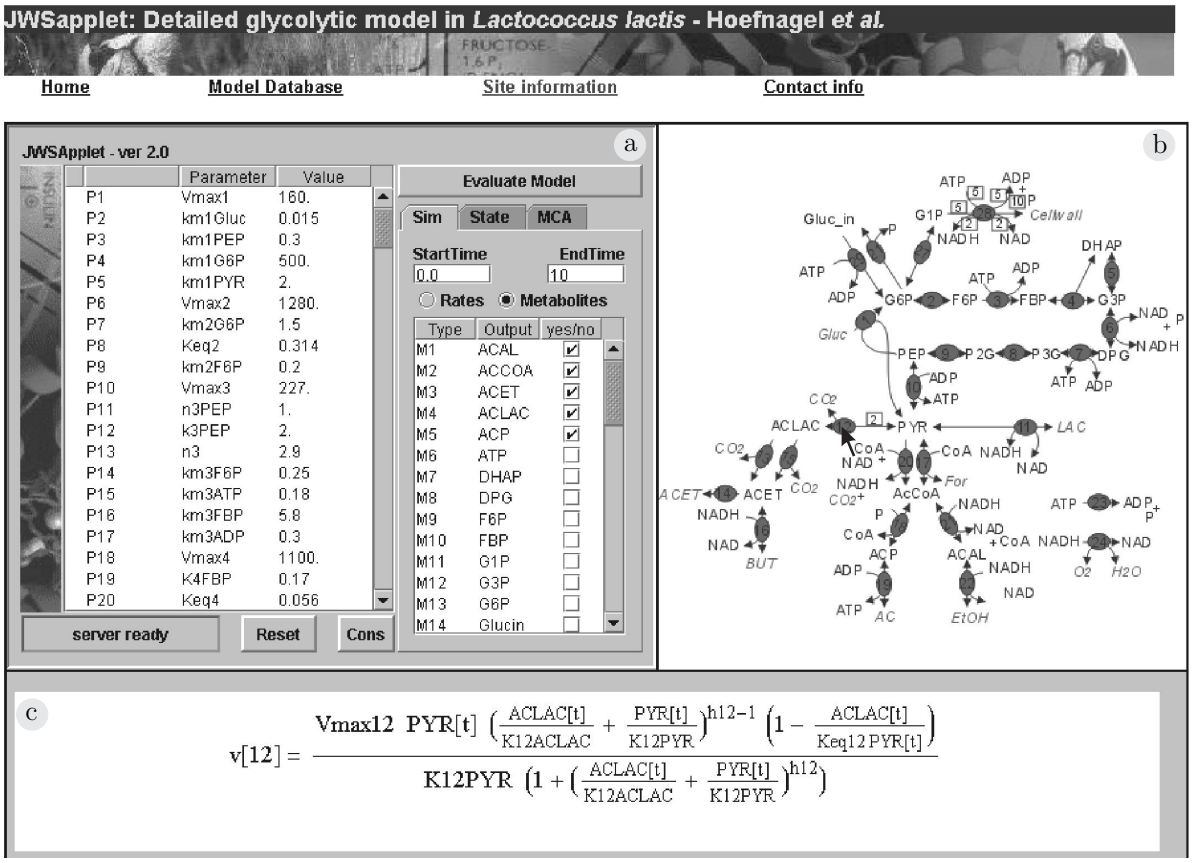


Fig. 7.2 *A JWS Online model page.* a the Java client applet, b an image map of the model reaction scheme which shows the rate equation in the lower panel, c when the pointer is moved over the relevant enzyme. This scheme based on the model presented in [188].

- Apple Macintosh OS X using Safari

Looking at the client interface, as shown in Fig. 7.3, it can be seen that the applet is divided into five main components. On the left side of the applet, Fig. 7.3 a, is a scrollable table of parameter values. This table contains the following columns:

- column one: the parameter number. This can be used for quick referencing of input parameters.
- column two: the parameter name. Parameter names are used as in the literature although additionally the initial values of the time simulation are displayed as `parameter+i`.
- column three: the parameter value. Unlike the previous two columns which are

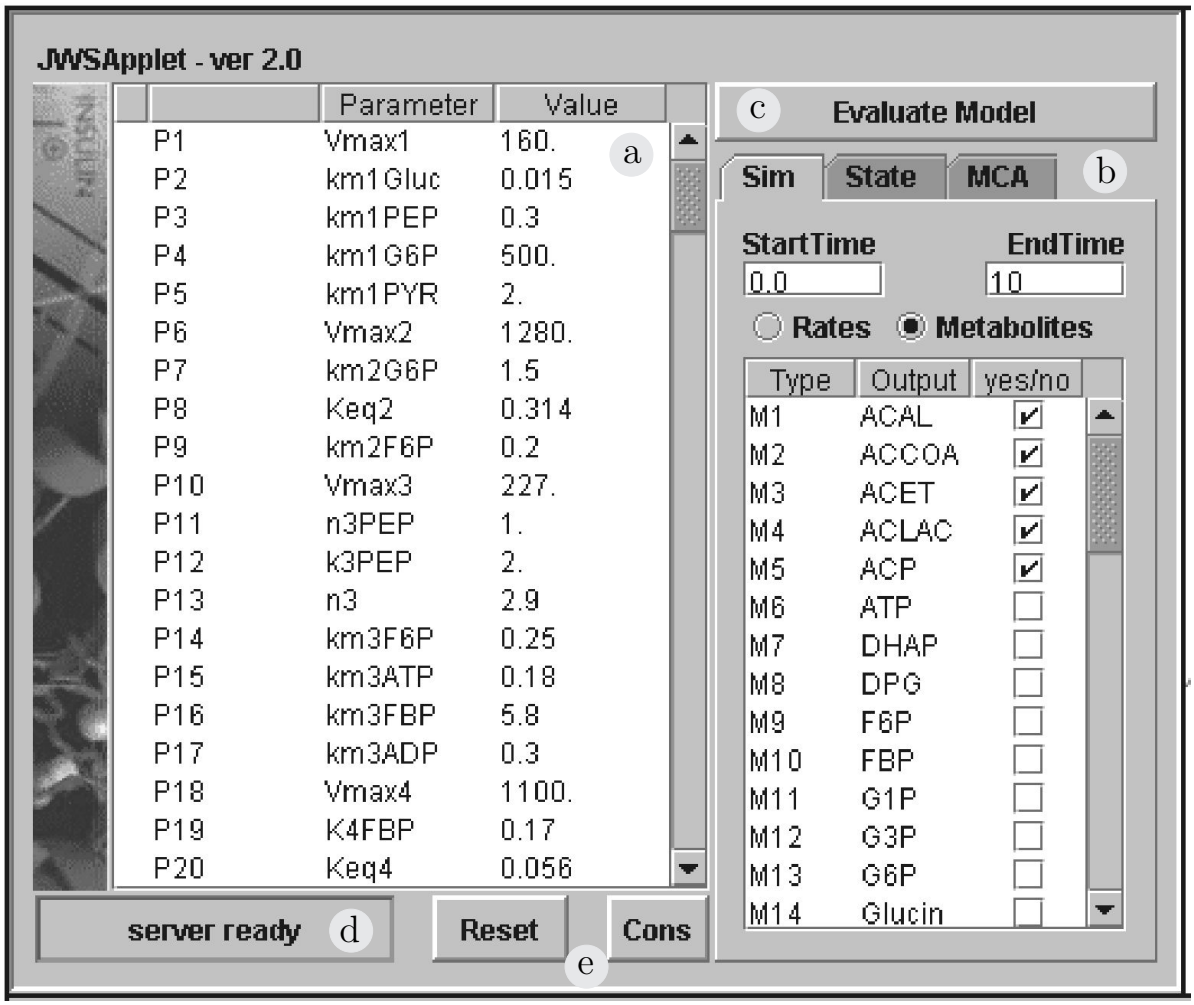


Fig. 7.3 *A typical JWS Online client applet.* (a) input table, (b) analysis selection panels, (c) Evaluate model button, (d) connection status panel and connection status (e) utility buttons.

read-only the values in this column can be changed and set to the desired value.

The input table's behaviour is similar to commonly used spreadsheet applications and it is necessary to 'enter' new data in a cell before the value is accepted as changed. In practice, this means pressing the action key (equivalent to the PC's `<enter>` key) or clicking on another cell in the table. If invalid data is entered into the value column, the default value is used for evaluation and `ERROR` is displayed in the cell. At any time, if the `Reset` button (Fig. 7.3 e) is pressed, the parameters are reset to their default values. When changing model parameters, it is important to be aware of any moiety conserved cycles. In order to display any moiety conservation in a model, the `Cons`

button (Fig. 7.3 e) can be used. Once the applet is started, the connection to the server or applet status is displayed in the message window (Fig. 7.3 d). Displayed messages are colour coded where green is normal, yellow is a warning message and red indicates a serious error. In the case of serious errors, a suggested action or error code is displayed. When an evaluation is started with the `Evaluate model` button (Fig. 7.3 c) the message window border should flash yellow to green, this indicates that the evaluation is proceeding normally.

The evaluation selection panels (Fig. 7.3 b) allow three different types of analysis to be performed. Highlighting a tab on the panel automatically selects the relevant analysis type which can be either, a time simulation (`Sim`), steady-state analysis (`State`) or metabolic control analysis (`MCA`) [60, 61].

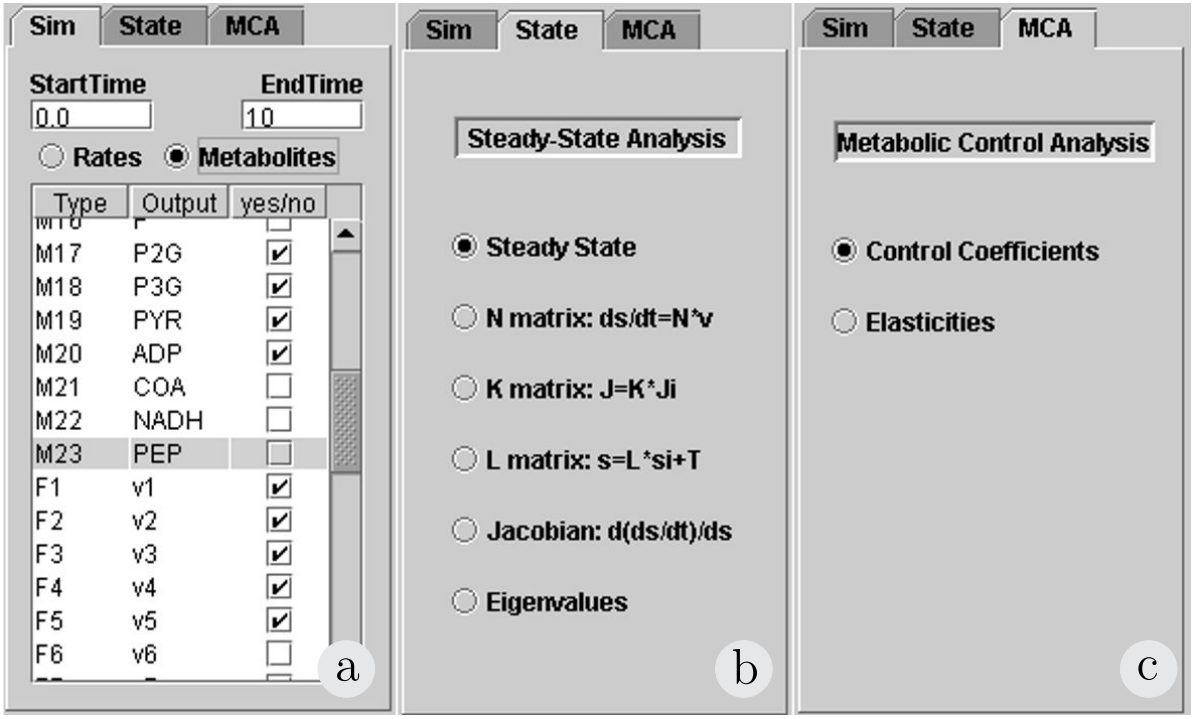


Fig. 7.4 *The three JWS Online applet, analysis types.* a simulation panel, b steady-state panel with structural analysis options and c control analysis panel.

All three analysis panels are simultaneously shown in Fig. 7.4. To begin with, the time simulation panel, Fig. 7.4 a , allows one to set the simulation start and end times and has various output formatting options. Next, either the metabolite concentrations (`Metabolites`) or reaction rates (`Rates`) evolution over time can be selected as output.

Once this selection has been made, the individual rates or concentrations to be displayed on the graph can be selected from the table. The selection table has three columns, an index, variable name and checkbox column, when either `Metabolites` or `Rate` is initially selected the relevant concentration or reaction rate is automatically brought into focus.

In Fig. 7.4 b the steady-state panel is shown which contains options to either display the steady-state fluxes and concentrations or structural properties of the system. These structural properties are displayed as either the stoichiometric (N), kernel (K) or link (L) matrices [78, 82]. The system stability can be investigated by way of the Jacobian matrix and eigenvalues. The final panel, as shown in Fig. 7.4 c, contains options to calculate either the metabolite elasticities or the flux and concentration control coefficients of the system. All elasticities and control coefficients are scaled to the current steady state.

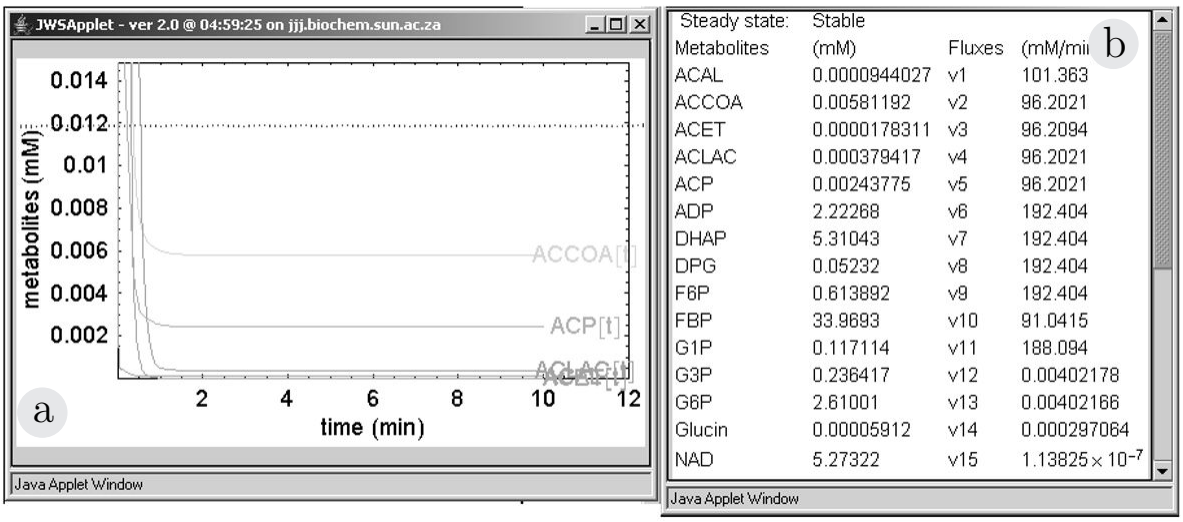


Fig. 7.5 *JWS Online output.* a shows the output of a time simulation, while b shows the results of a steady-state analysis.

