PySUNDIALS: Providing python bindings to a robust suite of mathematical tools for computational systems biology

by

James Gilmour Dominy

Thesis presented in partial fulfilment of the requirements for the degree of

Master of Science

at the University of Stellenbosch

Study leaders:

Prof J.-H.S. Hofmeyr (supervisor) Prof J. M. Rohwer (co-supervisor)

March 2009

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Abstract

A Python package called PySUNDIALS has been developed which provides an interface to the suite of nonlinear differential/algebraic equation solvers (SUNDIALS) using ctypes as a foreign function interface (FFI). SUNDIALS is a C implementation of a set of modern algorithms for integrating and solving various forms of the initial value problem (IVP). Additionally, arbitrary root finding capabilities, time dependent sensitivity analysis, and the solution of differential and algebraic systems are available in the various modules provided by SUNDIALS. A significant focus of the project was to ensure the python package conforms to Python language standards and syntactic expectations.

Multiple examples of the SUNDIALS modules (CVODE, CVODES, IDA and KINSOL) are presented comparing PySUNDIALS to C SUNDIALS (for verification of correctness), and comparing PySUNDIALS to various other comparable software packages. The examples presented also provide benchmark comparisons for speed, and code length. Specific uses of the features of the SUNDIALS package are illustrated, including the modelling of discontinuous events using root finding, time dependent sensitivity analysis of oscillatory systems, and the modelling of equilibrium blocks using a complete set of implicit differential and algebraic equations.

PySUNDIALS is available as open source software for download. It is being integrated into the systems biology software PySCeS as an optional solver set, on an ongoing basis. A brief discussion of potential methods of optimization and the continuation of the project to wrap the parallel processing modules of SUNDIALS is presented.

Opsomming

'n Python pakket genaamd PySUNDIALS is ontwikkel om 'n koppelfase te verskaf na die SUNDIALS stel van oplossers vir stelsels van nie-lineêre differensiaal/algebraïese vergelykings; hierdie pakket maak gebruik van ctypes as vreemde funksie koppelfase (FFI). SUNDIALS is a C implementering van 'n stel moderne algoritmes vir die integrasie en oplos van verskillende vorms van beginwaarde probleme (IVP). Aanvullend daartoe is die volgende beskikbaar in die verskeie modules van SUNDIALS: arbitrêre vind van wortels en tyd-afhanklike sensitiwiteitsanalise. 'n Belangrike fokus van die projek was om te verseker dat die pakket voldoen aan die taalstandaarde en sintaktiese vereistes van Python.

Veelvoudige voorbeelde van die SUNDIALS modules (CVODE, CVODES, IDA en KINSOL) word verskaf ter vergelyking van PySUNDIALS en C SUNDIALS (vir verifiëring van korrektheid) en ook om PySUNDIALS te vergelyk met verskeie ander sagteware pakkette. Die voorbeelde verskaf ook beginpuntvergelykings ten opsigte van spoed en lengte van kode. Spesifieke toepassings van die SUNDIALS pakket word geïllustreer, soos die modellering van diskontinuïteite deur gebruik van die vind van wortels, tyd-afhanklike sensitiwiteitsanalise van ossilerende sisteme, en die modellering van ewewigsblokke deur gebruik van implisiete algebraïese en differensiaalvergelykings.

PySUNDIALS is beskikbaar as oopbronsagteware. Dit word tans geïntegreer in die sisteembiologie sagtewarepakket PySCeS as 'n opsionele stel van oplossers. Die potensiële metodes vir optimisering van PySUNDIALS en die inkorporering van die parallele modules van SUNDIALS word kortliks bespreek.

Contents

List of Figures

List of Tables

Listings

Stylistic Conventions

This document maintains a number stylistic conventions to increase clarity and legibility.

- Names of software packages, or parts thereof, are presented in sans-serif, e.g. "PySUNDIALS".
- When variable names, type names, or small snippets of code are referred to in text they are presented in mono spaced font, e.g. "`import pysces`", or "when `a` is set to".
- URL's are presented in mono spaced font always beginning with a protocol specification, e.g. `http://www.google.com` or `svn://pysundials.sourceforge.net`

Acknowledgements

- Prof. J.H.S. Hofmeyr for supervision
- Prof. J. Rohwer for co-supervision
- Dr. B. G. Olivier for assistance with PySCeS and the provision of PySCeS model files
- Andrew Dalke for in depth technical help regarding Python and ctypes internals
- The National Bioinformatics Network (NBN) for funding

Chapter 1

Background

1.1 Systems biology models and nonlinear dynamics

"Systems biology" is a poorly defined term, loosely describing the interdisciplinary study of whole biological systems. Although there is no consensus on the term's precise meaning, Wikipedia (2008) lists multiple general applications of the term as identified within the scope of the biological sciences. The first is the study of interactions between biological components, such as enzymes and metabolites in metabolic pathways, and how these interactions form emergent systemic behaviours (Snoep and Westerhoff, 2005; Hofmeyr, 2007).

The second usage defines systems biology to be the opposite of traditionally reductionist scientific approaches, which have been used to successfully isolate and characterise components of biological systems, but yield little knowledge about the behaviour of the system as a whole. Systems biology in this sense, refers to the integration of knowledge acquired from reductionist methods, using equally scientifically rigorous methods, to form knowledge about the system as whole. (Sauer et al., 2007; Noble, 2006)

Thirdly, the term refers to those techniques and methods that treat whole systems experimentally. Given the objective of modelling interactions in a biological system, experimental techniques that collect large amounts of quantitative data about a system as a whole, such as those used in high-throughput biology, are differentiated from those techniques dealing with an individual component of the system. (Kholodenko et al., 2005)

For the purposes of this text we consider the first case. More specifically, we understand "systems biology" to mean the investigation of the nature and properties of intracellular reaction and interaction networks by the creation and manipulation of mathematical models thereof.

1.1.1 The properties of metabolic pathways

To understand how such models are structured, let us first consider the general biological properties of metabolic networks. Metabolic networks consist of a network of chemical reactions that interact by means of shared metabolite pools, such that one reaction's product is the substrate of another reaction. Reactions may interact via other mechanisms, in the form of regulatory loops. Such feedback or feed-forward loops are due to binding of effectors to enzymes, and are therefore not due to mass action. Metabolic networks bear large similarities to electrical or hydrodynamic networks, which are also models of flow through a system. However, chemical, and hence biochemical, networks have the unique structural property of stoichiometry, being the fixed ratios in which metabolites react with one another for a given reaction. The stoichiometry of a system fundamentally influences the distribution of flow through the system in ways not found in other flow networks.

The simplest situation of biochemical relevance is where one reaction is coupled to another when its product is used as the only substrate of another reaction (See Figure 1.1a). Such couplings can be joined together to form linear chains of reactions which represent the conversion of one metabolite to another through a number of intermediates.

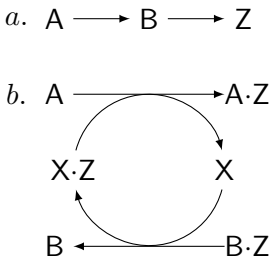

Figure 1.1: *Two mechanisms for coupling reactions.*

Individual reactions need not necessarily be coupled in a 'linear' manner, as a single reaction may consume multiple substrates, and likewise might produce multiple products. Two such reactions can be coupled via multiple shared metabolite pools (See Figure 1.1b), each of which shares a common chemical moiety (X in the figure). The stoichiometry of the system dictates that the total mass of the molecules that share this common moiety remains invariant, forming a *moiety conserving subnetwork*.

A common biological example of this is phosphorylation of a compound X by ATP, producing ADP and the phosphorylated product (X·P) (see Figure 1.2). For the reaction to continue, ATP must be replenished by the phosphorylation of ADP to produce ATP, which would involve at least one additional

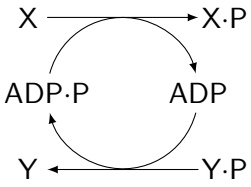

Figure 1.2: *An example of moiety conservation.*

substrate (Y·P) to provide the phosphate and produce one additional product (Y). The total amount of the moiety ADP, whether free or bound to an additional phosphate in the form of ATP, remains constant.