Once the analysis type and and output options have been set the model can be run and processed by the server. When the evaluation is complete the results are displayed, as an image, in a separate result window. Fig. 7.5 a shows the result of a time simulation. The results of a steady-state analysis (Fig. 7.5 b) contains the steady-state metabolite concentrations, fluxes and stability (stable or unstable). Fig. 7.6 shows a K matrix and a section of an elasticity matrix. The result of each evaluation is displayed in a separate window and can be quickly resized using the property menu that can be accessed by

right clicking anywhere on the result window.

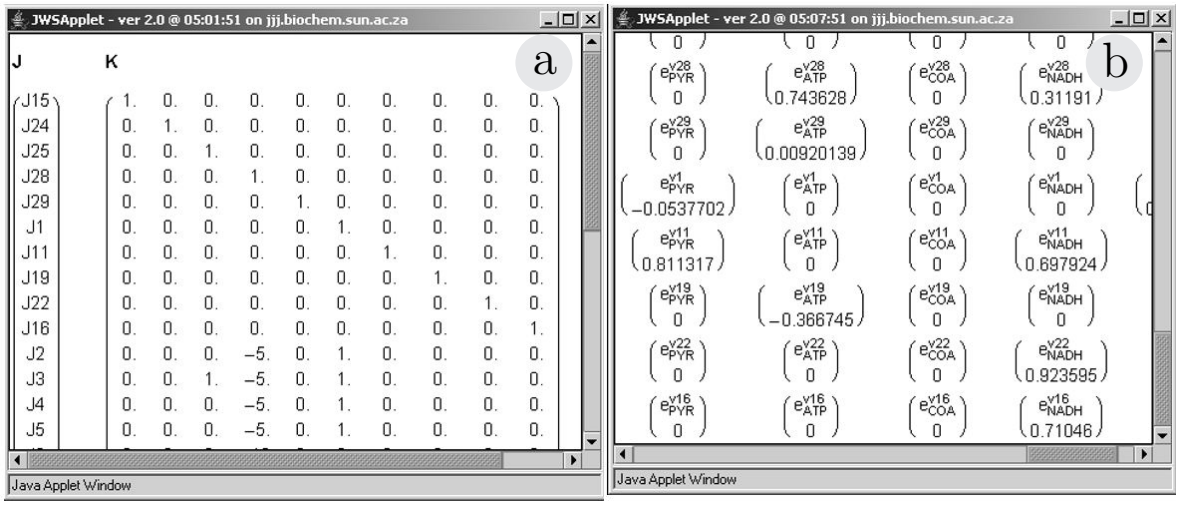


Fig. 7.6 *JWS Online output.* a is an example of a K matrix [82], while b shows a section of an elasticity matrix.

7.3 JWS Online: design and implementation

7.3.1 Introduction

The JWS Online system was initially designed to allow a single model to be run over the internet by a single client. Subsequent redevelopment has transformed this initial design into a multi-user, multi-model application.

Both the client and server software has been written in Sun Microsystem's Java[10] programming language and provide a client-server, web-based interface to kinetic models described in the J/Link language.

JWS Online uses a 'fixed model' approach where each model is encoded as a unique applet – model pair which communicates via a central server. This is in contrast to a 'general' modelling system which might have a single, generic, user interface to many models. The full system consists of three main components: the JWS Online dynamic model server, the graphical user interface (GUI) client applets and model classes.

[10] Java, J2SE, and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. `http://java.sun.com`

The JWS Online server uses the J/Link interface[11] to communicate with Wolfram[12] Research's Mathematica. Using a Mathematica Kernel as a back-end allows a Java application full access to Mathematica's numerical processing and result formatting capabilities. Compiled Java code is run in a Java Virtual Machine, which is available for most computing platforms and allows the design of secure, portable applications [189]. One particular type of application, the Java applet, uses a suitably configured web-browser as an operating environment. Practically this enables the applet to be automatically loaded and run without user intervention.

7.3.2 GUI client applet

Using Java 2 for development (compatible with Sun Microsystems Java versions 1.3 and newer) allows the use of the advanced graphics features contained in the Java Foundation Classes (Swing) graphics toolkit. All JWS Online applets are compiled as part of the `za.ac.sun.jwsapplet` package and accessed from the `<servername>/za/ac/sun/jwsapplet` subdirectory.

Initial versions of the applets were limited in their functionality, especially in the number of parameters that could be displayed and the size of the output data that could be accommodated (see Fig. 1 in [29] for an example of a first generation applet). When redesigning the applet, easy access to all the analysis types, the ability to accommodate large models and the possibility of simultaneously displaying different analysis results were the main factors taken into consideration. In order to have as fine control over the interface as possible, the Java GridBag layout manager was used to control the applet's layout.

As seen in Section 7.2.2, the functional parts of the applet are the parameter input table and the analysis selection panels. In order to keep the applet as compact as possible the analysis options are arranged using a tabbed panel configuration, as shown in Fig. 7.4. Generally, the analysis options do not vary greatly between models; if, however, certain types are not required, e.g. no metabolic control analysis, then the panel containing that analysis type can easily be removed from the applet by commenting out a single

[11] `http://www.wolfram.com/solutions/mathlink/jlink/`
[12] `http://www.wolfram.com`

line of code in the applet source.

On the other hand, what can vary between models is the number of input parameters (kinetic constants, initial concentrations) and the display options (lines shown in the simulation output). Using Swing tables, (e.g. Fig. 7.3 a) provides an elegant solution for this problem as these tables can be embedded into scroll-boxes where the visible window of the table is constant no matter how many parameters are actually listed in the table. One potential drawback of using tables is that after changing a value in the table, the user has to 'enter' the data before a change in a table's data field is registered. Although this behaviour is sometimes problematic, it allows data checking methods to be attached to a table's input fields. Checking methods verify that the input value is of the correct type, for example, string values instead of floating point numbers, parameter range errors such as a binding constants (e.g. K_{m}) having a value less than or equal to zero and general input typographical errors such as removing any spaces from the input. If invalid data is 'entered' an error message is immediately displayed in the relevant table cell.

Once the analysis type has been chosen and the data entered, the model is ready to be evaluated. All parameter and output information is collected and formatted into an ASCII, protocol string which is transmitted to the JWS Online server for processing. When the server has evaluated the model the resulting image is returned to the applet. As the client is multi-threaded, the applet simultaneously displays each result in its own result window.

7.3.3 The JWS Online dynamic model server

Central to the JWS Online system is the dynamic model server which is developed as a stand alone Java application and acts as a communications gateway between the client applets and their associated model processing classes. It has been developed to run on Windows 2000 and operates in conjunction with the Open Source Apache web-server[13].

[13] `http://www.apache.org`

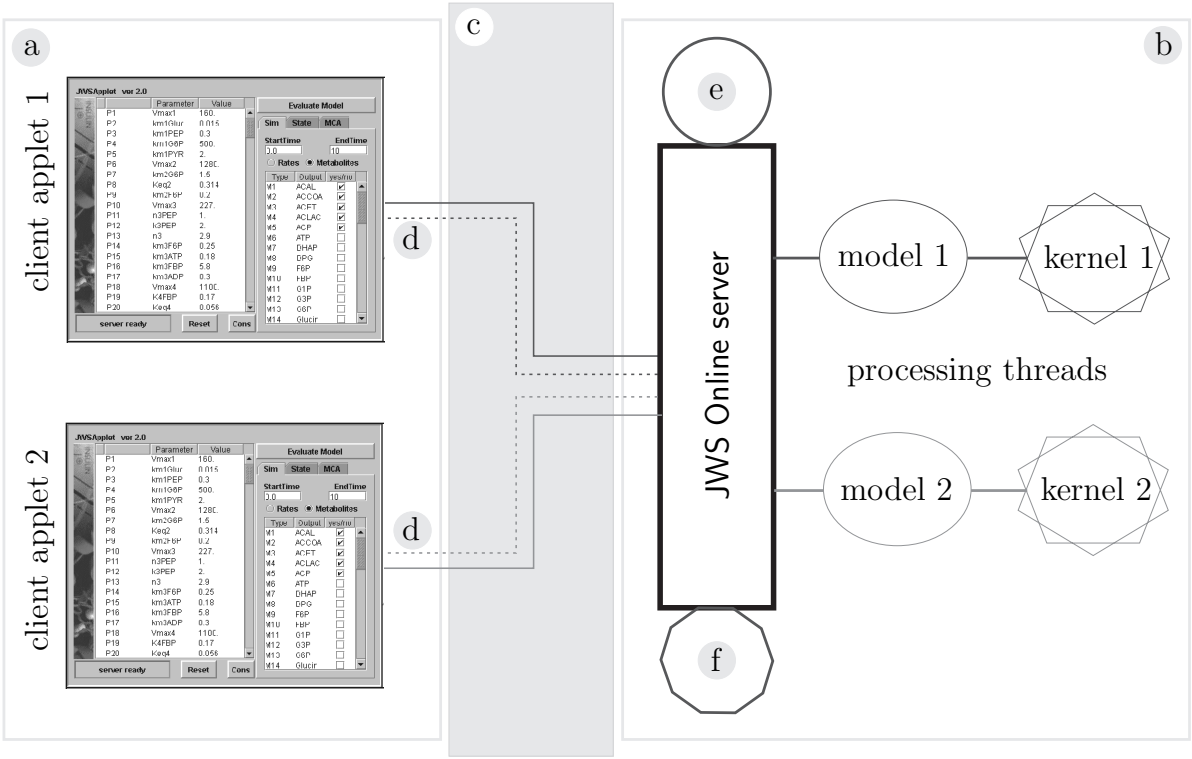


Fig. 7.7 *Schematic of the JWS Online system.* Client side machines (a) with initialized applets connect to a central JWS Online server (b) over a TCP/IP network (c) such as the internet. The applets establish dual socket connections (d) where the port 9000 and 9100 connections are represented by the dashed and solid lines respectively. When the server accepts a client connection, it creates a processing thread for each connecting applet and dynamically instantiates a model class (e) and establishes a Mathematica kernel link (f) which, together, make up a processing thread.

Server initialization

On startup the server loads an ASCII, server configuration file that contains the local path to the Mathematica installation and web server local root directory. It then verifies the existence of its working directories or creates them if necessary and initializes the server log writers. Two log writers are active, the first logs any error and administrative messages that might be generated by the server, while the second logs the IP addresses of connecting clients and model ID into a user log file. Both the log writers write their log files directly into an access limited sub-directory of the web-server root directory ((`<server>/var/log`)). In this way server status and usage can be accessed, via the internet, for any of the JWS Online servers.

A timer routine is started that restarts the server after a fixed time interval. Currently the interval is set to twenty-four hours and is necessary for two main reasons. Primarily the timer routine can help minimize the effects of unpredictable network events that lead to server downtime, for example, unstable client connections can prevent a processor thread from closing and releasing system resources or may even crash the main server routine, making it incapable of accepting new connections. Regularly restarting the server process also 'flushes' unused model class files from the system memory, improving overall memory efficiency. Secondly, once instantiated, the model classes are held in memory by the server and if they are subsequently updated or altered the server needs to reload them from the source files in order for the change to take effect. Updates can therefore be made by server maintainers to remote mirror sites, to which they do not have physical access, which will take effect at the next restart. As the JWS Online server has been designed to run on Windows, this restart is achieved by running the server in a batch file that responds to different exit codes when the server process is terminated. Finally, the server initializes a random number generator for use by the processing threads and the server initialization is complete.

Once the initial environment is set up, the server initializes a networking loop that listens for connections on TCP/IP ports 9000 and 9100. When an applet started on a client computer, Fig. 7.7 a, it attempts to establish connections to the two server ports. When the server, Fig. 7.7 b, detects an incoming connection request it spawns a new processor thread and passes it the active connection established with the client. This

allows the main server thread to return and wait for new incoming connections, making the server capable of dealing simultaneously with multiple clients. Fig. 7.7 illustrates the situation with two client applets. All further transactions between the client and server take place with the newly spawned processor thread. Once the new processing thread is established it initializes a connection to a Mathematica kernel and sets up networking components so that a bi-directional, ASCII connection on port 9000 (Fig. 7.7 d dotted lines) as well as a uni-directional, binary connection to the client on port 9100 (Fig. 7.7 d solid lines) is established.

Dynamic model class loading and processing

Once the applet has connected and is initialized it transmits a unique identification string (the model ID), via port 9000, to the processor thread which, in turn, allows it to dynamically load the model class corresponding to the requesting client. JWS Online model classes are compiled Java modules that contain the model written in J/Link. If, however, this ID string is not recognized as a valid model ID, the processing thread terminates immediately. The initial versions of the server would import the model and instantiate the modules at start up and therefore needed to be recompiled as every new model was added.

By developing the application as a Java package (`za.ac.sun.jws`) and using a Java interface class (the `jwsm` interface) model classes can be imported, instantiated and cast at run time, solely on the basis of the identification string supplied by the client. This differs from the 'normal' Java behaviour where any classes that are to be imported are defined in the source code and imported at compile time. Run-time class loading has the distinct advantage that the process of adding new models does not requires changing the server code or configuration. Once a new applet/model class has been generated the model class is simply placed into the server directory and is loaded when a request is received from its corresponding applet. Dynamic loading is critical to the use of the JWS Online server as an interface to an online model repository as it allows the updating of the server without disrupting its operation, as well as the maintenance of multiple, remote, mirror sites.

Instantiation of the loaded model class now takes place as the processor thread starts

up a Mathematica kernel and establishes a kernel link, Fig. 7.7 f. Each processor thread is linked to an individual Mathematica kernel which is then available for any numerical processing that may be required by the corresponding client applet.

When a model evaluation is run on the client the various analysis types, display options and model parameters are packaged, according to a preset protocol, into an ASCII string which is transmitted via port 9000 to the waiting processor thread. The processor thread parses the initial field of the protocol string, which indicates whether a time simulation or steady state method should be used. All the various analysis types are called via one of these two methods. Once the relevant method is determined the processor thread now passes both the user input and kernel link to the instantiated model class which parses the protocol string back into Java objects and initializes the model with the user input.

JWS Online model classes contain the entire J/Link code needed to evaluate a model in Mathematica. These J/Link expressions are passed to the Mathematica kernel, via the kernel link, which evaluates them and returns a GIF image, formatted as a byte array, as a result. During the evaluation process a `MessageListener` class eavesdrops on any messages generated by the kernel in order to catch any evaluation error messages that might be generated by the kernel. It is necessary to catch certain types of evaluation errors as; for example, the Mathematica non-linear solver FindRoot generates errors if it can't find a solution within a specific tolerance, however, after a few attempts it stops generating this error message and returns the *last best* result. This behaviour is problematic in a client-server environment where a user does not have access to error messages generated by the kernel and so can't differentiate between a genuine and 'last best' solution. The `MessageListener` class has been implemented to intercept specific kernel error messages and override this behaviour. If such a message is intercepted, the model class ignores the result returned by the kernel and instead returns a 100 byte array as a result. Further processing of these errors is discussed in Section 7.3.3.

Once the processor thread receives a byte array from the model, depending on whether the byte array is smaller or larger than a predetermined threshold (4000 bytes), the server utilizes one of two methods to return the result to the client.

Processing small images

If the processor thread receives a byte array smaller than the threshold (excluding 100 byte error arrays) it transmits the result image directly to the applet via a TCP/IP socket. First, however, the applet needs to know the size of the array that it should expect so that it knows how many bytes to read from the server. In order to keep the the applet synchronized with the server, a dual socket strategy (as shown in Fig. 7.7 d) is used. Once the applet has transmitted its evaluation data, it waits and listens on port 9000. In the case of a small image the server first sends the size of the byte array, via port 9000, to the applet and then waits to send the byte array via the binary port 9100 connection. Once the applet receives the array size on port 9000 it then begins reading the required number of bytes from port 9100. When the complete array has been received, the client converts the byte array into a GIF image and displays it in a result window.

Although utilizing a direct approach, i.e. a raw TCP/IP binary data stream, works efficiently on high bandwidth (10/100/1000 Mb/s) networks, over typical internet, low-bandwidth, connections (e.g. a V.90 analogue modem at ≈51 Kb/s), data loss can occur and cause the client to become unstable. Typically, this data loss presents itself as an incomplete result image (a large black area on the image) and the client crashing. This effect is most likely due to a lack of proper buffering on the connection and is dependent on both the size of the byte array being transmitted and the speed of the connection. In order to avoid this situation, the size threshold has been chosen so that even over limited bandwidth connections, data loss does not occur. While simulation results are generally still directly transferred to the applet, an alternate transfer system is used for large images such as the results of a steady-state or control analysis.

Processing large images

Although direct transfer was exclusively used by the original JWS Online server, as the size of the models grew so did the corresponding size of the result images. In order to deal with large images a new strategy was developed which does not involve transferring the image back to the applet directly, but instead allows the client to load the result using a Java URL loading mechanism.

Once the model class returns a byte array to the processing thread, and its size is determined to be larger than the preset threshold value, the processing thread uses the server's random number generator to generate a unique filename. This filename is then used to write the byte array, as a GIF image, to a work directory in the web-server root directory (`<server>/var/appletio`). Although the relative work URL is hard-coded into the applets, the physical work directory is configurable from the server configuration file and allows for different web servers to be used with the JWS Online server.

Once the file is successfully written to the work directory the server transmits the filename back to the applet via port 9000. When the applet receives a filename via port 9000, instead of trying to read the image directly, it forms a URL from the filename supplied by the server and uses the Java ImageIcon URL loader to load the image directly into a result window. This system works well for larger images as the ImageIcon class has built in buffering and image conversion, thereby eliminating the problem of data loss on low bandwidth connections. In order to prevent excessive use of server hard disk space, the processor thread keeps track of the result files that it generates during a modelling session. When a preset number of temporary files have been created, or the session is terminated by the client, all the files associated with the session are automatically deleted.

This strategy also provides a mechanism for the client to display kernel evaluation error messages generated by the model class. If the server detects a 100 byte 'error' array, instead of generating a result image the server transmits a specific array length back to the client on port 9000. If the client identifies this array as an error code, it loads the corresponding error message from a predefined URL on the server and displays it in place of the evaluation result.

7.4 Managing the JWS Online system

Once the main JWS Online server was established and an increasing number of models were being added to the site, it became essential to streamline the process of translating models into web ready applets. The first steps in this process are performed by my

co-developer (Prof. J.L. Snoep), a model is encoded from literature as a Mathematica model and then translated into a stand alone Java program. This Java program contains the parameter initializations and output options as Java statements and the J/Link code needed to evaluate the model.

7.4.1 From stand alone program to web-ready model

Once the stand-alone Java program is developed it can be broken apart and used to construct the applet/model pairs. Initial versions of the applets had custom built interfaces whose layout depended on the number of parameters and options that needed to be displayed. Building this applet (and coding the associated model file) was done by hand and took approximately a week to complete ... clearly too long.

With the redevelopment of the applet and the introduction of Swing tables, a template based approach could be used which reduced the development time of the applet/model pair to a single day. However, redevelopment of the applet/model pairs was coupled with an increase in their size and complexity which inadvertently lead to many 'human errors' slowing down the process. The next logical step was to develop software that could automate this process and reduce the human element.

7.4.2 Generating applet – model pairs

Automating the code generation process turned out to be yet another task to which Python [114] was ideally suited. The first step in the automation process was to develop a Python program that could write out the Java source code of the applet/model pairs using a standard Python based description file (see Appendix 12.1 for an example of this file). Once the Java code was generated, the J/Link code could be extracted by hand from the original Java program, modified so that it was compatible with the model class and inserted into the new model code. Although this significantly speeded up the conversion process, extracting the information needed for the description file and moving the J/Link into the model class, source code was tedious and prone to minor errors that could be difficult to locate.

In order to fully automate this process a (Python based) parser was developed that

uses multiple reads of the input source to decompose the program into a set of temporary files. Any code modifications that need to be performed to convert the Java code to the model class format are automatically performed. Although slow, this method of parsing seemed to be robust and required only minor changes to the format of the original Java source. By using constant parts of the Java program as keys, the parser can effectively deal with the large variations in the length and composition of the J/Link parts of the input file.

Finally these two programs could be integrated together to create the jws_java_generator which only requires the name of the stand alone program, the identifier number and model description (an example session is shown in Appendix 12.2) to be entered manually. Once this information has been supplied, the model is parsed and the Python based model description file generated. Subsequently the code generator performs some basic range tests and generates a complete Java model class and applet. A directory is created using the model number and the applet and model files are written as `a<model number>.java` and `m<model number>.java`. However, the process of developing a web ready applet does not end with the generation of the Java source. Once the applet is compiled, it needs to be tested and a specially tailored HTML page was developed (see Fig. 7.2) which enables it to be loaded in a web browser. Fortunately, this process can be automated and jws_java_generator writes a ready to use web page as `h<model number>.html`. Finally, commands needed to compile the applet/model pair are added to the various batch compiler files.

Each batch compiler file contains all the compiler directives needed to compile either the server and model classes or all the applets for a specific mirror site. Examples of these files can be found in Appendix 12.3. For each mirror site there is an applet compiler file which contains commands to read the server specific source code and compile the applets directly into the correct web site subdirectory.

At this stage the model files are complete and ready to be compiled and used with any JWS Online server. However, for security reasons, the Java Runtime Environment when loaded in a browser will only allow an applet to establish connections to the machine that it was loaded from. Therefore, in order for the applet to connect to a JWS Online server its source code must first be adjusted for a specific server before it can finally

be compiled. In order to streamline this process the JWS Class Management Console, Fig. 7.8 a, was developed.

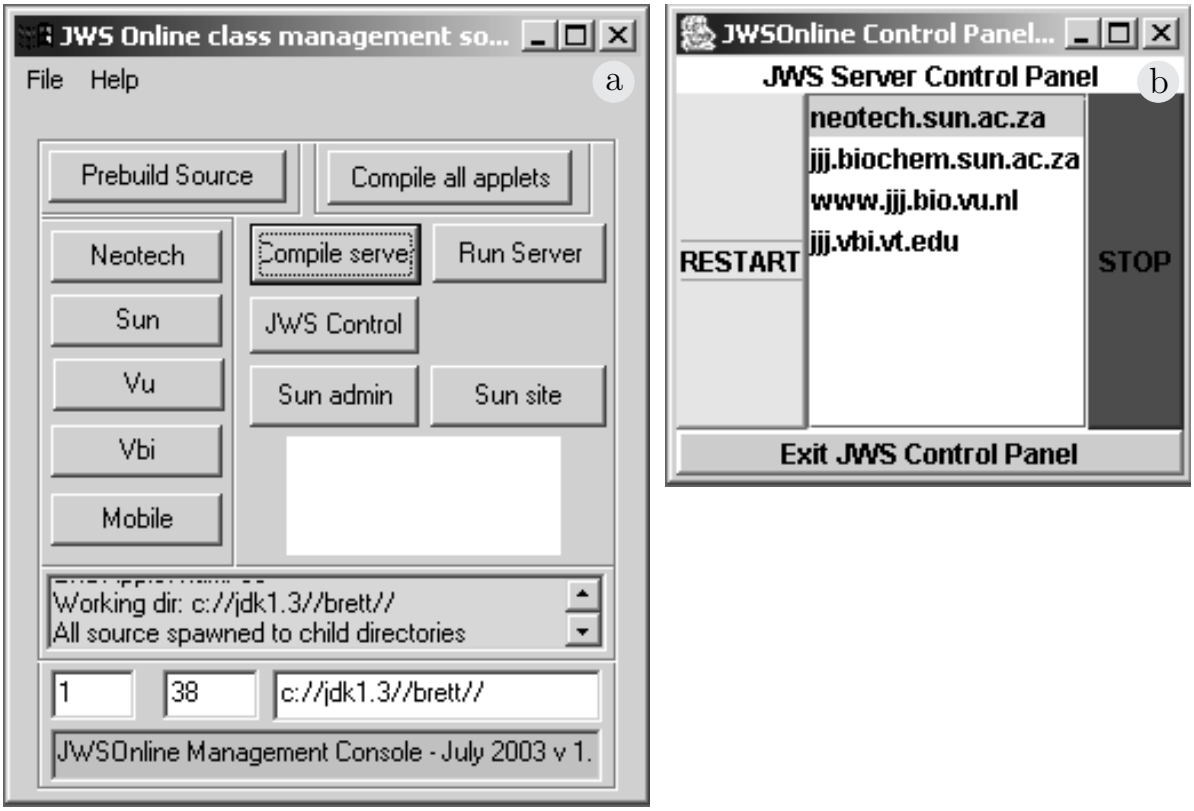


Fig. 7.8 *Tools for managing the JWS Online system.* a, the Class Management Console allows the applet code to be `Prebuilt` for each mirror site and can compile both the server and applets via buttons which run the various batch compiler files. b, the JWS Online Control Panel, allows remote management of the various mirror sites.

Developed using Python (with a PythonCard[14] GUI), the JWS Class Management Console reads the auto-generated applet code and modifies it for the various mirror sites, after which it writes the processed applet source into the relevant mirror's subdirectory where it is later read by the batch compiler files. All the batch compiler scripts can be run from the console, which has options to compile either the server or applets for any of the mirror sites. It is also possible to launch the JWS Control Panel, Fig. 7.8 b, a stand-alone Java program which can access any of the JWS Online servers and either restart or stop them.

Although these management tools may seem trivial, they are vitally important to

[14] `http://pythoncard.sourceforge.net/`

the overall functioning of the JWS Online system. This is especially true as the number of models and mirror sites increases.

7.5 Conclusion

With this chapter I have described the development of the JWS Online system into a usable web-based repository of kinetic models. Through the use of a combination of freely available programming languages – Java and Python, commercial software – Mathematica and J/Link, and open source software – the Apache Web server, it is now possible to take a model described 'on paper' and make it freely accessible via the internet on three international servers. Amongst other things, this means that many classical models that have been often referred to, but never actually run, can now be worked with and studied. It is also possible for an author to submit a paper containing a model to a journal and have the model accessible first for the paper's reviewers and then for the public, as soon as the paper is published. JWS Online has also been used as a concept for larger Silicon Cell projects[15]. Although these larger projects are in the conceptual stages, the JWS Online system has already been used very successfully in the teaching of modelling at both an undergraduate and postgraduate level.

At the time of writing, we are unaware of any freely available, online modelling system that provides services equivalent to those provided by JWS Online to the broader scientific community.

Although a classic cliché, *'the proof of the pudding is in the eating'* still contains a measure of truth in that some of a service's utility depends on its usage. Since its initial inception in 2002, we have seen a steady increase in the number users regularly using the JWS Online system. For example, analysing the log files of the Stellenbosch (South Africa) server shows that over a period of a year, we have repeated usage of the system from clients connecting from Asia, Europe, the USA and South America. This shows that JWS Online is starting to achieve its goal of becoming a global Computational Systems Biology resource.

[15] `http://www.siliconcell.net/` refers to the Amsterdam JWS Online mirror site as: 'Silicon Cell ready to use: the website with silicon cells that can be run over the web'

8 Conclusion

8.1 Future plans

8.1.1 PySCeS

Of course no software application is ever complete and PySCeS has a number of modules which are either in the planning or early development stages.

Systems Biology Markup Language

One such module currently being tested is an interface to the Systems Biology Markup Language[1](SBML) [121]. SBML provides a neutral formal language for exchanging models between various modelling programs. The current SBML implementation (Level 2) tries to cater for a wide range of cellular model types and such as metabolic networks, signal transduction cascades, and gene regulatory networks. However, its flexibility also makes it rather cumbersome and difficult to use. A real danger is that applications may use SBML in a non-standard way, so generating models which cannot be interpreted by other applications. Nevertheless, being able to read and write SBML is invaluable in any application that aims to be of use for the systems biology community. SBML-compatibility of PySCeS has therefore has been given a high development priority. To be fully inter-operable with other SBML application, it is important that PySCeS be able deal with multi-compartment models, which means that functionality for specifying and using compartment volumes must in future be incorporated.