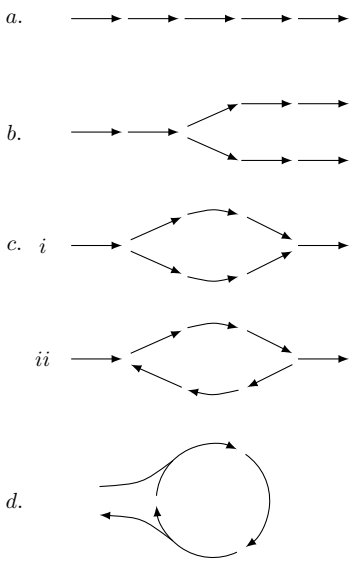

Figure 1.3: *The four structural motifs that comprise a metabolic network*: *a.* The linear chain, *b.* The branched chain, *c.* The loop, either parallel (*i*) or anti-parallel (*ii*), and *d.* The cycle.

Given these two fundamental methods of linking reactions together, we can now discuss higher level structural properties of metabolic networks. The most obvious structure is the linear pathway of reactions (Figure 1.3*a*), where two or more reactions are chained together by mutual metabolite pools.

Additionally, multiple reactions might consume the same substrate, forming a branch (Figure 1.3*b*) in the pathway. Similarly, multiple reactions might produce the same product, joining two initially separate pathways into one, allowing the formation of a parallel loop (Figure 1.3c_i). Anti-parallel loops (Figure 1.3c_{ii}) are formed when one side branch eventually produces a metabolite that is consumed by a reaction occurring before the branch point. Anti-

parallel loops differ from the final motif, the cycle (Figure 1.3*d*), in that their branch and join points are formed by multiple reactions either consuming or producing the same metabolite, respectively. Cycles, in contrast, have entry and exit points formed by reactions that produce or consume multiple metabolites.

It is important to remember that although we visualise a pathway as a series of arrows and boxes, implying a spatial relationship between reactions, that no such spatial relationship exists. A reaction that exists within a given physically bounded compartment occurs throughout that compartment. In cases where diffusion processes are slow compared to the elements of interest in the system, we cannot assume that a given metabolite is uniformly distributed throughout the volume, and we must model their diffusion explicitly, as in Rudiger et al. (2007) and Boderke et al. (2000). For the most part though, diffusion is rapid compared to metabolite production, and one can assume that metabolites are distributed evenly throughout any compartment in which they are present, a situation analogous to the engineering concept of a "well stirred reactor".

1.1.1.1 Equilibrium state

Chemical networks as a whole can be in three states. The first is a *state of equilibrium*, where the individual chemical pools have values that do not vary over time because each reaction is in chemical equilibrium, i.e. the net rate of each reaction is zero. Such systems can only be thermodynamically closed, meaning there is no transfer of mass into or out of the system. These systems are of almost no biological relevance, except as useful reference states, e.g., for judging how far a reaction system is out of equilibrium.

1.1.1.2 Steady state

Far more interesting is an open system through which a flow of mass occurs. Such systems tend to a final state in which the metabolite pools are time-invariant in size, but the reaction rates are nonzero; the system is said to be in *steady state*. The steady state itself is theoretically asymptotic, although in practice it is possible to treat a system as being in a steady state. Many systems do not achieve steady state, instead having metabolite concentrations that oscillate with constant period and amplitude and not settling to time-invariant values. Such metabolites are described as being in a "limit cycle".

A system may have a single steady state, which will be reached regardless of initial concentrations, or multiple steady states. The particular steady state achieved by systems with multiple steady states depends not only on the parameters of the system, such as maximal enzyme activities or equilibrium constants, but also on the initial concentrations of the system, i.e. its history. Steady states may be stable to various degrees, and in various ways. Struc-

tural stability refers to the fact that a particular system has a set of steady states that exists in a smooth continuum. Infinitesimal changes made to a parameter of the system alter the steady state of the system in a proportionally infinitesimal manner. Conversely, a structurally unstable steady state occurs at a discontinuity where a small change in a parameter will produce a finite jump to a new steady state; a typical example is a hysteretic system.

1.1.1.3 Transient state

A system can also be in a state of change or progression, such that its metabolite concentrations fluctuate and its reaction rates are nonzero. These systems are either described as being in the "transient state", as they are changing state over time. They may be progressing toward a steady state, either from a set of initial conditions, or after the perturbation of a steady state. They might be between steady states after a change in a system parameter, or they may simply be chaotic.

1.1.2 Mathematical models of metabolic networks

A full kinetic model of a reaction network is in actuality a hierarchical collection of smaller models. Each individual reaction is its own model. We model the coupling of reactions via shared metabolites with a higher level model.

1.1.2.1 Modelling an individual reaction

At the lowest level, we have mathematical models representing individual reactions. Various methods of modelling a single reaction exist. We can use the first principles methods of mass action modelling, in which a reaction is broken down into elementary chemical steps. Consider the reaction

$$A + B \rightleftharpoons C \tag{1.1}$$

The biological nature of the systems we investigate means the majority of the reactions being modelled are enzyme catalysed. If we assume this reaction is enzyme catalysed by some enzyme (E) using a random binding mechanism, then we can break the reaction down into explicit elementary chemical steps of association, isomerisation, and dissociation (Equations 1.2–1.5, 1.6, and 1.7 respectively).

$$E + A \overset{v_1}{\rightleftharpoons} E \cdot A \tag{1.2}$$

$$E \cdot A + B \overset{v_2}{\rightleftharpoons} E \cdot A \cdot B \tag{1.3}$$

$$E + B \overset{v_3}{\rightleftharpoons} E \cdot B \tag{1.4}$$

$$E \cdot B + A \overset{v_4}{\rightleftharpoons} E \cdot A \cdot B \tag{1.5}$$

$$E \cdot A \cdot B \overset{v_5}{\rightleftharpoons} E \cdot C \tag{1.6}$$

$$E \cdot C \overset{v_6}{\rightleftharpoons} E + C \tag{1.7}$$

These chemical equations are very simple to model mathematically. Each of these chemical equations represents a reversible reaction, and the mathematical expression of the reaction rate thus requires two terms, one for each direction of the reaction. Each term expresses the rate of the reaction in a particular direction in terms of concentrations and mass action constants. The combining of both directions into a single equation may lead to a negative value, which necessitates the definition of a particular direction of the reaction as forward, i.e. the positive direction.

$$v_1 = k_{1f}[E][A] - k_{1r}[E \cdot A] \tag{1.8}$$

$$v_2 = k_{2f}[E \cdot A][B] - k_{2r}[E \cdot A \cdot B] \tag{1.9}$$

$$v_3 = k_{3f}[E][B] - k_{3r}[E \cdot B] \tag{1.10}$$

$$v_4 = k_{4f}[E \cdot B][A] - k_{4r}[E \cdot A \cdot B] \tag{1.11}$$

$$v_5 = k_{5f}[E \cdot A \cdot B] - k_{5r}[E \cdot C] \tag{1.12}$$

$$v_6 = k_{6f}[E \cdot C] - k_{6r}[E][C] \tag{1.13}$$

Using mass action kinetics leads to a model with very simple kinetics, but a large number of reactions, and hence a large number of ordinary differential equations (ODEs).

$$\frac{dA}{dt} = -v_1 - v_4 \tag{1.14}$$
$$\frac{dB}{dt} = -v_2 - v_3 \tag{1.15}$$
$$\frac{dEA}{dt} = v_1 - v_2 \tag{1.16}$$
$$\frac{dEB}{dt} = v_3 - v_4 \tag{1.17}$$
$$\frac{dEAB}{dt} = v_2 + v_4 - v_5 \tag{1.18}$$
$$\frac{dEC}{dt} = v_5 - v_6 \tag{1.19}$$
$$\frac{dC}{dt} = v_6 \tag{1.20}$$
$$\frac{dE}{dt} = v_6 - v_1 - v_3 \tag{1.21}$$

If we assume a steady state, and set these ODEs to zero, we can derive a steady state rate law for the individual enzyme catalysed reaction. Thus, the second method of modelling an individual reaction is introduced, namely the rate law. The rate law method provides a satisfactory aggregate model of each of the mass action step involved in a reaction, as a single steady-state rate law. Rate laws, or rate equations, are nonlinear functions of substrate, product and effector concentrations, which are system variables, and constants, such as maximal enzyme activity ($V_{max} = k_{cat}[E]$), kinetic constants (K_M, K_i, etc), or equilibrium constants (K_{eq}). Constants are either fundamental rate constants as used in the King-Altman form (King and Altman, 1956), or statistically derived phenomenological constants as used in, for example, the Cleland form (Cleland, 1967).

A variety of rate equations exist to model the kinetic behaviours exhibited by various enzymes, including the Michaelis-Menten equation (Equation 1.22) (Michaelis and Menten, 1913), its reversible form (Equation 1.23), the reversible Hill equation (Equation 1.24) (Hofmeyr and Cornish-Bowden, 1997; Hanekom, 2006) and the Monod-Wyman-Changeux equation (Equation 1.25) (Monod et al., 1965).

$$v = \frac{\frac{V_f}{K_A K_B}[A][B]}{1 + \frac{[A]}{K_A} + \frac{[B]}{K_B} + \frac{[A][B]}{K_A K_B}} \tag{1.22}$$

where V_f is the forward limiting velocity and K_A, K_B and K_C are the Michaelis

constants.

$$v = \frac{\frac{V_f}{K_A K_B}\left([A][B] - \frac{[C]}{K_{eq}}\right)}{1 + \frac{[A]}{K_A} + \frac{[B]}{K_B} + \frac{[A][B]}{K_A K_B} + \frac{[C]}{K_C}} \tag{1.23}$$

$$v = \frac{\frac{V_f}{A_{0.5}B_{0.5}}\left([A][B] - \frac{[C]}{K_{eq}}\right)\left(\frac{[A]}{A_{0.5}}\frac{[B]}{B_{0.5}} + \frac{[C]}{C_{0.5}}\right)^{h-1}}{1 + \left(\frac{[A]}{A_{0.5}} + \frac{[C]}{C_{0.5}}\right)^{h} + \left(\frac{[B]}{B_{0.5}} + \frac{[C]}{C_{0.5}}\right)^{h} + \left(\frac{[A]}{A_{0.5}}\frac{[B]}{B_{0.5}} + \frac{[C]}{C_{0.5}}\right)^{h} - 2\left(\frac{[C]}{C_{0.5}}\right)^{h}} \tag{1.24}$$

where $A_{0.5}$, $B_{0.5}$ and $C_{0.5}$ are half-saturation concentrations and h is the Hill coefficient.

$$v = \frac{\frac{V_f}{K_A K_B}\left([A][B] - \frac{[C]}{K_{eq}}\right)\left(1 + \frac{[A]}{K_A} + \frac{[B]}{K_B} + \frac{[A][B]}{K_A K_B} + \frac{[C]}{K_C}\right)^{n-1}}{\left(1 + \frac{[A]}{K_A} + \frac{[B]}{K_B} + \frac{[A][B]}{K_A K_B} + \frac{[C]}{K_C}\right)^{n} + L_0} \tag{1.25}$$

where $L_0 = T_0/R_0$, K_A, K_B and K_C are the intrinsic binding constants and n is the number of subunits.

The validity of an overall model relies on appropriate rate equations being chosen for each reaction represented in the model. The reversible Michaelis-Menten equation is an appropriate choice for reactions displaying non-cooperative binding, but for reactions displaying cooperative binding, the Monod-Wyman-Changeux or reversible Hill equations are appropriate choices.

Another possible method, developed by Savageau (1969) and known as the power-law method, aggregates even further, combining all reactions that produce a particular metabolite into a single power-law rate equation, and all reactions that consume that same metabolite into another power-law rate equation. This ensures that the differential equation expressing change in concentration over time for a single metabolite will never contain more than two terms. This method makes the model mathematically more tractable, allowing for an analytical solution of the steady-state equations, but has the disadvantage that it bears little discernible relationship to the original biological system.

For our purposes, the rate law method provides a middle ground both sufficiently tractable and whose relationships with biological and chemical mechanisms are comparatively easy to infer. Given models of individual reactions, we must now explore methods of linking these reaction models together. The change in time of a given metabolite amount can be represented by an ordinary differential equation, which is the summation of all rate equations in

which the metabolite appears as substrate or product, such that rate equations which model reactions that produce the metabolite are positive, and those rate equations that consume the metabolite are negated. Thus, for a given metabolite, its ODE represents the overall change in amount of that metabolite within the entire system over time. In the case of a moiety conservation, one metabolite's system variable is defined in terms of its partner or partners, not requiring an ODE in and of itself, but rather a simple algebraic equation describing the conservation relationship independent of time. As the total amount of the shared component is conserved, we can mathematically express the conservation relationship depicted in Figure 1.2 as

$$c = ATP + ADP \tag{1.26}$$

where c is a constant equal to the sum of the initial values of ATP and ADP.

1.1.2.2 A worked example

A description of the complete kinetic model of a reaction network is best described with an accompanying worked example. To this purpose we introduce the system depicted in Figure 1.4. This system uses reversible Michaelis-Menten kinetics for all reactions, and the rate equations and ODEs that form the model are listed below (Equations 1.27 - 1.34).

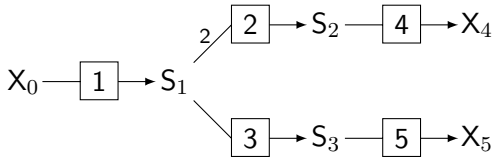

Figure 1.4: *A worked example consisting of a simple single branch system.*

$$v_1 = \frac{\frac{V_1}{K_{X_0,1}}\left([X_0] - \frac{[S_1]}{K_{eq,1}}\right)}{1 + \frac{[X_0]}{K_{X_0,1}} + \frac{[S_1]}{K_{S_1,1}}} \tag{1.27}$$

$$v_2 = \frac{\frac{V_2}{K_{S_1,2}}\left([S_1] - \frac{[S_2]}{K_{eq,2}}\right)}{1 + \frac{[S_1]}{K_{S_1,2}} + \frac{[S_2]}{K_{S_2,2}}} \tag{1.28}$$

$$v_3 = \frac{\frac{V_3}{K_{S_1,3}}\left([S_1] - \frac{[S_3]}{K_{eq,3}}\right)}{1 + \frac{[S_1]}{K_{S_1,3}} + \frac{[S_3]}{K_{S_3,3}}} \tag{1.29}$$

$$v_4 = \frac{\frac{V_4}{K_{S_2,4}}\left([S_2] - \frac{[X_4]}{K_{eq,4}}\right)}{1 + \frac{[S_2]}{K_{S_2,4}} + \frac{[X_4]}{K_{X_4,4}}} \tag{1.30}$$

$$v_5 = \frac{\frac{V_5}{K_{S_3,5}}\left([S_3] - \frac{[X_5]}{K_{eq,5}}\right)}{1 + \frac{[S_3]}{K_{S_3,5}} + \frac{[X_5]}{K_{X_5,5}}} \tag{1.31}$$

$$\frac{dS_1}{dt} = v_1 - 2v_2 - v_3 \tag{1.32}$$

$$\frac{dS_2}{dt} = v_2 - v_4 \tag{1.33}$$

$$\frac{dS_3}{dt} = v_3 - v_5 \tag{1.34}$$

1.1.2.3 The kinetic model of a reaction network

A complete kinetic model of a reaction network is simply the set of all its ODEs. We can neatly express these ODEs in matrix notation in the form

$$\frac{d\mathbf{s}}{dt} = \mathbf{N}\mathbf{v} \tag{1.35}$$

where $\mathbf{s}$ is a vector of species, or metabolite, concentrations, $\mathbf{N}$ is a matrix reflecting the stoichiometry of the system, and $\mathbf{v}$ is a column vector of the rate equations. From the worked example we would have

$$\frac{d\mathbf{s}}{dt} = \mathbf{N}\mathbf{v} = \begin{bmatrix} 1 & -2 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} \tag{1.36}$$

The stoichiometry matrix ($\mathbf{N}$) consists of rows, one for each metabolite, and columns, one for each reaction. An entry in the matrix indicates the number of molecules of that row's metabolite that react in the column's reaction. Positive numbers indicate that the metabolite is produced by the reaction,

negative numbers indicate consumption, and zero indicates non-participation. The rate equations themselves are placed in a column vector ($\mathbf{v}$) called the rate vector, in an order corresponding to the columns in the stoichiometry matrix of the reactions they govern. Multiplying the stoichiometry matrix by the rate vector produces a vector of ODEs ($\frac{d\mathbf{s}}{dt}$) describing the entire model.

Where moiety conservation is encountered, where a (sub)set of ODEs are linearly dependent, a single ODE from the set is chosen as dependent and can be omitted. As an example, consider, briefly, the system outlined in Figure 1.2. The system is composed of two ODEs,

$$\frac{dATP}{dt} = v_2 - v_1 \tag{1.37}$$

$$\frac{dADP}{dt} = v_1 - v_2 \tag{1.38}$$

Simple visual inspection suffices to determine that

$$\frac{dADP}{dt} + \frac{dATP}{dt} = 0 \tag{1.39}$$

from which it follows that

$$ADP + ATP = c \tag{1.40}$$

where c is constant.