[1] `http://www.sbml.org/`

PySCeS will soon provide a facility for saving models in either SBML level 1 or 2 format, and converting SBML model files to PySCeS input files. It will use the standard `libsbml`-library which is freely available from the SBML group and already has Python bindings.

Stochastic modelling

At present PySCeS is limited to the simulation of deterministic systems. Recently, the modelling of signal transduction and gene regulatory networks that deal with small numbers of component molecules has become increasingly important. It is therefore vital for any modelling application to be able to perform stochastic simulations with algorithms such as those developed by Gillespie and others [190, 191].

Since PySCeS developed in a research environment stochastic that concentrates mostly on deterministic modelling, incorporating the capability for stochastic simulation initially had a low priority. However, no modern cellular simulator is complete without the ability to model stochastic processes and we plan to write PySCeS interfaces to the relevant algorithms.

Strategies for dealing with large models

As more and more of the cell's components are identified and characterized and with the modelling of new types of cellular system (for example signal transduction networks) it is inevitable that models will grow rapidly in size and interconnectedness. This has a direct impact on the speed of an application; for example, matrix operations become significant bottlenecks in program execution. As far as PySCeS is concerned there are a few strategies that we plan to employ to deal with this problem at a computational level.

The first strategy would be to include support for sparse matrix data types and basic operations on sparse matrices. An interesting development in this regard is that the SciPy developers have begun to build sparse matrix support into SciPy and, when fully integrated, these routines will be available for use by PySCeS. A second strategy, which could be especially relevant for stoichiometric analysis, involves switching from the conventional LAPACK routines to modern iterative solvers and factorization algorithms

[192]. A third approach involves parallelization of the core solver and linear algebra routines so that PySCeS is able to take advantage of a high performance, multiprocessor architecture such as a Beowulf cluster[2]. This could also include a job scheduler that distributes parameter and optimization routines over multiple processors.

The wider use of PySCeS

PySCeS has now developed up to a point where it fulfills its original design specifications of accessibility and flexibility, and includes the tools that were planned when it was first conceptualized. However, to move beyond being an in-house tool and become an widely-used modelling program it needs to be accessible to the systems biology community. To that end it is significant that PySCeS is the only interactive modelling application that runs on all the major operating systems, namely Windows, Linux and OS10. It also has significant advantages in being based on a standardized library of algorithms (SciPy) and in that it can be simply included as a module in a larger application. This makes PySCeS an attractive point of departure for other projects; the design of PySCeS should allow it to integrate with other systems biology tools and frameworks, such as the Systems Biology Workbench (SBW) [23]. To make PySCeS available as widely as possible is can be obtained freely from the major repository of open source software, namely Sourceforge.

8.1.2 JWS Online

The JWS Online project already plays an important role in the systems biology community in that it is at present the only interactive repository of metabolic models. However, up until now we have concentrated on optimising its interactivity rather than its function as a database. To rectify this, we have started to make every model in the repository available for download in either SBML or as a PySCeS input file. This will strengthen JWS Online's claim to be a repository of curated models.

On the technical side of things JWS Online is being redeveloped to use webMathematica which will greatly simplify the management and administrative aspects of the JWS Online system.

[2] `http://www.beowulf.org/`

8.2 Summary

> "Contrariwise", continued Tweedledee, "if it was so, it might be; and if it weren't so, it would be: but as it isn't, it ain't. That's logic."
> – Lewis Carrol in 'Through the Looking Glass'

In the introduction I argued that the program of mechanization, which has been so successful at providing an insight into the physical world, was now being continued in the biological sciences in the form of digitization. Implicit in this process is the assumption that in some way or another it must be possible to quantify the processes which make up a 'living' cell. We then briefly considered two such quantifying frameworks: kinetic modelling and metabolic control analysis. Underlying these frameworks is the realisation that biological systems are inherently complex, particularly when considered in terms of cellular processes. As a consequence a purely analytical study of these systems will provide little insight into their systemic behaviour: for this one needs an extensive set of tools to model (digitize) it[3].

In this dissertation this interplay between biology, theory and computational ability is well illustrated by the investigation into the cause of multi-stationarity in the behaviour of an end-product inhibited linear metabolic pathway. This study acted as a catalyst for much of the work presented in this dissertation as it *could not proceed* with the computational tools available at the time. PySCeS was developed to continue the research into this problem. As described in Chapter 6, with PySCeS we were finally able to investigate and characterise the multiple steady states. However, although this might have been one of the initial reasons for the development of a new software package from scratch, as the project developed new aims were set. These were to develop a general modelling platform that was accessible to a wide range of users, flexible in its design that had all the features needed to model cellular systems.

In this regard PySCeS has succeeded in that it is the only interactive modelling package, capable of doing metabolic control analysis, that runs natively on all the major operating systems. It is also the only fully featured Python based modelling solution

[3]It should be noted, however, while the quantification of cellular processes is necessarily a reductionistic process, biological systems are not necessarily deterministic in the sense that the functioning of the whole is completely describable as an aggregate sum of its parts.

that is not designed to work with any specific user interface. PySCeS can therefore be easily used as a modelling component in any Python based application. Through its continuation capabilities it is also uniquely suited to study cellular systems that exhibit multi-state behaviour. Not only has PySCeS proved extremely useful in our own research and teaching, but it we hope that it will do the same for the computational systems biology community in their study of the integrated processes of life.

Finally, using a completely different approach, JWS Online has been developed and serves a unique purpose as being the only freely available, interactive model repository. The growing number of collaborations between JWS Online and various journals and systems biology projects such as the Silicon Cell shows that it fulfils a real need and is already an important resource for the computational systems biology community.

> "The scientific community has emerged in this century as the only genuine world community that I can think of. It has had nothing to connect it to the special interests of nation-states, it carries out its work of inquiry without respect for national borders, it passes its information around as though at a great party; and because of these habits, science itself has grown and prospered. Every researcher, in whatever laboratory, depends for the work's progress on the cascades of information coming in from other laboratories everywhere, and sends out his laboratory's latest findings on whatever wind is, at the moment, blowing.
>
> I do not believe science can be done in any other way, and I hope and pray that the world community of basic scientists can stay free, exchanging everything they know for the pure pleasure of exchange–*competing*, of course, but playing with delight against all odds in a huge endless game rather than in a narrow contest for new business or new weapons. For science is only at its beginning, and almost everything important lies still ahead to be learned about and comprehended." – Lewis Thomas in 'The Fragile Species' [193]

9 Modelling with Python and SciPy appendix

This appendix contains the complete program listing for the model described in Chapter 3.

9.1 Complete code listing

```
# PySCeS model of simple system containing a branch and a
# moiety-conserved cycle - Version 1.1 20-04-2002
# Brett Olivier and Jannie Hofmeyr, Triple-J Group
# for Molecular Cell Physiology, University of Stellenbosch
#
# This model is based on one published in
# 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 2nd International Conference on Systems Biology,
# pp. 291--300. Omnipress, Madison, WI, USA.

,,,
A simple model to illustrate that the Python/Scipy
combination provides everything needed to
1. calculate the time-course and steady state;
2. do a parameter scan of the steady state;
3. graph the results;
4. and many other things too fierce to mention...

The reaction network:
X0 + S2 <=1=> S1 + S3
S3 + X6 <=2=> S2 + X7
```

```
S1 <=3=> X4
S1 <=4=> X5

with rate equations
v1 = V1*X0*S2/K1_X0*K1_S2/
   (1 + X0/K1_X0 + S2/K1_S2 + X0*S2/K1_X0*K1_S2)
v2 = V2*S3*X6/K2_S3*K2_X6/
   (1 + S3/K2_S3 + X6/K2_X6 + S3*X6/K2_S3*K2_X6)
v3 = V3*S1/(S1 + K3_S1)
v4 = V4*S1/(S1 + K4_S1)
'''

# Step 1: Import the scipy packages
import scipy

# Step 2: Define rate and differential equations in functions
v = scipy.zeros((4),'d')          # create rate vector
def rate_eqs(S):
    v[0] = V1*X0*S[1]/K1_X0*K1_S2/\
         (1 + X0/K1_X0 + S[1]/K1_S2 + X0*S[1]/K1_X0*K1_S2)
    v[1] = V2*S[2]*X6/K2_S3*K2_X6/\
         (1 + S[2]/K2_S3 + X6/K2_X6 + S[2]*X6/K2_S3*K2_X6)
    v[2] = V3*S[0]/(S[0] + K3_S1)
    v[3] = V4*S[0]/(S[0] + K4_S1)
    return v

def diff_eqs(S,t):
    Sdot = scipy.zeros((3),'d')       # create ODE vector
    # Calculate dependent S3 conc
    S[2] = S_tot - S[1]
    # Calculate rates
    v = rate_eqs(S)
    # Differential equations
    Sdot[0] = v[0] - v[2] - v[3] # dS1/dt
    Sdot[1] = v[1] - v[0]        # dS2/dt
    Sdot[2] = v[0] - v[1]        # dS3/dt  (only for output)
    return Sdot

# Step 3: Initialise variables and parameters
X0 = 1.0; X4 = 0.0; X5 = 0.0; X6 = 1.0; X7 = 0.0
```

```
S1 = 0.0; S2 = 0.7; S3 = 0.3; S_tot = S2 + S3
V1 = 1.0; V2 = 1.0; V3 = 2.0; V4 = 1.0

K1_X0 = 1.0; K1_S2 = 1.0
K2_S3 = 1.0; K2_X6 = 1.0
K3_S1 = 1.0; K4_S1 = 1.0

# Step 4: Set values for time-course calculation
t_start = 0.0; t_end = 10.0; t_inc = 0.1
t_range = scipy.arange(t_start,t_end+t_inc,t_inc)

# Step 5: Call integration routine to calculate time course
t_course = scipy.integrate.odeint(diff_eqs,[S1,S2,S3],t_range)

print 'Integration output: concentrations'
print t_course

# Step 6: Calculate steady-state
# using last state of integration as initial values
fin_t_course = scipy.copy.copy(t_course[-1,:])

# Call steady-state solver
# args=None needed because diff_eqs has extra input t
ss = scipy.optimize.fsolve(diff_eqs, fin_t_course, args=None)

print 'Steady-state concentrations'
print ss

# Calculate steady-state fluxes from [steady state]
print 'Steady-state fluxes'
print rate_eqs(ss)

# Step 7: Parameter scan (V4) of steady state solving for fluxes
# Set up parameter scan conditions
S_start = 0.0; S_end   = 20.0; S_inc   = 0.1
S_range = scipy.arange(S_start,S_end+S_inc,S_inc)

ss_init = scipy.copy.copy(fin_t_course)
scan_table = scipy.zeros((len(S_range),5), 'd')
```

```
# Scan parameter V4 and solve for steady-state fluxes
for i in range(len(S_range)):
    V4 = S_range[i]
    ss = scipy.optimize.fsolve(diff_eqs, ss_init, args=None)
    J = rate_eqs(ss)
    scan_table[i,0] = S_range[i]
    scan_table[i,1:] = J
    ss_init = ss

# Step 8: Plot results using Scipy gnuplot
# Access Scipy Gnuplot module
from scipy import gplt

# Plot the time course results
gplt.plot(t_range, t_course[:,0], t_range, t_course[:,1],\
          t_range, t_course[:,2])
gplt.xtitle('Time'); gplt.ytitle('Concentration')
gplt.text((3,0.1),'s1'); gplt.text((3,0.58),'s2')
gplt.text((3,0.42),'s3'); gplt.grid('off')

# Plot the parameter scan results
# uncomment the following 6 lines to plot parameter scan results
##gplt.plot(S_range, scan_table[:,1], S_range, scan_table[:,2],\
##          S_range, scan_table[:,3], S_range, scan_table[:,4])
##gplt.xtitle('V4'); gplt.ytitle('Flux'); gplt.grid('off')
##gplt.text((5,0.172),'v1'); gplt.text((5,0.16),'v2')
##gplt.text((5,0.053),'v3'); gplt.text((5,0.112),'v4')

raw_input('\n Press Return to continue\n')
```

10 PySCeS appendix

10.1 Licences for external modules used by PySCeS

10.1.1 NLEQ2

The ZIB institute[1] has kindly given us permission to use and distribute the NLEQ2 algorithm as part of PySCeS, providing such usage does not contravene their licence terms as described below.

```
* Licence
  You may use or modify this code for your own non commercial
  purposes for an unlimited time.
  In any case you should not deliver this code without a special
  permission of ZIB.
  In case you intend to use the code commercially, we oblige you
  to sign an according licence agreement with ZIB.
```

If you are not using PySCeS for research or personal use you **must** disable NLEQ2 installation by setting `nleq2 = 0` in setup.py or obtain a licence from ZIB. As this is a plug-in solver, not installing NLEQ2 will not affect the way that the rest of PySCeS functions except that the solver won't be available for steady-state calculations.