Any one of the ODEs can be designated the dependent ODE and replaced by Equation 1.40.

1.1.2.4 Time hierarchy of reactions on the system border

Open systems are formed by clamping the concentration values of metabolites on the borders of the system, known as external metabolites, creating constant sources and sinks. This amounts to the assumption that these external metabolites are well-buffered by very fast reactions outside the system.

1.1.2.5 Modelling multiple compartments

One should note briefly that it is possible to model systems with multiple physical compartments that separate the pool of a particular metabolite physically. In this case, models consider two or more separate pools of the same metabolite, one in each compartment. Despite being chemically identical, the model treats the separated pools as different metabolites.

1.2 The goals of systems biology

Mathematically, the models described earlier fall into the category of initial value problems (IVP), wherein we have a system describing changes in a collection of variables from some time point to another time point shortly thereafter. We must however provide initial values for these variables. Apart from simply creating models of metabolic networks, let us review the goals and common problems we wish to solve in the field of systems biology.

1.2.1 Behaviour in time

Observation of the time dependent behaviour of a system is a common goal in systems biology. Fortunately, a simple numerical integration over time can produce plots of fluctuating metabolite levels. Unfortunately, metabolic networks are often considered stiff systems, and many software integrators may struggle to integrate complex networks. In addition, we wish to observe the system over time whilst being able to introduce discontinuous events, such as major fluctuations in variable values or even changes in parameter values.

1.2.2 Finding steady state fluxes and metabolite levels

A system is in steady state when its net flux is zero, i.e.

$$\frac{d\mathbf{s}}{dt} = \mathbf{N}\mathbf{v} = 0 \tag{1.41}$$

Solving this system requires a nonlinear solver, which in turn requires the reduction of the system to a set of independent algebraic equations. Alternatively, numerical integration can performed until such time as a steady state can be inferred from the insignificantly small changes in metabolite levels over time. Often the latter method is required to provide a usable set of initial conditions for the nonlinear solver, or when the former method proves unable to solve a complex system. It would be convenient to be able to specify models directly as a set of possibly dependent differential and algebraic equations, and use a solver capable of dealing with systems of differential and algebraic equations.

1.2.3 Stoichiometric analysis

Investigation of the stoichiometry of a metabolic network can identify flux relationships in the steady state, as well as conservation relationships. One method for finding both these relationships starts by augmenting the stoichiometry matrix ($\mathbf{N}$) with an identity matrix of dimension equal to the num-

ber of metabolites ($\mathbf{I}_m$), in which each column represents a single metabolites change in time, to form $\mathbf{N}|\mathbf{I}_m$.

$$\mathbf{N}|\mathbf{I}_m = \left[\begin{array}{ccccc|ccc} 1 & -2 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 \end{array}\right] \tag{1.42}$$

Gaussian elimination is then used to reduce $\mathbf{N}|\mathbf{I}_m$ to row echelon form, yielding $(\mathbf{N}|\mathbf{I}_m)'$. In our example, the matrix is already in row echelon form.

1.2.3.1 Identifying steady state flux relationships using the system kernel

If we split $(\mathbf{N}|\mathbf{I}_m)'$ into its original constituent matrices, we have the transformed stoichiometry matrix $\mathbf{N}'$ and the transformed identity matrix $\mathbf{I}'$. Placing these in the equation

$$\mathbf{N}'\mathbf{v} = \mathbf{I}'\mathbf{d} \tag{1.43}$$

where $\mathbf{d}$ is a column vector of the metabolite time derivatives, we can set the right hand side to zero to represent the steady state, where all those derivatives are zero.

$$\mathbf{N}'\mathbf{J} = 0 \tag{1.44}$$

where $\mathbf{J}$ now refers to a vector of steady-state fluxes.

The pivots of $\mathbf{N}'$ identify a valid set of columns which are *dependent* fluxes, meaning each of these can be expressed as a linear combination of the remaining *independent* fluxes. Equation 1.45 demonstrates our continuing example, with the pivots identified in bold face. In this system, fluxes J_4 and J_5 are independent.

$$\mathbf{N}' = \begin{bmatrix} \mathbf{1} & -2 & -1 & 0 & 0 \\ 0 & \mathbf{1} & 0 & -1 & 0 \\ 0 & 0 & \mathbf{1} & 0 & -1 \end{bmatrix} \tag{1.45}$$

Multiplying $\mathbf{N}'$ by a vector of fluxes, which we use instead of rates since we are dealing with the steady state, we obtain

$$0 = \mathbf{N}'\mathbf{J} = \begin{bmatrix} 1 & -2 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} J_1 \\ J_2 \\ J_3 \\ J_4 \\ J_5 \end{bmatrix} \tag{1.46}$$

Expressing each flux in terms of only independent fluxes we obtain the steady state flux relationships of the system.

$$J_1 = 2J_2 + J_3 = 2J_4 + J_5 \tag{1.47}$$
$$J_2 = J_4 \tag{1.48}$$
$$J_3 = J_5 \tag{1.49}$$
$$J_4 = J_4 \tag{1.50}$$
$$J_5 = J_5 \tag{1.51}$$

Putting Equations 1.47 to 1.51 into a matrix form we find a valid kernel matrix (**K**) for the system.

$$\begin{bmatrix} J_1 \\ J_2 \\ J_3 \\ J_4 \\ J_5 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} J_4 \\ J_5 \end{bmatrix} = \mathbf{K} \begin{bmatrix} J_4 \\ J_5 \end{bmatrix} \tag{1.52}$$

1.2.3.2 Identifying conservation relationships using the link matrix

Identifying conservation relationships involves finding the dependent and independent metabolite variables, i.e. seeking out those variables which can be expressed completely in terms of other metabolites. The worked example introduced in Section 1.1.2.2 contains no conservation relationships so we will fall back on the ATP/ADP example illustrated in Figure 1.2. We give $(\mathbf{N}|\mathbf{I})$ for this example, and $(\mathbf{N}|\mathbf{I})'$

$$\mathbf{N}|\mathbf{I} = \begin{array}{c|cc|cc} & v1 & v2 & A\dot{T}P & A\dot{D}P \\ \hline ATP & -1 & 1 & 1 & 0 \\ ADP & 1 & -1 & 0 & 1 \end{array} \tag{1.53}$$

$$(\mathbf{N}|\mathbf{I})' = \begin{array}{c|cc|cc} & v1 & v2 & A\dot{T}P & A\dot{D}P \\ \hline ATP & -1 & 1 & 1 & 0 \\ ADP & 0 & 0 & 1 & 1 \end{array} \tag{1.54}$$

Each zero row of $\mathbf{N}'$ identifies one linear dependency among the ODEs, given by the corresponding row of $\mathbf{I}'$, the columns of which refer to the ODEs.

$$\frac{dADP}{dt} + \frac{dATP}{dt} = 0 \tag{1.55}$$

which yields the conservation relationship $ATP + ADP = c$ obtained previously.

One of the ODEs is chosen as dependent and the corresponding row in $\mathbf{N}$ is deleted to give the reduced stoichiometric matrix $\mathbf{N}_R$. If, in our example, ADP is chosen as dependent then we can, using the linear constraint on the ODEs,

construct a matrix $\mathbf{L}$, called the link matrix, that captures the relationship between $\mathbf{N}$ and $\mathbf{N}_R$, namely $\mathbf{N} = \mathbf{L}\mathbf{N}_R$:

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} -1 & 1 \end{bmatrix} \tag{1.56}$$

1.2.4 Metabolic control analysis

The analysis and quantification of the control exerted on the variable concentrations and rates through the system by various system parameters both at steady state and over time, is a well developed framework, similar to that used in electrical engineering (Ingalls, 2004), known as metabolic control analysis (MCA) (Kacser and Burns, 1973; Heinrich and Rapoport, 1974), a concise summation of which can be found in Hofmeyr (2001).

1.2.5 A brief overview of software tools

The mathematical solutions to most of our problems exist already, and are well developed. As such any general mathematical toolbox software provides abstracted computational solutions. Software such as Mathematica (Wolfram, 1991), Matlab (Gilat, 2004), and its open source competitor, GNU Octave (Eaton, 2002), can thus be considered software tools for the systems biology field. Scientific library packages such as the Linear Algebra Package (LAPACK) (Anderson et al., 1999) and SciPy (Jones et al., 2001), offer similar capabilities for the development of custom software tools. But, the general nature of these tools makes the specification of a model tedious and overly complex, and often requires detailed knowledge of the underlying mathematics combined with some skill in programming. The more focused software tools actually perform most of their computation using these toolboxes or libraries, but provide automated creation of the required custom scripts by parsing less complex model description files.

1.2.5.1 Systems biology software history

Some of the earliest uses of computers in the enzyme kinetics field were of analogue computers by Hommes (1962), Walter and Morales (1964) and Walter (1966) in the establishment of the range of validity of the quasi-steady-state-assumption implied by the use of Michaelis-Menten kinetics. As digital computers rapidly replaced analogue ones and FORTRAN became widely available (See Section 1.3.1) more generalised mainframe software packages for systems biology, although the term did not yet exist, were developed (Curtis, 1976; Roman and Garfinkel, 1978).

With the prevalence of home microcomputers rising, modelling software able to run on these scaled down architectures was developed, including METAMOD (Hofmeyr and van der Merwe, 1986), SCoP (Kootsey et al., 1986), and

later SIMFIT (Holzhutter and Colosimo, 1990), and CONTROL (Letellier et al., 1991).

The advent of modern 32-bit protected mode architectures brought about the creation of a set of modelling applications regarded as formative in the field of computational systems biology, including Metamodel (Cornish-Bowden and Hofmeyr, 1991), Scamp (Sauro, 1993), and Gepasi (Mendes, 1993).

1.2.5.2 Current tools

At the time of writing, a large variety of software tools has been developed by various groups around the world, most providing a common subset of abilities but ultimately designed to specialise in a particular form of analysis. Some of the more popular and recent ones are discussed briefly, and tabular comparison of their features is presented in Table 1.1. A similar comparison of available software, dealing with fewer individual tools but in greater depth was performed by Pettinen et al. (2005).

CoPaSi: The COmplex PAthway SImulator (Hoops et al., 2006) is descended from Gepasi, and available free for non-commercial use from `http://www.copasi.org`. CoPaSi provides a both a command line option for batch processing of models, and a graphical user interface (GUI) for interactive model design and analysis.

Jarnac: Jarnac (Sauro, 2000) is the direct descendent of Scamp, also developed by Sauro. It is essentially an interpreter for a powerful model description and manipulation language. The language has both descriptive elements, allowing the specification of a model, and procedural elements which allow programatic manipulation and analysis of the model. The language is similar to BASIC in many respects, and common programming constructs such as loops, conditionals and sub-routines are available. Jarnac implements both stochastic and deterministic modelling frameworks, and can be extended via user defined functions and modules. It is available for download from `http://www.sys-bio.org`.

ScrumPy: ScrumPy (Poolman, 2006) provides an interactive command line prompt via the Python interpreter. It is capable of both kinetic and structural modeling, but focuses on structural analysis of large (genome-scale) networks. Because it is offered as a collection of utilities, it is readily extensible. ScrumPy is available under the GNU General Public License (GPL) from `http://mudshark.brookes.ac.uk/ScrumPy`.

iBioSim: iBioSim (Myers et al., 2008) is a recently released GUI tool primarily developed for the analysis of genetic circuits. iBioSim consists of a collection of inferior tools, meaning "executed subordinate to" and not a reference to their quality. Although the majority of these tools deal with

the creation, inference, and analysis of genetic circuits, iBioSim is also capable of modelling metabolic networks. iBioSim can be downloaded from `http://www.async.ece.utah.edu/iBioSim`.

SBMLOdesolver: The SBML ODE Solver Library (SOSLib) (`http://www.tbi.univie.ac.at/~raim/odeSolver` is a programming library with a command line interface written in C. It is capable of time dependent sensitivity analysis.

MathSBML: MathSBML `http://sbml.org/Software/MathSBML` is a Mathematica module which provides an interface for loading SBML (See Section 1.2.6) files as Mathematica models. Once the model is loaded the full set of Mathematica features can be used, and several convenience functions are provided to make common tasks such as time simulation easy.

SBToolbox2: The Systems Biology Toolbox (Schmidt and Jirstrand, 2006) is a Matlab package that provides an interface to SBML and various systems biology tools within the Matlab framework.

Roadrunner: Roadrunner is a compiled C# program that runs as a background service. It accepts models and requests for analyses and serves the results.

JWS Online: The Java Web Simulation Project (JWS) Online (Olivier and Snoep, 2004) is both a software tool, and an online repository of published and curated models. Using a downloadable Java applet and WebMathematica as a computational back end, JWS Online facilitates online kinetic modelling. JWS Online can be used at `http://jjj.biochem.sun.ac.za`.

PySCeS: The Python Simulator for Cellular Systems (Olivier et al., 2005) is a command line driven software tool that is run as a Python module, allowing easy extension. It provides a modular collection of tools and automates their use to allow numerical analysis and exploration of kinetic models. The user may specify which solvers/integrators to use, and can plot or export resulting data. As the entire model is available for introspection in the running Python interpreter, all data and results can be accessed in a programmatic manner, either interactively using the Python interactive command line, or in an automated manner, by running a saved script. PySCeS is distributed under a BSD opensource licence, and is available from `http://pysces.sourceforge.net`.

	Time Course Integration	Steady State Analysis	Elementary Flux Mode Determination	Flux Balance Analysis	Mass Conservation Determination	Metabolic Control Analysis	Spatial Modelling	SBML Import	SBML Editing	SBML Export	Time-dependent Sensitivity Analysis	Stability Analysis	Stochasitic Simulation
CoPaSi	•	•	•	•	•	•		•		•	•	•	•
Jarnac	•	•	•	•	•	•		•		•		•	•
ScrumPy	•	•	•	•	•	•						•	
iBioSim	•	•						•	•	•			•
SBMLOdesolver	•	•						•			•		•
MathSBML	•	•		•	•	•		•				•	•
SBToolbox2	•	•		•	•	•		•				•	•
Roadrunner	•	•		•	•	•		•					•
JWS Online	•	•	•	•	•	•				•		•	
PySCeS	•	•	•	•	•	•		•		•	†	•	

Table 1.1: *A comparison of features provided by available software*: A more exhaustive list of systems biology software can be found at `http://sbml.org/SBML_Software_Guide/SBML_Software_Matrix` along with a similar matrix comparison of features. Only SBML compatible software is recorded. † Soon to be implemented.

1.2.6 Model interchange and SBML

Given the multitude of available software tools for systems biology one could be led to believe that a modeller has a cornucopia of choice, but, sadly, this is not the case. No single tool is capable of solving all of our stated problems, and indeed when trying to solve a particular problem one is often forced to use a particular software tool. There are times where one has two or more separate aims for the same model, but each requires a different software package. Different software packages use different model specification formats, and indeed a given format for a particular software package may not be capable of specifying one's model in its entirety. The problem of being forced to hand code the same model for multiple tools, led to the development of the community driven Systems Biology Markup Language (SBML) (Hucka et al., 2003). SBML is an eXtensible Markup Language (XML) derived language, designed to describe models independently of any specific software tool, and independent of any solution implementations. A software library developed in

conjunction with the language specification provides a mechanism for software tools to both export and import models to and from an SBML file, facilitating the transfer of models between software tools. As such SBML has become the *de facto* standard for model interchange.

Although a particular software tool may not implement all the features in a given model, it is free to ignore those features, meaning a single model description is sufficient for achieving both original aims.

1.2.7 Systems biology frameworks

Examining Table 1.1 it is evident that no single tool is sports every feature, and certainly not every tool performs every task well. In an attempt to mitigate these problems, and to provide some level of automation to the use of SBML, systems biology frameworks are being developed.

BioSPICE: BioSPICE (`http://biospice.sourceforge.net`) is an open source framework that provides integrated use of a large collection of specific tools. The BioSPICE tool set concentrates on spatio-temporal modelling. This project is no longer developed or maintained.

SBW: The Systems Biology Workbench (`http://sbw.sourceforge.net`) is another framework that treats other software tools as 'modules', such as Jarnac and Roadrunner for most of the computation, and metatool (Pfeiffer, 1999) for the determination of elementary modes. The details of file formats and data marshalling are managed by SBW. Many modules serve as user friendly graphical front ends to more complex back end modules, as is the case with JDesigner being used to graphically design models for analysis with Jarnac.

Virtual Cell: Virtual Cell (Loew and Schaff, 2001) is a framework with a different aim to SBW and BioSPICE. It focuses heavily on spatial modelling of diverse cellular structures consisting of multiple compartments of varying sizes. Coupled with this focus is a strong set visualisation components.

Other frameworks include ECell (`http://www.e-cell.org`) and BioUML (`http://www.biouml.org`), which function in much the same way as BioSPICE and SBW.