10.1.2 MetaTool

MetaTool which is fully described in metatool/readme.txt

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

[1] http://www.zib.de/

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

10.2 MetaTool compile scripts

10.2.1 build_win32.bat

```
rem PysCeS: Metatool compile script

g++ meta4.3_double.cpp -Wno-deprecated -O3 -o meta43_double.exe
g++ meta4.3_int.cpp -Wno-deprecated -O3 -o meta43_int.exe

exit
```

10.2.2 build_linux

```
#!/bin/sh
# PysCeS: Metatool compile script

g++ meta4.3_double.cpp -Wno-deprecated -O3 -o meta43_double
g++ meta4.3_int.cpp -Wno-deprecated -O3 -o meta43_int

exit
```

10.3 PySCeS configuration files

10.3.1 Windows configuration file

```
[Pysces]
metatool_dir = c:\python23\lib\site-packages\pysces\metatool
install_dir = c:\python23\lib\site-packages\pysces
output_dir = c:\python23\lib\site-packages\pysces\output
model_dir = c:\python23\lib\site-packages\pysces\pscmodels

[PyscesModules]
metatool = 1
pitcon = 1

[XternalModules]
nleq2 = 1
```

10.3.2 Linux configuration file

```
[Pysces]
metatool_dir = /usr/lib/python2.3/site-packages/pysces/metatool
install_dir = /usr/lib/python2.3/site-packages/pysces
output_dir = os.path.join(os.path.expanduser('~'),'pysces')
model_dir = os.path.join(os.path.expanduser('~'),'pysces')

[PyscesModules]
metatool = 1
pitcon = 1

[XternalModules]
nleq2 = 1
```

10.4 Formal PySCeS input file syntax

```
Extract from lexparse.py (c) Copyright Jannie Hofmeyr, 2002

Lexical analyser and parser for a string or file containing the
set of reaction equations defining a model.

2. REACTION LINE (required)

ReactionID:       {StoichCoef}Reagent + ... + {StoichCoef}Reagent
            = | > {StoichCoef}Reagent + ... + {StoichCoef}Reagent
                  Rate equation (optional)
ReactionID:    name given to the reaction, e.g., R1 or ATPase
               (must begin with a letter,otherwise any sequence of [a-zA-Z0-9_],
               no spaces allowed)
StoichCoef:    stoicheiometric coefficient
               (integer or decimal number) enclosed in curly brackets
Reagent:       name of a reactant or product
               (must begin with a letter,otherwise any sequence of [a-zA-Z0-9_],
               no spaces allowed)
=              denotes reversible reaction
>              denotes irreversible reaction
```

10.5 An example PySCeS input file

```
# PySCeS input file
# Simple linear pathway (2004)

FIX: x0 x3

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

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

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

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

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

10.6 PySCeS user configuration files

10.6.1 Windows: pys_usercfg.ini

```
[Pysces]
output_dir = C:\\mypysces
model_dir = C:\\mypysces\\pscmodels

# Notes:
# It is recommended to use escaped backslashes
# for user defined paths eg. c:\\temp\\dir

# PySCeS (0.1.3) configuration file
```

10.6.2 Linux: .pys_usercfg.ini

```
[Pysces]
output_dir = /home/bgoli/pysces
model_dir = /home/bgoli/pscmodels

# Notes:
# Pysces (0.1.3) configuration file
```

10.7 F2PY wrapper written for NLEQ2

```
!    -*- f90 -*-
python module nleq2__user__routines
    interface nleq2_user_interface
        subroutine fcn(n,x,f,ifail) ! in nleq2.f
            integer optional,check(len(x)>=n),depend(x) :: n=len(x)
            double precision dimension(n),intent(in) :: x
            double precision dimension(n),depend(n),intent(out) :: f
            integer intent(hide) :: ifail
        end subroutine fcn
        subroutine jac(n,m1,x,a,ifail) ! in nleq2.f:n1int:unknown_interface
            integer optional,check(len(x)>=n),depend(x) :: n=len(x)
            integer optional,check(shape(a,0)==m1),depend(a) :: m1=shape(a,0)
            double precision dimension(n),intent(in) :: x
            double precision dimension(m1,n),depend(n),intent(out) :: a
            integer intent(hide) :: ifail
        end subroutine jac
    end interface nleq2_user_interface
end python module nleq2__user__routines
python module nleq2 ! in nleq2.f
    interface  nleq2
        subroutine nleq2(n,fcn,jac,x,xscal,rtol,iopt,ierr,liwk,iwk,lrwk,rwk)
            use nleq2__user__routines
            integer optional,check(len(x)>=n),depend(x) :: n=len(x)
            external fcn
            external jac
            double precision dimension(n), intent(in,out) :: x
            double precision dimension(n),depend(n), intent(in,out) :: xscal
            double precision intent(in,out):: rtol
            integer dimension(50), intent(in,out):: iopt
            integer intent(out):: ierr
            integer optional,check(len(iwk)>=liwk),depend(iwk) :: liwk=len(iwk)
            integer dimension(liwk),intent(in) :: iwk
            integer optional,check(len(rwk)>=lrwk),depend(rwk) :: lrwk=len(rwk)
            double precision dimension(lrwk),intent(in) :: rwk
        end subroutine nleq2
    end interface
end python module nleq2.f
! See http://cens.ioc.ee/projects/f2py2e/
```

10.8 PySCeS code used to generate Fig. 5.6

10.8.1 PySCeS input file

```
# Edelstein, B.B. (1970) Biochemical Model with Multiple Steady States
# and Hysteresis : J. Theor. Biol., 29(1), 57-62

FIX: A B

Flux1:
    A + X > {2}X
    A*X*k1 - X*X*k_1

Flux2f:
    X + E > C
    X*E*k2

Flux2r:
    C > X + E
    C*k_2

Flux3f:
    C > E + B
    C*k3

Flux3r:
    B + E > C
    B*E*k_3

A = 8.35
B = 0.2
C = 1E-7
E = 30.0
X = 0.1
k1 = 1.0
k_1 = 1.0
k2 = 1.0
k_2 = 1.0
k3 = 1.0
k_3 = 1.0
```

10.8.2 PySCeS script demonstrating parameter continuation

```
# version of a model by Edelstein
# converted from an example that ships with Jarnac 2.0

import scipy, pysces

mod = pysces.model('edelstein.psc')
mod.doLoad()

mod.pitcon_par_space = scipy.linspace(5,9,60)
mod.pitcon_iter = 5
mod.pitcon_flux_gen = 0

scan_parameter = 'A'
res = mod.PITCON(scan_parameter)

F = open('pysx_edelstein_data.html','w')
mod.Write_array_html(res,F,Col=['A']+list(mod.metabolites))
F.close()

pysces.plt.plot2D(res,0,range(1,len(mod.metabolites)+1),\
                  ykey=mod.metabolites,fmt='w p')
pysces.plt.xrng(8.0,9.0); pysces.plt.yrng(0,22)
pysces.plt.xlabel(scan_parameter)
pysces.plt.title('Parameter plot - Edelstein model')

pysces.plt.save_html('pysx_edelstein')

print '\n**\nLoad pysx_edelstein.html in ' + pysces.output_dir +\
      ' to view result\n'
```

10.9 Creating a PITCON extension library

10.9.1 Fortran subroutine PITCON1

Fortran wrapper routine used to wrap the PITCON routine. The lines beginning with `cf2py` are F2PY directives which allow a Python extension library to be generated directly from this Fortran source without any intermediate customization.

```
C dpcon61w.f 29 February 2004

CFILE: dpcon61w.f
      SUBROUTINE PITCON1(DF,FPAR,FX,IERROR,IPAR,IWORK,LIW,NVAR,RWORK,
     *LRW,XR,IMTH)
      EXTERNAL  DF
      EXTERNAL  FX
      EXTERNAL  DENSLV
      EXTERNAL  BANSLV
      INTEGER   LIW
cf2py integer optional,check(len(iwork)>=liw),depend(iwork) :: liw=len(iwork)
      INTEGER   LRW
cf2py integer optional,check(len(rwork)>=lrw),depend(rwork) :: lrw=len(rwork)
      INTEGER   NVAR
cf2py integer optional,check(len(xr)>=nvar),depend(xr) :: nvar=len(xr)
      DOUBLE PRECISION FPAR(*)
cf2py double precision dimension(*)),intent(in) :: fpar
      INTEGER   IPAR(*)
cf2py integer dimension(*),intent(in) :: ipar
      INTEGER   IWORK(LIW)
cf2py integer dimension(liw),intent(in,out,copy) :: iwork
      INTEGER   IERROR
cf2py integer intent(out) :: ierror
      DOUBLE PRECISION RWORK(LRW)
cf2py double precision dimension(lrw),intent(in,out,copy) :: rwork
      DOUBLE PRECISION XR(NVAR)
cf2py double precision dimension(nvar),intent(in,out,copy) :: xr
cf2py DOUBLE PRECISION X(NVAR), FVEC(NVAR), FJAC(NVAR,NVAR)
cf2py double precision,intent(out) :: fvec
cf2py double precision,intent(out) :: fjac
cf2py INTEGER IMTH = 1
      INTEGER IMTH
```

```
cf2py CALL FX(NVAR,FPAR,IPAR,X,FVEC,IERROR)
cf2py CALL DF(NVAR,FPAR,IPAR,X,FJAC,IERROR)
      IF (IMTH.eq.1) THEN
         CALL PITCON(DF,FPAR,FX,IERROR,IPAR,IWORK,LIW,NVAR,RWORK,
     *LRW,XR,DENSLV)
      ELSE
         CALL PITCON(DF,FPAR,FX,IERROR,IPAR,IWORK,LIW,NVAR,RWORK,
     *LRW,XR,BANSLV)
      ENDIF
      END
```

10.10 Calculating the eigenvalues

The following code example illustrates the use of the `scipy.info()` function to obtain more information on the eigenvalue evaluation function `scipy.linalg.eig`. In particular, the definitions of the left/right eigenvectors as well as the eigenvalues returned by PySCeS are formally defined.

```
import scipy
scipy.info(scipy.linalg.eig)

 eig(a, b=None, left=0, right=1, overwrite_a=0, overwrite_b=0)

Solve ordinary and generalized eigenvalue problem
of a square matrix.

Inputs:

  a     -- An N x N matrix.
  b     -- An N x N matrix [default is identity(N)].
  left  -- Return left eigenvectors [disabled].
  right -- Return right eigenvectors [enabled].

Outputs:

  w      -- eigenvalues [left==right==0].
  w,vr   -- w and right eigenvectors [left==0,right=1].
  w,vl   -- w and left eigenvectors [left==1,right==0].
```

```
  w,vl,vr  -- [left==right==1].

Definitions:

  a * vr[:,i] = w[i] * b * vr[:,i]

  a^H * vl[:,i] = conjugate(w[i]) * b^H * vl[:,i]

where a^H denotes transpose(conjugate(a)).
```

10.11 Coding a 2D parameter scan

```
# Example 2 dimensional parameter scan
# Uses the linear1 model supplied with PySCeS
# Brett 2004

import scipy,pysces
mod = pysces.model('linear1')
mod.doLoad()

# create a list to hold the results
scan_results = []

for D1par in range(1,11,1):
    mod.x0 = D1par
    for D2par in range(1,21,1):
        mod.k2 = D2par
        mod.doMca()
        # append the scan parameters and results to the list
        scan_results.append([D1par, D2par, mod.s0ss,\
                             mod.s1ss, mod.s2ss, mod.JR1,\
                             mod.ecR1_x0,mod.ccJR1_R3])

# recast the list as an array ... why we like Python!
scan_results = scipy.array(scan_results)

# plot the results
pysces.plt.plot3D(scan_results,0,1,4,zkey='JR1',fmt='w p')
pysces.plt.xlabel('x0'); pysces.plt.ylabel('k2')

raw_input('\nPress enter to continue ...')
```

10.12 Using the PySCeS unit tests

```
>>> import pysces

>>> pysces.CopyModels()
src : c:\python23\lib\site-packages\pysces\pscmodels
dest: C:\mypysces\pscmodels
branch1.psc ok
linear1.psc ok
moiety1.psc ok

>>> pysces.test(lvl=2)

                C  PCPRB1.FOR  The Freudenstein-Roth function.
                C  Limit points in the first variable occur at:
                C    (14.28309, -1.741377,  0.2585779)
                C    (61.66936,  1.983801, -0.6638797)


Limit point in run = 4
[ 14.28309125  -1.74137688   0.25857788]

Limit point in run = 10
[ 61.66936258   1.98380112  -0.66387974]
.
----------------------------------------------------------------------
Ran 1 test in 0.010s

OK
<pysces.PyscesTest.PyscesTest instance at 0x0112BE40>

>>>
```

11 Using PySCeS appendix

11.1 PySCeS scripts online

The PySCeS input files (*.psc files) for the models used in Chapter 6 and certain example scripts are described in this appendix. All the results (including some not shown in Chapter 6) can be regenerated by downloading and running the following example PySCeS programs.

Investigate the effect of changing $a_{0.5}$

http://glue.jjj.sun.ac.za/~bgoli/pysces/examples/isola_thesis_ex1.py

Generating supply–demand surfaces

http://glue.jjj.sun.ac.za/~bgoli/pysces/examples/isola_thesis_ex2.py

Investigating elasticities and Eigen values

http://glue.jjj.sun.ac.za/~bgoli/pysces/examples/isola_thesis_ex3.py

Time simulations showing two stable states

http://glue.jjj.sun.ac.za/~bgoli/pysces/examples/isola_thesis_ex4.py

Investigating the effect of changing $s_{0.5}$ - part 1

http://glue.jjj.sun.ac.za/~bgoli/pysces/examples/isola_thesis_ex5.py

Investigating the effect of changing $s_{0.5}$ - part 2

http://glue.jjj.sun.ac.za/~bgoli/pysces/examples/isola_thesis_ex6.py

11.2 PySCeS input file: thesis_isola.psc

```
# A three enzyme linear system with feedback
# which exhibits multiple steady-state solutions
# Brett Olivier, Johann Rohwer and Jannie Hofmeyr

FIX: S P

R1:
    S = A
    V1*(S/S05)*(1 - A/(S*Keq1))*((S/S05)+(A/A05))**(h-1)/
    ((S/S05 + A/A05)**h + (1 + (P/P05)**h)/(1 + alpha*(P/P05)**h))

R2:
    A = B
    V2*(A - B/Keq2)/(A + K2A*(1 + B/K2B))

R3:
    B = P
    V3*(B - P/Keq3)/(B + K3B*(1 + P/K3P))

Keq1 = 400.0
V1 = 200.0
S05 = 1.0
A05 = 3.0
P05 = 1.0
h = 4.0
alpha = 0.01

V2 = 100.0
Keq2 = 10.0
K2A = 1.0
K2B = 1.0

V3 = 100.0
Keq3 = 10.0
K3B = 1.0
K3P = 1.0

S = 1.0
```

```
A = 10.0
B = 10.0
P = 1.0
```

11.3 Creating a 3D rate characteristic

```
import scipy, pysces

mod = pysces.model('isola_thesis')
mod.doLoad()
mod.doState()
mod.showState()

range_low = scipy.log10(0.1)
range_high = scipy.log10(100000)

mod.pitcon_iter = 15
A05range = scipy.concatenate((scipy.logspace(0,1,7)*0.5,\
                              scipy.logspace(1,4,5)*0.5),1)

scan_parameter = 'P'

print '\n*** Running continuation ***\n'
res3d = []
res3dout = []

mod.mode_state_init = 3
mod.mode_state_init3_factor = 0.5
mod.SetQuiet()

mod.write_array_header = 0
G = open('isola_thesis_3d.dat','w')

go1 = 1
cntr = 0

timer = pysces.TimerBox()
timer.step_timer('s1',len(A05range))
```

```
for A05 in A05range:
    timer.s1_cstep += 1
    print '\n', timer.s1.next()
    mod.A05 = A05
    cntr += 1
    if mod.A05 > 3 and mod.A05 < 30:
        mod.pitcon_par_space = scipy.logspace(range_low,range_high,130)
        mod.pitcon_iter = 20
    elif mod.A05 >= 1 and mod.A05 <= 3:
        mod.pitcon_par_space = scipy.logspace(range_low,range_high,140)
        mod.pitcon_iter = 15
    else:
        mod.pitcon_par_space = scipy.logspace(range_low,range_high,70)
        mod.pitcon_iter = 15
    if go1:
        res3d = mod.PITCON(scan_parameter,A05)
        mod.Write_array(res3d,G)
        go1 = 0
    else:
        resTmp = mod.PITCON(scan_parameter,A05)
        res3d = scipy.concatenate((res3d,resTmp))
        mod.Write_array(resTmp,G)
G.close()

pysces.plt.plot3D(res3d,0,1,4,zkey='Flux',fmt='w p')
pysces.plt.xrng(0.1,10000)
pysces.plt.yrng(0.1,100000)
pysces.plt.logx()
pysces.plt.logy()
pysces.plt.ylabel(scan_parameter)
pysces.plt.title('The effect on the supply rate characteristic\
                 of a decreasing A05')
pysces.plt.xlabel('A05')
pysces.plt.save_html('isola_thesis_3d')

print pysces.session_time()
raw_input("\n\tPress <enter> to continue\n")
```

11.4 Decomposing the supply block

PySCeS Input file: isola_thesis2.psc

```
# A three enzyme linear system with feedback
# which exhibits multiple steady-state solutions
# this model breaks the system into 2 pieces by
# fixing A along with S and P
# Brett Olivier, Johann Rohwer and Jannie Hofmeyr

FIX: S A P
R1:
    S = A
    V1*(S/S05)*(1 - A/(S*Keq1))*((S/S05)+(A/A05))**(h-1)/
    ((S/S05 + A/A05)**h + (1 + (P/P05)**h)/(1 + alpha*(P/P05)**h))

R2:
    A = B
    V2*(A - B/Keq2)/(A + K2A*(1 + B/K2B))

R3:
    B = P
    V3*(B - P/Keq3)/(B + K3B*(1 + P/K3P))

Keq1 = 400.0
V1 = 200.0
S05 = 1.0
A05 = 3.0
P05 = 1.0
h = 4.0
alpha = 0.01

V2 = 100.0
Keq2 = 10.0
K2A = 1.0
K2B = 1.0

V3 = 100.0
Keq3 = 10.0
K3B = 1.0
```

```
K3P = 1.0

S = 1.0
A = 10.0
B = 10.0
P = 1.0
```

PySCeS script

```
"""
Simultaneously generates 6 data sets from two model
description files. Files saved as isola_thesis2_rc_XXX.dat
are 3D rate characteristic surfaces that are compatible with
GnuPlot while isola_thesis2_pitcon_XXX.dat are
2D rate characteristics that can be plotted with either Gnuplot
or PySCeS.
"""

import scipy, pysces
import sys, os


mod = pysces.model('isola_thesis2')
mod.doLoad()
mod.doState()
mod.SetQuiet()
mod.write_array_header = 0

mod2 = pysces.model('isola_thesis')
mod2.doLoad()
mod2.SetQuiet()
mod2.write_array_header = 0

A_steps = 36
A_range = scipy.logspace(-2,2,A_steps)
P_steps = 48
P_range = scipy.logspace(-1,5,P_steps)
A05_range = [1.,3.,10000.]

mod2.mode_state_init = 3
mod2.mode_state_init3_factor = 0.5
mod2.pitcon_par_space = scipy.logspace(-1,5,120)
mod2.pitcon_iter = 15

for A05 in A05_range:
    mod.A05 = A05
    final_output = []
```

```
print 'Working on plot for A05 = ' + `mod.A05`
for P in P_range:
    mod.P = P
    result = []
    scanpar = []
    par3d = []
    for x in A_range:
        mod.A = x
        mod.State()
        result.append(mod.state_flux)
        scanpar.append(x)
        par3d.append(P)

    out1 = scipy.array(result)
    out2 = scipy.array(scanpar)
    out3 = scipy.array(par3d)
    out2.shape = (len(out2),1)
    out3.shape = (len(out3),1)
    output = scipy.concatenate((out2,out3,out1),1)
    final_output.append(output)

    out = file('isola_thesis2_rc_' + str(int(A05)) + '.dat','w')

for x in final_output:
    mod.Write_array(x,out)
out.close()

mod2.A05 = A05
result = mod2.PITCON('P')

pysces.plt.plot2D(result, 0, [3], ykey=['JR1'], log=1, fmt='w p')
pysces.plt.xrng(0.1,100000)
pysces.plt.yrng(0.01,120)
pysces.plt.gridoff()
pysces.plt.xlabel('P'); pysces.plt.ylabel("Flux")
pysces.plt.title('PITCON parameter scan (A05='+str(A05)+')')

mod2.Write_array(result,open('isola_thesis2_pitcon_' + str(int(A05))\
 + '.dat','w'))
```

```
raw_input("\n\tPress <enter> to continue\n")
```

This PySCeS code generates the complete data set for the 3D rate characteristics, however, `scipy.gplt` can not display surface grids properly and GnuPlot is needed to visualize the data. The following is the GnuPlot plot file that was used to display the results of preceding PySCeS program.

```
#!/gnuplot
#       G N U P L O T
#       Version 3.8f patchlevel 0
#       last modified Wed Apr 11 19:12:46 IST 2001
#       System: MS-Windows 32 bit
#
#       Copyright(C) 1986 - 1993, 1999, 2000
#       Thomas Williams, Colin Kelley and many others

set xrange [ 0.01 : * ] noreverse nowriteback
set yrange [ 0.01 : *] reverse nowriteback
set zrange [ 0.01 : 120 ] noreverse nowriteback

set xlabel "A2" -1,-1
set ylabel "P" -2,-1
set zlabel "Flux" -5,-6
set hidden3d
set view 60,300 # 3D V-X view

set logscale xyz

set origin 0,0
set size 1,1

set title "A05 = 1" 2,4
splot 'c:\mypysces\isola_thesis2_rc_1.dat' u 1:2:3 t 'supply' w l,\
'c:\mypysces\isola_thesis2_rc_1.dat' u 1:2:4 t 'demand' w l
pause -1 "Next"

set origin 0,0
set size 1,1

set title "A05 = 3" 2,4
```

```
splot 'c:\mypysces\isola_thesis2_rc_3.dat' u 1:2:3 t 'supply' w l,\
'c:\mypysces\isola_thesis2_rc_3.dat' u 1:2:4 t 'demand' w l
pause -1 "Next"

set title "A05 = 10000" 2,4
splot 'c:\mypysces\isola_thesis2_rc_10000.dat' u 1:2:3 t 'supply' w l,\
'c:\mypysces\isola_thesis2_rc_10000.dat' u 1:2:4 t 'demand' w l
pause -1 "Next"

##########################

set xrange [ 0.1 : 100000 ] noreverse nowriteback
set yrange [ 0.01 : 120] noreverse nowriteback
set xlabel "P" -1,-1
set ylabel "Flux" 0,-1


set pointsize 0.1

set origin 0,0
set size 1,1

set title "A05 = 1" 2,4
plot 'c:\mypysces\isola_thesis2_pitcon_1.dat' u 1:4 t 'JR1' w p
pause -1 "Next"

set origin 0,0
set size 1,1

set title "A05 = 3" 2,4
plot 'c:\mypysces\isola_thesis2_pitcon_3.dat' u 1:4 t 'JR1' w p
pause -1 "Next"

set title "A05 = 10000" 2,4
plot 'c:\mypysces\isola_thesis2_pitcon_10000.dat' u 1:4 t 'JR1' w p
pause -1 "End"
```

11.5 Exploring stability with PySCeS

```
import scipy, pysces
import sys

mod = pysces.model('isola_thesis')

mod.doLoad()
mod.doState()
mod.showState()

mod.mode_state_init = 3
mod.mode_state_init3_factor = 0.5
mod.SetQuiet()

mod.pitcon_iter = 15
mod.pitcon_flux_gen = 0

range_low = scipy.log10(0.1)
range_high = scipy.log10(100000)
density = 75

mod.pitcon_par_space = scipy.logspace(range_low,range_high,density)

scan_parameter = 'P'
gformat = 'w p'

print '\n*** Running continuation ***\n'
res = mod.PITCON(scan_parameter)

timer = pysces.TimerBox()
timer.step_timer('s1',res.shape[0])

elasRes = []
eigRes = []
unstableRes = []
cntr = 0

print '\n*** Calculating the elasticities and eigen values ***\n'
for x in range(res.shape[0]):
```

```
    timer.s1_cstep += 1
    cntr += 1
    if timer.s1_cstep == 1 or cntr == 200:
        print '\n', timer.s1.next()
        cntr = 0

    mod.P = res[x,0]

    mod.state_metab = res[x,1:]
    mod.state_flux = mod.FluxGen(res[x,1:])

    mod.EvalEvar(mod.state_metab,mod.state_flux)
    mod.EvalEpar(mod.state_metab,mod.state_flux)
    mod.EvalEigen()

    elasRes.append([res[x,0]] + [mod.state_flux[0],mod.ecR1_S,\
                                 mod.ecR1_A, mod.ecR1_P])

    eigRes.append([res[x,0]] + [mod.state_flux[0],max(mod.eigen_values[0].real,\
                                mod.eigen_values[1].real),\
                                min(mod.eigen_values[0].real,\
                                mod.eigen_values[1].real)])

    if mod.eigen_values[0].real >= 0.0 or mod.eigen_values[1].real >= 0.0:
        unstableRes.append([res[x,0]] + [mod.state_flux[0],\
                                         max(mod.eigen_values[0].real,\
                                         mod.eigen_values[1].real),\
                                         mod.ecR1_S, mod.ecR1_A, mod.ecR1_P])

elasRes = scipy.array(elasRes)
eigRes = scipy.array(eigRes)
unstableRes = scipy.array(unstableRes)

mod.write_array_header = 0

F = open('isola_thesis3_elas.dat','w')
mod.Write_array(elasRes,F)
F.close()
G = open('isola_thesis3_eigen.dat','w')
mod.Write_array(eigRes,G)
```

```
G.close()
H = open('isola_thesis3_unstable.dat','w')
mod.Write_array(unstableRes,H)
H.close()

if raw_input("\n\tProceed with plot (y/n): ") in ['n','N','no','No','NO']:
    sys.exit(0)

pysces.plt.plot2D(elasRes, 0, [2,3,4], ykey=['ecR1_S', 'ecR1_A',\
                                            'ecR1_P'], fmt='w p')
pysces.plt.yrng(-6.0,6.0)
pysces.plt.xrng(0.1,100000)
pysces.plt.logx()
pysces.plt.gridoff()
pysces.plt.xlabel('P')
raw_input("\n\tPress <enter> to continue\n")

pysces.plt.plot2D(eigRes, 0, [2,3], ykey=['max(eigen.real)',\
                                          'min(eigen.real)'], fmt='w p')
pysces.plt.xrng(0.1,100000)
pysces.plt.logx()
pysces.plt.gridoff()
pysces.plt.xlabel('P')
raw_input("\n\tPress <enter> to continue\n")

pysces.plt.plot2D(eigRes, 0, [1], ykey=['Flux'], fmt='w p', log=1)
scipy.gplt.hold('on')
pysces.plt.plot2D(unstableRes, 0, [1], ykey=['Flux (unstable)'],\
                  fmt='w p', log=1)
scipy.gplt.hold('off')
pysces.plt.xrng(0.1,100000)
pysces.plt.yrng(0.01,120)
pysces.plt.gridoff()
pysces.plt.xlabel('P'); pysces.plt.ylabel('Flux')
raw_input("\n\tPress <enter> to continue\n")
```

11.6 Simulation of a system with two stable states

```
import scipy, pysces
import sys

mod = pysces.model('isola_thesis')

mod.doLoad()
mod.SetQuiet()

#Set the value of the solution you are looking for
mod.pitcon_targ_val = 2.3

#Set the Fortran index in array [s_vec,P]
mod.pitcon_targ_val_idx = 3

mod.pitcon_iter = 10

# Define an interval around the target value
low_i = mod.pitcon_targ_val - (mod.pitcon_targ_val/2.0)
high_i = mod.pitcon_targ_val + (mod.pitcon_targ_val/2.0)
mod.pitcon_par_space = scipy.linspace(low_i,high_i,20)

dump = mod.PITCON('P')

print "\nTarget points given as [P,A,B]"

for x in mod.pitcon_target_points:
    print x

print "\nMax Target point stored as [P,A,B]"
print max(mod.pitcon_target_points)

print "\nMin Target point stored as [P,A,B]"
print min(mod.pitcon_target_points)

mod.P = mod.pitcon_targ_val

# Initialize the system at double the maximum target
print "\nHigh initialization"
```

```
init_val = max(mod.pitcon_target_points)
mod.Ai = init_val[1]*2.0
mod.Bi = init_val[2]*2.0
print 'mod.Ai = '+ '%2.2f' % mod.Ai
print 'mod.Bi = '+ '%2.2f' % mod.Bi

mod.doSimPlot(5,50,'met')

# Initialize the system at half the minimum target
print "\nLow initialization"
init_val = min(mod.pitcon_target_points)
mod.Ai = init_val[1]/2.0
mod.Bi = init_val[2]/2.0
print 'mod.Ai = '+ '%2.2f' % mod.Ai
print 'mod.Bi = '+ '%2.2f' % mod.Bi

scipy.gplt.hold('on')
mod.doSimPlot(5,50,'met')
scipy.gplt.hold('off')

raw_input("\n\tPress <enter> to continue\n")
```

12 JWS Online appendix

12.1 A Python based model description file

```
#7
varName = ['Di', 'sr', 'um', 'Ks', 'xi', 'si', 'pi', ]

#7
varValue = ['0.1', '50', '0.4', '5', '0.01', '0', '0', ]