1.3 The rise of **Python** as a scientific programming language

1.3.1 A brief history of scientific computing

Scientific programming dates back to the 1950's with the creation of the first FORTRAN compiler (Backus et al., 1957), which opened the arena of scien-

tific computation to scientists who had no formal experience or education in programming. Formerly, scientific applications were hand written in assembly code for the exact hardware on which they would be run. This was an arduous task and required extensive knowledge of the intimate working details of the processor used to execute the code. The portability of FORTRAN code entrenched FORTRAN as the language of choice in the scientific community, aided in large part by its speed of execution, which was only slightly slower than hand coded assembly when a good compiler was used, and ease of coding, when compared to assembly language.

In the early 1990's, with the advent of reasonably priced powerful desktop personal computers, the scientific computing community expanded rapidly. As these computers grew in power the complexity and scale of problems that could be solved grew with them. The demand for high performance, generic, flexible, and reliable libraries that abstracted the implementation details of code patterns common to many fields, such as linear algebra algorithms, increased relative to the complexity of the problems being tackled (Drummond et al., 2005).

The result was a move towards the use of C/C++ (Kernighan and Ritchie, 1978; Stroustrup, 1986) for performance oriented libraries of high complexity and general nature (Hindmarsh et al., 2005). C and C++ offered language features like pointers, dynamic memory allocation, and were more structured in nature than FORTRAN. These features allowed for better performance, implementation of parallel algorithms, which were becoming increasingly attractive as component costs came down in price, and far greater modularization of code, and hence greater code flexibility and reusability. Altogether, *co-operative* development of complex high performance software was easier in C/C++ than, the then standard, FORTRAN-77. FORTRAN-90, although an official standard at that point in time, suffered from a lack of stable compilers. Unfortunately, the very language features that made C/C++ attractive incur additional syntactic overhead, and in the case of memory management a heavy semantic overhead, making C/C++ comparatively difficult to master.

1.3.2 Python as a scientific programming language

Python (`http://www.python.org`) is an interpreted, object-oriented, procedural programming language developed by Guido van Rossum. It features a clean syntax, enforced indentation style, is highly structured and highly expressive, meaning fewer lines of code are needed to accomplish a particular task. The language offers a comprehensive set of basic data types, as well as advanced high level data structures such as lists and dictionaries (associative arrays). Being an interpreted language means there is no compilation step in the testing cycle, and it is not necessary to learn additional meta-language idioms required to compile and link a newly written program.

Python runs on a wide variety of platforms, making programs written in Python extremely portable. Python code can even be executed on some cellphones (`http://opensource.nokia.com/projects/pythonfors60`)! Python comes with an interactive interpreter, which facilitates easy and quick exploration of the basic language as well as the numerous modules provided with the standard Python library.

Python has an active user community, allowing newcomers to the language to obtain assistance in a variety of ways, including extensive online documentation (`http://docs.python.org`), a wiki containing tutorials and cookbook solutions for common tasks, an interactive IRC channel (`irc://freenode.org/#python`), and the usual plethora of books from the introductory (van Rossum, 2003a; Maruch and Maruch, 2006) to the advanced (van Rossum, 2003b; Lutz, 2006; Martelli, 2003).

Perhaps one of the strongest indications of Python's easy learning curve, is its increasing adoption as a language of instruction not only in computer science itself (Downey et al., 2002; Gauld, 2000; Zelle, 2003), but also as an educational tool in more traditional sciences (Urner, 2004).

Python boasts an impressive standard library capable of dealing with common programming tasks, *inter alia* date and time manipulation, networking, multi threaded and multi processor programming, database access, XML parsing, and web based programming. The majority of the standard library is implemented in C and hence provides exemplary performance, despite being available to Python programmers as standard Python modules via Python's extension interface. For those libraries not available in the standard library, and not implemented in Python, the language offers built in methods for extending the language by building extension modules in C which are then readily available to Python as standard Python modules.

However, scientific programming is no introductory or common task, often involving the use of complex, highly optimised algorithms. It is here that Python shines, through the inclusion of two Python packages: NumPy (`http://numpy.scipy.org`), and Scientific Python, known as SciPy (Jones et al., 2001). NumPy provides efficient support for arrays of arbitrary dimension and homogeneous type, and a strong suite of linear algebra functions to act on them. SciPy, which depends on NumPy, provides a collection of generic scientific utilities including ODE solvers, nonlinear solvers, integrators, general mathematical functions not provided by the standard Python math module, genetic algorithms, and more. Like the standard library modules, much of SciPy is implemented in lower level compiled languages such as C/C++ or FORTRAN, thus providing excellent performance whilst still benefiting from the higher level nature of the Python language. Accompanying the code resources provided by NumPy and SciPy, there there are books dedicated to scientific programming in Python (Langtangen, 2004; Kiusalaas, 2005).

1.4 Languages competing with **Python** in scientific programming

Python is not the only high level language available for scientific computing. The Practical Extraction and Reporting Language (PERL), originally written by Larry Wall and available from (`http://www.perl.org`) has a firm hold in the traditional bioinformatics world of nucleic acid and protein sequences analysis, due in great part to its syntactically convenient, if somewhat arcane, string manipulation features and tightly integrated regular expression usage. Despite the existence of many large software projects written in PERL, many in the bioinformatics arena such as BioPERL (`http://www.bioperl.org/wiki/Main_Page`) and BioMart/ENSEMBL (`http://www.ensembl.org`), its use is commonly restricted to its original purpose, i.e. string processing and file handling in server administration, and CGI scripting; Both of which benefit from the same string processing convenience that the bioinformatics world uses.

Other languages worth mentioning include Ruby and R. Ruby (`http://www.ruby-lang.org/en`), written by Yukihiro Matsumoto, is loosely based on a combination of PERL, Smalltalk and others. It is a mature general purpose scripting language, that is quickly increasing in popularity due in large part to the Ruby on Rails web development framework. Scientific computing in Ruby is, however, still somewhat immature. Ruby features a complete set of bindings for the GNU Scientific Library (`http://www.gnu.org/software/gsl`), and an equivalent to NumPy in the form of the NArray library. Nevertheless, the nature of these bindings is still imperative in style, and considered to be poorly integrated with the Ruby language stylistically. Ongoing efforts on the part of the SciRuby project (`http://sciruby.codeforpeople.com`) are rapidly correcting this problem. The Internet abounds with a series of blog posts, mailing list threads and forum comments discussing the various merits and drawbacks of Ruby compared to Python. While the zealotry common to such online discussions clouds the issue, the sane consensus is that Ruby and Python are substantially equivalent, that is equivalent in substance, specifically power and expressiveness, but that they differ greatly in style, making the selection of preferred language for a particular project a matter of personal choice rather than one of technical merit.

R (`http://www.r-project.org`) is not a general purpose programming language. R is a mature, open source, software package that aims to be compatible with the proprietary statistical software package S. S, and hence R, are implemented as command line interpreters of a statistical language. The language specialises in simply expressed methods for data retrieval, from file or networked resources, analysis, and visualisation, and is self-described as a "strongly functional language". Extensive scientific computing support is provided with the basic package. Specialised algorithms or other packages are

actively developed by a large community and made available via the Comprehensive R Archive Network (CRAN), which is modelled on the TEX (CTAN) and PERL (CPAN) package distribution networks, respectively. R also supports the dynamic runtime loading of shared library files obeying both C and FORTRAN calling conventions.

1.5 The SUNDIALS package

The SUite of Nonlinear DIfferential/ALgebraic equation Solvers (SUNDIALS) (Hindmarsh et al., 2005) is a collection of robust time integrators and nonlinear solvers written in C under a BSD style open-source license. Parts of the suite are descended from ODEPACK (Hindmarsh, 1983). SUNDIALS focuses on flexibility, requiring minimal information from the user, thus facilitating easy and quick incorporation into existing code, but allowing the user to specify as much information as they desire, via a collection of optional function calls. Users are able to specify their own data structures for use with the solvers and can easily incorporate their own linear solvers and preconditioners where necessary. SUNDIALS is available as C source code for download from `https://computation.llnl.gov/casc/sundials/main.html` and for Matlab users, an interface plugin known as SundialsTB is available too.

1.5.1 The `N_Vector`

SUNDIALS offers multiple integrator and solver modules, introduced shortly, all of which are capable of being run in parallel or serial. All these modules rely on a common abstraction called the `N_Vector`. [1] This abstraction is two-fold. It is a conceptual abstraction of the problem domain, for example specifying metabolite concentrations as a vector of values. Additionally, all parallel code is encapsulated within the specific vector operations, making the `N_Vector` a code abstraction, providing a uniform application programming interface (API) to the other modules for vector and matrix storage and operations, regardless of whether the underlying implementation is serial or parallel.