#5
expName = ['v1', 'v2', 'v3', 'v4', 'v5', ]

moiList = []

simTime = '150'
simStart = '0'
maxSteps = '20000'

metabName = ['p', 's', 'x', ]

fluxName = ['v1', 'v2', 'v3', 'v4', 'v5', ]

var = 7
exps = 5
metab = 3
fluxes = 5
smallVal = '10e-10'
appletNum = '333'
modelName = 'This demo generates an applet model pair'
```

12.2 An example jws generator session

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

c:\>python jws_java-input_gen.py

*****************************************************************
"jws_java_generator: autogenerate JWSApplet and JWSmodel code"
"Copyright (c) 2003-2004 Brett G. Olivier;"
"All Rights Reserved"
"Author: Brett G. Olivier"
*****************************************************************

Model/File Name = sample_program
ModNumber = 333
ModDescrip = This demo generates an applet model pair

Input checks for applet 333
var - len(varValue) should equal zero, value is: 0
len(varName) - len(varValue) should equal zero, value is: 0
metab - len(metabName) should equal zero, value is: 0
fluxes - len(fluxName) should equal zero, value is: 0
exps - len(expName) should equal zero, value is: 0
Basic checks passed file generation commencing ...

Changing directory to c://Python23//lib/site-packages//brettlib//
JWS_generator//M_out//333

Directory does not exist creating dir: 333

a333 applet autocode generation complete
Generated code written to a333.java
m333 model autocode generation complete
Generated code written to m333.java
a333 html autocode generation complete
Generated code written to h333.html
Update compile scripts (y/n):  y
        Updating: e_server_compile.bat
        Updating: e_neo_app_compile.bat
```

```
        Updating: e_JJJ_app_compile.bat
        Updating: e_vu_app_compile.bat
        Updating: e_vbi_app_compile.bat
        Updating: e_cal_app_compile.bat
        Updating: e_mobile_app_compile.bat
a333 batch autocode generation complete
Generated code written to t333.bat
```

12.3 JWS Online batch compiler files

12.3.1 extract from e_server_compile.bat

```
javac -d c:\jdk1.3\brett jwsm.java
echo Interface jwsm ..... done
javac -d c:\jdk1.3\brett JWSserver.java
echo NextGen JWSserver Granite ..... done

javac -d c:\jdk1.3\brett m101.java
echo m101 ..... done

javac -d c:\jdk1.3\brett m102.java
echo m102 ..... done
```

12.3.2 extract from e_JJJ_app_compile.bat

```
echo Compiling JWSApplet evolution applet for JJJ.BIOCHEM.SUN.AC.ZA

del c:\mytex\webdev\jjj.biochem.sun.ac.za\za\ac\sun\jwsapplet\*.class

echo Compiling a101 ...
javac -d c:\mytex\webdev\jjj.biochem.sun.ac.za\ sun\a101.java

echo Compiling a102 ...
javac -d c:\mytex\webdev\jjj.biochem.sun.ac.za\ sun\a102.java
```

12.3.3 extract from e_VU_app_compile.bat

```
echo Compiling JWSApplet evolution applet for www.jjj.bio.vu.nl

del c:\mytex\webdev\www.jjj.bio.vu.nl\za\ac\sun\jwsapplet\*.class
```

```
echo Compiling a101 ...
javac -d c:\mytex\webdev\www.jjj.bio.vu.nl\ vu\a101.java

echo Compiling a102 ...
javac -d c:\mytex\webdev\www.jjj.bio.vu.nl\ vu\a102.java
```

13 Bibliography

[1] Westerhoff, H. V. and Palsson, B. O. (2004). The Evolution of Molecular Biology Into Systems Biology. *Nature Biotechnology 22*, 1249–1252.

[2] Ideker, T., Galitski, T., and Hood, L. (2001). A New Approach to Decoding Life: Systems Biology. *Annu. Rev. Genomics Hum. Genet. 2*, 343–372.

[3] Kitano, H. (2002). Systems Biology: A Brief Overview. *Science 295*, 1662–1664.

[4] Wolkenhauer, O. and Klingmüller, U. (2004). Systems Biology: From a Buzzword to a Life Sciences Approach. *BIOforum Europe 04*, 22–23.

[5] Kitano, H. (2002). Computational Systems Biology. *Nature 420*, 206–210.

[6] Hofmeyr, J.-H. S. and Van der Merwe, K. J. (1986). METAMOD: Software for steady state modelling and control analysis of metabolic pathways on the BBC microcomputer. *Comput. Appl. Biosci. 2*, 243–249.

[7] Sauro, H. M. and Fell, D. A. (1991). SCAMP: a metabolic simulator and control analysis program. *Math. Comput. Modelling 15*, 15–28.

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

[9] Mendes, P. (1993). GEPASI: a software package for modelling the dynamics, steady states and control of biochemical and other systems. *Comput. Appl. Biosci. 9*, 563–571.

[10] Mendes, P. (1997). Biochemistry by numbers: simulation of biochemical pathways with Gepasi 3. *Trends Biochem. Sci. 9*, 361–363.

[11] Olivier, B. G. (1999). Designing a 4-Way Metabolic Junction. Master's thesis, University of Stellenbosch.

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

[13] Hofmeyr, J.-H. S., Olivier, B. G., and Rohwer, J. M. (2000). Putting the Cart Before the Horse: Designing a Metabolic System in Order to Understand It. In Cornish-Bowden, A. and Cárdenas, M. L., editors, *Technological and Medical Implications of Metabolic Control Analysis*, pp. 299–308. Kluwer, Dordrecht, Netherlands.

[14] Umbarger, H. and Brown, B. (1957). Threonine Deamination in Escherichia Coli. II. Evidence for Two L-Threonine Deaminases. *J. Bacteriol. 73*, 105–112.

[15] Yates, R. and Pardee, A. (1957). Control by Uracil of Formation of Enzymes Required for Orotate Synthesis. *J. Biol. Chem. 227*, 677–692.

[16] Umbarger, H. E. (1961). Feedback control by endproduct inhibition. *Cold Spring Harbor Symposia "Cellular Regulatory Mechanisms" 26*, 301–312.

[17] Hofmeyr, J.-H. S. and Cornish-Bowden, A. (1997). The Reversible Hill-Equation: How to Incorporate Cooperative Enzymes Into Metabolic Models. *Comput. Appl. Biosci. 13*, 377–385.

[18] Hofmeyr, J.-H. S., Olivier, B. G., and Rohwer, J. M. (2000). From Mushrooms to Isolas: Surprising Behaviour in a Simple Biosynthetic System Subject to End-Product Inhibition. In Hofmeyr, J.-H. S., Rohwer, J. M., and Snoep, J. L., editors, *Animating the Cellular Map*, pp. 199–205, Stellenbosch, South Africa. Stellenbosch University Press.

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

[20] Beazley, D. et al. (1995–). *SWIG: Simplified Wrapper and Interface Generator*. url: `http://www.swig.org/`.

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

[22] Jones, E., Oliphant, T., Peterson, P., et al. (2001–). *SciPy: Open Source Scientific Tools for Python*. url: `http://www.scipy.org/`.

[23] Hucka, M., Finney, A., Sauro, H., and Bolouri, H. (2001). The ERATO Systems Biology Workbench: Architectural Evolution. In Yi, T.-M., Hucka, M., Morohashi, M., and Kitano, H., editors, *Proceedings of the* 2^{nd} *International Conference on Systems Biology*, pp. 352–361. Omnipress, Madison,WI, USA.

[24] Poolman, M. G. (2002). *ScrumPy*. url: `http://bms-mudshark.brookes.ac.uk/ScrumPy/`.

[25] Olivier, B. G., Rohwer, J. M., and Hofmeyr, J.-H. S. (2002). Modelling Cellular Processes with Python and SciPy. *Mol. Biol. Rep. 29*, 249–254.

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

[27] Olivier, B. G. and Snoep, J. L. (2004). Web-Based Kinetic Modelling Using JWS Online. *Bioinformatics 20*, 2143–2144.

[28] Snoep, J. L. and Olivier, B. G. (2003). JWS Online Cellular Systems Modelling and Microbiology. *Microbiology 149*, 3045–3047.

[29] Snoep, J. L. and Olivier, B. G. (2002). Java Web Simulation (JWS): a Web Based Database of Kinetic Models. *Mol. Biol. Rep. 29*, 259–263.

[30] Garfinkel, D., Garfinkel, L., Pring, M., Green, S. B., and Chance, B. (1970). Computer applications to biochemical kinetics. *Annu. Rev. Biochem. 39*, 473–498.

[31] Heinrich, R., Rapoport, S. M., and Rapoport, T. A. (1977). Metabolic regulation and mathematical models. *Progr. Biophys. Molec. Biol. 32*, 1–82.

[32] Stucki, J. W. (1978). Stability analysis of biochemical systems - a practical guide. *Prog. Biophys. Molec. Biol. 33*, 99–187.

[33] Hess, B., Goldbeter, H., and Lefever, R. (1978). Temporal, spatial and functional order in regulated biochemical and cellular systems. *Adv. Chem. Phys. 38*, 363–413.

[34] Ottaway, J. H. (1979). Simulation of metabolic events. *Tech. Life Sci. Metab. Res. B2/11*, B219/1–25.

[35] Wright, B. E. and Kelly, P. J. (1981). Kinetic models of metabolism in intact cells, tissues and organisms. *Curr. Top. Cell. Regul. 19*, 103–158.

[36] Hess, B. (1983). Non-equilibrium dynamics of biochemical processes. *H.-S. Z. Physiol. Chem. 364*, 1–20.

[37] Massoud, T. F., Hademenos, G. J., Young, W. L., Gao, E., Pile-Spellman, J., and Vinuela, F. (1998). Principles and Philosophy of Modeling in Biomedical Research. *FASEB J. 12*, 275–285.

[38] Wiechert, W. (2002). Modelling and Simulation: Tools for Metabolic Engineering. *J. Biotechnol. 94*, 37–63.

[39] Rossler, O. E. and Hoffman, D. (1972). Repetitive hard bifurcation in a homogeneous reaction system. In Hemker, H. C. and Hess, B., editors, *Analysis and Simulation of Biochemical Systems*, volume 25, pp. 91–101. FEBS VII Meeting Proceedings.

[40] Prigogine, I. (1977). *Self organisation in non-equilibrium systems.* Wiley-Interscience.

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

[42] Savageau, M. A. (1976). *Biochemical systems analysis.* Addison-Wesley, London.

[43] Segel, L. A. (1981). *Mathematical models in cellular and molecular biology.* Cambridge University Press, Cambridge.

[44] Blum, J. J. and Stein, R. B. (1982). On the analysis of metabolic networks. In Goldberger, R. F. and Yamamoto, K. R., editors, *Biological Regulation and Development*, volume 3A, pp. 99–125. Plenum Press, New York.

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

[46] Westerhoff, H. V. and Dam, K. V. (1987). *Mosaic non-equilibrium thermodynamics and the control of biological free energy transduction*. Elsevier, Amsterdam.

[47] King, E. L. and Altman, C. (1956). A schematic method of deriving the rate laws for enzyme-catalyzed reactions. *J. Amer. Chem. Soc. 60*, 1375–1381.

[48] Cornish-Bowden, A. (2004). *Fundamentals of Enzyme Kinetics (3rd Edn.)*. Portland Press, London.

[49] Rhoads, D. G., Achs, M. J., Peterson, L., and Garfinkel, D. (1968). A method of calculating time-course behaviour of multi-enzyme systems from the enzymatic rate equations. *Comp. Biomed. Res. 2*, 45–50.

[50] Burns, J. A. (1969). Steady states of general multi-enzyme networks and their associated properties. Computational approaches. *FEBS Lett. 2 (Supp)*, S30–S33.

[51] Garfinkel, D. and Hess, B. (1964). Metabolic control mechanisms. VII. A detailed computer model of the glycolytic pathway in ascites cells. *J. Biol. Chem. 239*, 971–981.

[52] Garfinkel, D. (1969). Construction of biochemical computer models. *FEBS Lett. 2 (Supp)*, S9–S13.

[53] Chance, B., Garfinkel, D., Higgins, J., and Hess, B. (1960). Metabolic control mechanisms. *J. Biol. Chem. 235*, 2426–2439.

[54] Chance, E. M. and Curtis, A. R. (1970). Fast numerical simulation of biochemical systems. *FEBS Lett. 7*, 47–50.

[55] Chance, B. (1960). Analogue and digital representations of enzyme kinetics. *J. Biol. Chem. 235*, 2440–2443.

[56] Chance, B. (1961). Control characteristics of enzyme systems. *Cold Spring Harb. Sym. 26*, 289–299.

[57] Kacser, H. and Burns, J. A. (1973). The control of flux. *Symp. Soc. Exp. Biol. 32*, 65–104.

[58] Kacser, H. and Burns, J. A. (1979). Molecular democracy: who shares the controls? *Biochem. Soc. Trans. 7*, 1149–1160.

[59] Kacser, H. (1983). The control of enzyme systems *in vivo*: elasticity analysis of the steady state. *Biochem. Soc. Trans. 11*, 35–40.

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

[61] 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.

[62] Rapoport, T. A. and Heinrich, R. (1975). Mathematical analysis of multi-enzyme systems. 1. Modelling of the glycolysis of human erythrocytes. *BioSystems 7*, 120–129.

[63] Heinrich, R. and Rapoport, S. M. (1983). The utility of mathematical models for the understanding of metabolic systems. *Biochem. Soc. Trans. 11*, 31–35.

[64] Savageau, M. A. (1969). Biochemical systems analysis. 1. Some mathematical properties of the rate law for the component enzymatic reactions. *J. Theor. Biol. 25*, 365–369.

[65] Savageau, M. A. (1969). Biochemical systems analysis. 2. The steady-state solutions for an n-pool system using a power-law approximation. *J. Theor. Biol. 25*, 370–379.

[66] Savageau, M. A. (1970). Biochemical systems analysis. 3. Dynamic solutions using a power-law approximation. *J. Theor. Biol. 26*, 215–226.

[67] Fell, D. (1996). *Understanding the control of metabolism.* Portland Press, London.

[68] Cornish-Bowden, A., editor (1999). *Metabolic Control Design: Implications and Applications.* Kluwer Academic Publishers, Dordrecht.

[69] Hofmeyr, J.-H. S., Kacser, H., and Van der Merwe, K. J. (1986). Metabolic control analysis of moiety-conserved cycles. *Eur. J. Biochem. 155*, 631–641.

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

[71] Cascante, M., Puigjaner, J., and Kholodenko, B. (1996). Steady-state characterization of systems with moiety-conservations made easy: Matrix equations of metabolic control analysis and biochemical system theory. *J. Theor. Biol. 178*, 1–6.

[72] Fell, D. A. and Sauro, H. M. (1985). Metabolic control and its analysis: additional relationships between elasticities and control coefficients. *Eur. J. Biochem. 148*, 555–561.

[73] Hofmeyr, J.-H. S. (1989). Control-pattern analysis of metabolic pathways: Flux and concentration control in linear pathways. *Eur. J. Biochem. 186*, 343–354.

[74] Kholodenko, B. N. and Erlikh, L. I. (1989). Theory of the regulation of metabolism: a complete system of equations for the regulation coefficients. *Biophysics 34*, 802–807.

[75] Hofmeyr, J.-H. S. and Cornish-Bowden, A. (1991). Quantitative assessment of regulation in metabolic systems. *Eur. J. Biochem. 200.*

[76] 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.

[77] Hofmeyr, J.-H. S. and Westerhoff, H. V. (2001). Building the cellular puzzle: control in multi-level reaction networks. *J. Theor. Biol. 208*, 261–285.

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

[79] Small, J. R. and Fell, D. A. (1989). The matrix method of metabolic control analysis: its validity for complex pathway structures. *J. Theor. Biol. 136*, 181–197.

[80] 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.

[81] Fell, D. A., Sauro, H. M., and Small, J. R. (1990). Control coefficients and the matrix method. In Cornish-Bowden, A. and Cárdenas, M. L., editors, *Control of Metabolic Processes*, pp. 139–148, New York. Plenum Press.

[82] 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.

[83] Ainscow, E. K. and Brand, M. D. (1995). Top-down control analysis of systems with more than one common intermediate. *Eur. J. Biochem. 231*, 579–586.

[84] Schuster, S., Kahn, D., and Westerhoff, H. V. (1993). Modular analysis of the control of complex metabolic pathways. *Biophys. Chem. 48*, 1–17.

[85] Gutfreund, H. (1971). Transient and relaxation kinetics of enzyme reactions. *Annu. Rev. Biochem. 40*, 315–344.

[86] Curtis, A. R. (1976). FACSIMILE - a computer program for simulation and optimization. *Biochem. Soc. Trans. 4*, 364–371.

[87] Roman, G.-C. and Garfinkel, D. (1978). BIOSSIM - a structured machine-independent biological simulation language. *Comp. Biomed. Res. 11*, 3–15.

[88] Kootsey, J. M., Kohn, M. C., Feezor, M. D., Mitchell, G. R., and Fletcher, P. R. (1986). SCoP: An Interactive Simulation Control Program for Micro and Minicomputers. *Bull. Math. Biol. 48*, 427–441.

[89] Holzhütter, H. G. and Colosimo, A. (1990). SIMFIT: A Microcomputer Software-Toolkit for Modelistic Studies in Biochemistry. *CABIOS 6*, 23–28.

[90] Letellier, T., Reder, C., and Mazat, J.-P. (1991). CONTROL - Software for the Analysis of Control of Metabolic Networks. *CABIOS 7*, 383–390.

[91] Cornish-Bowden, A. and Hofmeyr, J.-H. S. (1991). A program for the modelling and control analysis of metabolic pathways on the IBM PC and compatibles. *Comput. Appl. Biosci. 7*, 89–93.

[92] 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.

[93] Downey, A., Elkner, J., and Meyers, C. (2002). *How to Think Like a Computer Scientist*. Green Tea Press, Wellesley, USA.

[94] Harms, D. and McDonald, K. (1999). *The Quick Python Book*. Manning Publications, Greenwich.

[95] Lutz, M. and Ascher, D. (1999). *Learning Python*. O'Reilly, Sebastopol, USA.

[96] Martelli, A. (2003). *Python in a Nutshell*. O'Reilly, Sebastopol, USA.

[97] Holden, S. (2002). *Python Web Programming*. New Riders, Indianapolis, USA.

[98] Lundh, F. (2001). *Python Standard Library*. O'Reilly, Sebastopol, USA.

[99] Martelli, A. and Ascher, D. (2002). *Python Cookbook*. O'Reilly, Sebastopol, USA.

[100] Pilgrim, M. (2004). *Dive Into Python*. url: `http://diveintopython.org/`.

[101] Eckel, B. (2002). *Thinking in Python: Design Patterns and Problem-Solving Techniques*. url: `http://www.mindview.net/Books/TIPython/`.

[102] van Rossum, G. (1999). *Computer Programming for Everybody*. url: http://www.python.org/doc/essays/cp4e.html.

[103] Kuchling, A. M. (2004). *What's New in Python 2.3*. url: http://www.python.org/doc/2.3.4/.

[104] van Rossum, G. (1998). *Glue It All Together With Python*. url: http://www.python.org/doc/essays/omg-darpa-mcc-position.html.

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

[106] Oliphant, T. (2004). *SciPy Tutorial*. url: http://www.scipy.org/documentation/tutorial.pdf.

[107] Golub, G. H. and Loan, C. F. V. (1996). *Matrix Computations*. Johns Hopkins University Press, Baltimore, USA.

[108] Watkins, D. S. (1991). *Fundamentals of Matrix Computations*. Wiley, New York, USA.

[109] O'Malley, R. E. (1997). *Thinking About Ordinary Differential Equations*. Cambridge University Press, Cambridge, England.

[110] Lambert, J. D. (1974). *Computational Methods in Ordinary Differential Equations*. Wiley, London, England.

[111] Harel, D. (1987). *Algorithmics: The Spirit of Computing*. Addison-Wesley, Wokingham, England.

[112] Hultquist, P. F. (1988). *Numerical Methods for Engineers and Computer Scientists*. Benjamin-Cummings, Menlo Park, USA.

[113] Anderson, E. (1995). *LAPACK User's Guide*. Siam, Philadelphia, USA.

[114] van Rossum, G. et al. (1991–). *The Python programming language*. url: http://www.python.org/.

[115] Goryanin, I., Hodgman, T. C., and Selkov, E. (1999). Mathematical Simulation and Analysis of Cellular Metabolism and Regulation. *Bioinformatics 15*, 749–758.

[116] Fields, S. (2001). The Interplay of Biology and Technology. *Proc. Natl. Acad. Sci. 98*, 10051–10054.

[117] Cascante, M., Boros, L. G., Comin-Anduix, B., de Atauri, P., Centelles, J. J., and Lee, P. W.-N. (2002). Metabolic Control Analysis in Drug Discovery and Disease. *Nat. Biotechnol. 20*, 243–249.

[118] Doyle, J. (2001). Beyond the Spherical Cow. *Nature 411*, 151–152.

[119] Palsson, B. O. (2004). In silico Biotechnology Era of Reconstruction and Interrogation. *Curr. Opin. Biotech. 15*, 50–51.

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

[121] Hucka, M., A. Finney, H. M. S., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J.-H. S., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Novere, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E. D., Nakayama, Y., Nelson, M. R., Nielsen, P. F., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., and Wang, J. (2003). The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models. *Bioinformatics 19*, 524–531.

[122] Kubíček, M. (1976). Algorithm 502: Dependence of Solution of Nonlinear Systems on a Parameter [C5]. *ACM Transactions on Mathematical Software 2*, 98–107.

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

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

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

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

[127] 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.

[128] Strang, G. (1993). *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA.

[129] 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.

[130] 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.

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

[132] 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.

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

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

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

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

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

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

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

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

[141] 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.

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

[143] 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.

[144] 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.

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

[146] 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.

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

[148] 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.

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

[150] 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.

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

[152] 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).

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

[154] 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.

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

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

[157] 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.

[158] 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.

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

[160] Poolman, M. G., Fell, D. A., and Thomas, S. (2000). Modelling photosynthesis and its control. *J. Exp. Bot. 51*, 319–28.

[161] 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.

[162] Atkinson, D. E. (1965). Biological feedback control at the molecular level. *Science 150*, 851–857.

[163] Tyson, J. J. and Othmer, H. G. (1978). The dynamics of feedback control circuits in biochemical pathways. *Progr. Theor. Biol. 5*, 1–62.

[164] Easterby, J. S. (1986). The effect of feedback on pathway transient response. *Biochem. J. 233*, 871–875.

[165] Hofmeyr, J. and Olivier, B. (2002). The regulatory design of an allosteric feedback loop: the effect of saturation by pathway substrate. *Biochem. Soc. T. 30*, 19–25.

[166] Palsson, B. O. and Lightfoot, E. N. (1985). Mathematical Modelling of Dynamics and Control in Metabolic Networks. IV. Local Stability Analysis of Single Biochemical Control Loops. *J. Theor. Biol. 113*, 261–277.

[167] Palsson, B. O. and Lightfoot, E. N. (1985). Mathematical Modelling of Dynamics and Control in Metabolic Networks. V. Static Bifurcations in Single Biochemical Control Loops. *J. Theor. Biol. 113*, 279–298.

[168] Palsson, B. O. and Groshans, T. M. (1988). Mathematical Modelling of Dynamics and Control in Metabolic Networks. VI. Dynamic Bifurcations in Single Biochemical Control Loops. *J. Theor. Biol. 131*, 43–53.

[169] Zeeman, E. C. (1977). *Catastrophe Theory.* Addison-Wesley, London.

[170] Woodcock, A. and Davis, M. (1978). *Catastrophe Theory.* E. P. Dutton, New York.

[171] Okninski, A. (1992). *Catastrophe Theory.* Comprehensive Chemical Kinetics. Elsevier, Amsterdam.

[172] Westerhoff, H. W. (2001). *The Silicon Cell (SiC!).* url: `http://www.siliconcell.net/`.

[173] Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., Rapp, B. A., and Wheeler, D. L. (2002). GenBank. *Nucleic Acids Res. 30*, 17–20.

[174] Stoesser, G., Baker, W., Broek, A. V. D., Kanz, M. G.-P. A., Kulikova, T., Leinonen, R., Lin, Q., Lombard, V., Lopez, R., Mancuso, R., Nardone, F., Stoehr, P., Tuli, M. A., Tzouvara, K., and Vaughan, R. (2003). The EMBL Nucleotide Sequence Database: Major New Developments. *Nucleic Acids Res. 31*, 17–22.

[175] Boeckmann, B., Bairoch, A., Apweiler, R., Estreicher, M.-C. B. A., Gasteiger, E., Martin, M. J., ODonovan, K. M. A., Phan, I., Pilbout, S., and Schneider, M. (2003). The SWISS-PROT Protein Knowledgebase and its Supplement TrEMBL in 2003. *Nucleic Acids Res. 31*, 365–370.

[176] Schomburg, I., Chang, A., Hofmann, O., Ebeling, C., Ehrentreich, F., and Schomburg, D. (2002). BRENDA: A Resource for Enzyme Data and Metabolic information. *Trends Biochem. Sci. 27*, 54–56.

[177] Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Venter, J., and Hutchison, C. (1999). E-Cell : Software Environment for Whole Cell Simulation. *Bioinformatics 192*, 72–84.

[178] Loew, L. M. and Schaff, J. C. (2001). The Virtual Cell: A Software Environment for Computational Cell Biology. *Trends Biotechnol. 19*, 401–409.

[179] Curien, G., Ravanel, S., and Dumas, R. (2003). Metabolic Engineering of Lactic Acid Bacteria, the Combined Approach: Kinetic Modelling, Metabolic Control and Experimental Analysis. *Eur. J. Biochem. 270*, 1–13.

[180] Hoefnagel, M. H. N., Starrenburg, M. J. C., Martens, D. E., Hugenholtz, J., Kleerebezem, M., Swam, I. I. V., Bongers, R., Westerhoff, H. V., and Snoep., J. L. (2002). Metabolic Engineering of Lactic Acid Bacteria, the Combined Approach: Kinetic Modelling, Metabolic Control and Experimental Analysis. *Microbiology 148*, 1003–1013.

[181] Wolf, J., Passarge, J., Somsen, O. J. G., Snoep, J. L., Heinrich, R., and Westerhoff, H. V. (2000). Transduction of Intracellular and Intercellular Dynamics in Yeast Glycolytic Oscillations. *Biophys. J. 78*, 1145–1153.

[182] Teusink, B., Passarge, J., Reijenga, C. A., Esgalhado, E., Van der Weijden, C. C., Schepper, M., Walsh, M. C., Bakker, B. M., Van Dam, K., Westerhoff, H. V., and Snoep, J. L. (2000). Can Yeast Glycolysis Be Understood in Terms of in Vitro Kinetics of the Constituent Enzymes? Testing Biochemistry. *Eur. J. Biochem. 267*, 5313–5329.

[183] Cronwright, G. R., Rohwer, J. M., and Prior, B. A. (2003). Metabolic Control Analysis of Glycerol Synthesis in *Saccharomyces cerevisiae*. *Appl. Environ. Microb. 68*, 4448–4456.

[184] Rohwer, J. M., Meadow, N. D., Roseman, S., Westerhoff, H. V., and Postma, P. W. (2000). Kinetic Model: Bacterial Phosphotransferase System. *J. Biol. Chem. 275*, 34909–34921.

[185] Bhartiya, S., Rawool, S., and Venkatesh, K. V. (2003). Dynamic Model of Escherichia Coli Tryptophan Operon Shows an Optimal Structural Design. *Eur. J. Biochem. 270*, 2644–2651.

[186] Tyson, J. J. and Novak, B. (2001). Regulation of the Eukaryotic Cell Cycle: Molecular Antagonism, Hysteresis and Irreversible Transitions. *J. Theor. Biol. 210*, 249–263.

[187] Kholodenko, B. N., Demin, O. V., Moehren, G., and Hoek, J. B. (1999). Quantification of Short Term Signaling by the Epidermal Growth Factor Receptor. *J. Biol. Chem. 274*, 30169–30181.

[188] Hoefnagel, M. H. N., Hugenholtz, J., and Snoep, J. L. (2002). Time dependent responses of glycolytic intermediates in a detailed glycolytic model of *Lactococcus Lactis* during glucose run-out experiments. *Mol. Biol. Rep. 29*, 157–161.

[189] Friedman-Hill, E. J. (2001). *Java: Your Visual Blueprint for Building Portable Java Programs.* Hungry Minds Inc., New York.

[190] Gillespie, D. (1977). Exact Stochastic Simulation of Coupled Chemical Reactions. *J. Phys. Chem. 81*, 2340–2361.

[191] Gibson, M. and Bruck, J. (2000). Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J. Phys. Chem. A. 104*, 1876–1889.

[192] Barrett, Berry, Chan, Demmel, Donato, Dongarra, Eijkhout, Pozo, Romine, and der Vorst, V. (1993). *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Publications.

[193] Thomas, L. (1992). *The Fragile Species.* Maxwell Macmillan, New York.

14 Publications