1.5.2 CVODE

CVODE is an integrator for stiff and nonstiff systems of ODEs given in the explicit form $y' = f(t, y)$ (Cohen and Hindmarsh, 1996), i.e. initial value problems. CVODE uses variable-order, variable-step multi-step methods, and offers Adams-Moulton formulas for the solution of nonstiff problems, and

[1] We refer to a C data structure in the original SUNDIALS code as `N_Vector` (with the underscore) as this is the actual type name from the code. To differentiate we refer to the Python wrapper class in PySUNDIALS as `NVector` (without the underscore), which is the actual class name in the PySUNDIALS code. The underscore was dropped to maintain consistency with accepted Python naming schemes.

Backward Differentiation Formulas in fixed-leading coefficient form for stiff systems. CVODE is regarded as the next generation of the LSODA solver (Petzold and Hindmarsh, 1997; Pettinen et al., 2005). The approximate solution of the nonlinear system resulting from each integration step can be found using various versions of Newton iteration, or functional iteration, in the case of nonstiff systems. For large systems, the user can supply a preconditioner for use with a Krylov solver.

CVODE also provides root finding functionality, whereby a user can specify an arbitrary number of functions, which are evaluated at each internal time step during integration. If at any point, one of these functions evaluates to zero, the integrator stops prematurely, returning the time the function evaluated to zero. This powerful feature allows the specification of various model constraints, and facilitates the inclusion of discontinuous events.

1.5.3 CVODES

CVODES (Serban and Hindmarsh, 2005) is an integrator for stiff and nonstiff initial value problems of the form $y' = f(t, y, p)$, where p is a vector of problem parameters. CVODES provides both forward and adjoint sensitivity analysis capabilities. CVODES implements a superset of CVODE's functionality.

Forward sensitivity analysis can be used when the gradients of many outputs with respect to few parameters need to be calculated, whereas adjoint analysis is more practical in the case of large numbers of parameters, but only a few gradients need to be calculated.

1.5.4 IDA

IDA (Hindmarsh, 2000) is a module for the solution of of differential-algebraic equation (DAE) systems in the form $F(t, y, y') = 0$. It was derived from an earlier DAE solver (DASPK) which was written in Fortran[2]. IDA allows the direct specification of a system with algebraic equations, which can be used to specify models with algebraic constraints, or to specify models without the need to reduce the system to a set of strictly independent ODEs.

1.5.5 KINSOL

KINSOL is a nonlinear solver for algebraic systems using Newton-Krylov solver technology. It has been reimplemented in C, based on the Fortran package NKSOL, written by Brown and Saad (1990). It is intended for use on large systems and supplies only iterative methods to solve resulting linear systems.

[2]Naming schemes for various Fortran standards are inconsistent. In this text, we refer to all versions of FORTRAN up to FORTRAN-77 in uppercase, as was standard at the time. More recent versions have been referred to mixed case, and uppercase. We choose to use the title case, and most common form, for all versions subsequent to FORTRAN-77

1.6 The need for Python interfaces to SUNDIALS

SUNDIALS is a robust modern suite of solvers that is being widely adopted (Modin et al., 2005) by the physics and applied mathematics communities (Forster et al., 2002). It is still under active development, and sports a large and active community of users (`mailto:sundials-users@llnl.gov`). Still the adoption of this suite is slow in fields where computer science does not form an integral part of the undergraduate curriculum, due to lack of experience with compiled languages like C. The biological sciences are a prime example of this, although CVODE is beginning to see increasing use in the systems biology field in software packages such as Jarnac, SBMLOdesolver, SBToolbox2 and Roadrunner. In addition, the SUNDIALS suite provides a set of integrators and solvers accessible in a consistent manner and which collectively have more features than competitors.

As Python becomes a more popular language for scientific computing, its need for interfaces to scientific software such as SUNDIALS increases. While SUNDIALS is freely available in C, its use by Python programmers is impeded by the complexities of interfacing C and Python code. It is hoped that a Python interface for SUNDIALS would prove useful to the scientific community as a whole, not just the systems biology field.

The production of a Python interface to SUNDIALS was initially conceived as an extension to PySCeS, aiming to extend its functionality.

1.7 PySUNDIALS

PySUNDIALS is a collection of Python modules, wrapping each of the SUNDIALS components. Wrapping means the creation of a set of translation mechanisms that exist between code written in one language and code written in another language. The wrappers are conceptualised as a layer surrounding, or wrapping, some code, and facilitating the use of that code by code in another language. PySUNDIALS aims to be more than a thin wrapping layer, and provides a number of convenience features in addition to simple translation.

PySUNDIALS presents its users with a collection of objects that act in ways familiar to them as Python programmers, a so-called "pythonic" interface. This is done by taking care of tedious and error prone programming activities, such as memory management, automatically, through the use of class constructors and finalisation. Data structures such as the SUNDIALS `N_Vector` are provided as classes that act as Python sequences, a pseudo-type referring to all classes that implement certain common operations on ordered collections. The various matrix types used by SUNDIALS are wrapped as sequences of sequences, apparent to the user as uniformly dense, regardless of the underlying storage mechanisms, and of uniform majority.

In SUNDIALS C code, operations applied to `N_Vector`s are called as explicit functions with explicit parameters (Listing 1.1).

Listing 1.1: `N_Vector` addition in C

```
1  N_Vector a, b, result;
2
3  a = N_VNew_Serial(3);
4  b = N_VNew_Serial(3);
5  result = N_VNew_Serial(3);
6
7  NV_Ith_S(a,0) = 1;
8  NV_Ith_S(a,1) = 2;
9  NV_Ith_S(a,2) = 3;
10
11 NV_Ith_S(b,0) = 3;
12 NV_Ith_S(b,1) = 2;
13 NV_Ith_S(b,2) = 1;
14
15 N_VAdd_Serial(a,b,result);
16
17 N_VDestroy_Serial(a);
18 N_VDestroy_Serial(b);
19 N_VDestroy_Serial(result);
```

This means that the result vector must either be explicitly declared and have memory allocated, and then later freed, or one of the operand vectors stores the result, i.e. result is the same variable as either `a` or `b`. Python takes a different approach, preferring "inline" operators (Listing 1.2), in addition to transparently creating a new object, `result`, and managing its destruction automatically via garbage collection.

Listing 1.2: `NVector` addition using PySUNDIALS

```
1  a = NVector([1,2,3])
2  b = NVector([3,2,1])
3  result = a+b
```

PySUNDIALS implements both Python features via operator overloading, meaning Python users need not fear inadvertently modifying operands, nor concern themselves with mundane and error prone management of memory for potentially throw away values.

Python functions may be called by SUNDIALS, and are known as callback functions. Callback functions need to be "reverse" wrapped to make them callable by the integrator, or solver as the case may be. This reverse wrapping is performed implicitly and transparently by PySUNDIALS, when the SUNDIALS modules are informed which functions to use as callbacks.

Together, these behaviours form a Python module that acts cleanly, and in an *expected* manner, keeping in theme with the goals of the Python language.

Chapter 2

Implementation

2.1 Implementation overview

In this chapter the software package PySUNDIALS is introduced and its implementation details expounded upon. PySUNDIALS is a hand-coded collection of Python modules that accesses the SUNDIALS shared libraries. Various technologies to enable the linking and use of these shared libraries are investigated. The differences in source code layout and structure between PySUNDIALS and C SUNDIALS, which result from the use of compiled libraries, are enumerated and explained. A major focus of these differences, on which we elaborate, is the development of a Python style syntax for what are, in essence, C data structures. This process is known as "pythonification". Difficulties encountered during the development of PySUNDIALS are considered, and the solutions used, if any, are introduced. Finally we briefly explain methods provided for integration with NumPy and SciPy, being the major scientific library for Python.

2.2 The foreign function interface (**ctypes**)

2.2.1 The purpose and functionality of a foreign function interface

A foreign function interface (FFI) is a software mechanism facilitating the calling of routines, or the use of services, written in one language (the *guest* or *foreign* language, from a piece of code written in another language (the *host* language). It should be pointed out that this is different from remote method invocation (RMI) or remote procedure calling, in which one process communicates with another via some form of inter-process communication, e.g. XML-RPC. An FFI interfaces between two languages within a single process. There are a number of complexities that arise from the interfacing of two languages.

Even though running on the same hardware, code written in different languages may represent primitive data types differently. These differences may be as simple as using more or fewer bytes to store the same information, to larger differences where the data is placed in memory, and the order of bytes used to store multi byte data, known as "endianness". Complex data types often have completely different meta-data storage mechanisms, such as the difference between null-terminated strings, a la C/C++, and length indexed strings (Pascal).

Most languages allow for the creation of new types, usually compounded from the languages' primitive types. One language may treat these non-primitive types as a separate class, whereas another might treat all types identically, regardless of origin.

Similarly, host and guest languages may call subroutines with parameters or arguments in different orders, or more fundamentally different ways. The method a particular language chooses is called its *calling convention*. Often arguments are placed on a stack, in which case the order of placement is the sole difference, but some languages may not use the stack directly, instead forming a complex data type instance from the arguments and placing only a reference to that instance on the stack. This is especially true for interpreted or weakly typed languages.

Memory management differs significantly from one language to another. Lower level languages often leave memory management to the programmer, whereas middle to high level languages can implement garbage collection mechanisms. Where the host and guest languages use disparate memory management systems, they may come into conflict. One language may attempt to automatically manage memory for a given object but do so incorrectly because the object is created by the other language, and memory management meta-data, such as reference counting or ownership semantics, is not available to the automated system. If, for example, the host language uses garbage collection, it might attempt to discard an object created by the guest language and passed to the host language, because the guest language does not notify the garbage collector correctly that the object is still in use. Contrarily, manual memory management code might legitimately destroy an object, but when linked with code from another language which automates memory management, the legitimacy of this action might be invalidated, unless express notification is given to the automatic management processes. As an example, the guest language part of the program is a library, which under normal circumstance can discard a particular object. When used as a guest language, the host language might be given references to the object by the FFI, which the guest language knows nothing about, and thus cannot modify or discard correctly when the true object is destroyed. This leaves an invalid reference to a discarded object in the host language.

Thus, the functionality of a foreign function interface is three fold:

1. The FFI manipulates system memory directly and translates data types between host and guest conventions. Some FFIs may additionally provide type translation for user-specified types, although this is not always the case.

2. The FFI inserts translation routines at the entry and exit points of subroutines to perform reordering or other necessary translations of calling conventions.

3. The FFI makes guest code available to the host either at compile time, in which case the FFI serves as an assistant to the compilers working on each language, mediating symbol lookups, type translation, and calling convention differences eventually resulting in a single unified link. Alternatively, the FFI performs these actions at runtime by loading a pre compiled guest binary file into memory, and linking it into the running host process.

2.2.2 A comparison of foreign function interface technologies

A number of FFI technologies exist for the Python language. We will briefly explore each of them, and then explain the reasons for the ultimate choice of ctypes in PySUNDIALS.

Python C API: The Python language supplies a C application programmer's interface (API) which can be used to make code written in C available as a Python extension module. Whilst built into Python, and thus extremely portable, this method requires writing large amounts of C code by hand, and is error prone and complex. In addition, documentation is sketchy, in particular regarding the creation of user specified callback mechanisms.

Boost.Python: Boost is a collection of libraries intended for eventual inclusion into the C++ standard. Boost.Python (`http://www.boost.org/doc/libs/1_37_0/libs/python/doc/index.html`) is one of these libraries and is designed to allow "seamless interoperability between C++ and the Python programming language" (`http://www.boost.org/doc/libs/1_37_0/libs/python/doc/index.html`). It is well documented, provides a mechanism for user specified callbacks, is extremely flexible, and has a code generator for automated wrapper generation.

However, it is designed for C++ rather than C, is overly complex compared to other solutions, despite the code generator, which relies on GCC-XML, which is highly sensitive to gcc versions, and did not run correctly on the author's machine.

weave: weave (`http://www.scipy.org/Weave`) is a Python package from the SciPy collection, but also available as a standalone package, which allows

the inline inclusion of C code. It is very simple to use, but is designed only for short snippets of C code to be included, not for wrapping entire libraries. It functions by automatically compiling and creating the specified snippets into Python extension modules, and caching these modules for future use. It requires that the user has a C compiler.

SWIG: The Simplified Wrapper and Interface Generator (SWIG) (`http://www.swig.org/`) is a multi-language automated wrapper generator that provides a compile time FFI to C/C++ for a variety of languages including Python. It is well documented, and for simple cases the most easy and convenient method of creating Python wrappers of C libraries. Our use case is complicated by the requirement of callback capabilities, which require user intervention in the form of hand coded section to specify callback function types.

instant: instant (`http://heim.ifi.uio.no/~kent-and/software/Instant/doc/Instant.html`) is a small Python module that functions very similarly to weave, except that its underlying compilation mechanism is based on SWIG, rather than on a pure C compiler. Until very recently it has only been available for POSIX style operating systems, and hence was rejected as an FFI choice for PySUNDIALS.

Pyrex: Pyrex (`http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`) is a language specifically designed for the creation of Python extension modules. Syntactically is is a hybrid between C and Python, tending toward the higher level Python side. It is a compiled language, with the generated output being a Python extension module. Sections of code that do not involve Python variables or functions are automatically converted to C and compiled. As it lacks a code generator, it is more suited to the creation of new extension modules, rather than the wrapping of already extant libraries. When the PySUNDIALS project was started, Pyrex was a new project, still undergoing significant development. It was considered too unstable for use.

ctypes: ctypes (`http://www.python.org/doc/2.5.2/lib/module-ctypes.html`) is a Python module that is part of the Python standard library from version 2.5 onwards. It provides methods to load shared libraries dynamically and link them into the running Python process. Functions and symbols exported by these loaded libraries can then be accessed via a specialised class instance that results from the loading of the library. Type conversion for primitive types such as strings, floats and integers is automatic, and more complex types can be defined. Where more complex types are used, either as function parameters or returned by a function in the guest library, these types must be specified. Type specification of complex parameter or return types is done via a simple

mechanism of setting attributes of the library's class instance. Using ctypes requires only Python code and is comparatively simple to use. A code generator is available, but again, it requires GCC-XML which would not run correctly on the author's machine.

SWIG and ctypes were the front runners in the choice of an FFI for PySUNDIALS. ctypes was chosen over SWIG for the following reasons:

1. ctypes was more familiar to the author.
2. ctypes coding is done entirely in Python; Other technologies involved either learning a new language, or a considerable amount of coding in C, leading to slower development time.
3. ctypes is a built-in module from Python 2.5 onwards, and so does not create additional dependencies for PySUNDIALS.
4. callback functions, which are a central working mechanism of SUNDIALS, are significantly simpler to implement in ctypes than in SWIG.
5. SWIG automation, although a potent labour saving device, generates a wrapper interface that closely resembles the original C code being wrapped. Substantial effort is required to wrap the wrappers to provide a more pythonic interface.

2.3 Structural differences in code layout between SUNDIALS and PySUNDIALS source

The source layouts of SUNDIALS and PySUNDIALS differ significantly. SUNDIALS source contains separate header files for each matrix storage mechanism (dense, banded, or parallel banded) in a subdirectory shared between each of the four main modules. In turn each module has its own subdirectory with a header file for the module itself, and one for each available linear solver for that module. However when compiled all these separate sources are linked together into six shared libraries, one for each module, one for the serial implementation of the `N_Vector`, and one for the parallel.

PySUNDIALS matches the compiled/linked structure of SUNDIALS rather than the source structure, as the shared libraries cannot be separated into their original source components by ctypes post compilation, and it is these that ctypes loads. A single shared library must therefore be treated as a single unit. PySUNDIALS, thus, consists of one module for each shared library, namely `cvode`, `cvodes`, `ida`, `kinsol`, and `nvecserial`. The parallel implementation of the `N_Vector` has not yet been wrapped.

2.4 Pythonification

2.4.1 `NVector`s, dense matrices and banded matrices

The `N_Vector` type in SUNDIALS contains many meta-data items for use by the solver(s) that will be accessing any `N_Vector` variables. However, to the programmer, an `N_Vector` is simply an ordered list of real numbers, and could in an abstract sense be represented equally well as an array or a Python list. The Python language, although dynamically typed, provides abstract mechanisms for classing similar types together. These mechanisms are known as protocols, and PySUNDIALS has implemented the `NVector` wrapper class using the sequence protocol. Any class or type in Python implementing the sequence protocol must implement

- an ordered collection of elements
- a subscription operator that returns a specific element, or returns a specified range of elements as a new sequence of the same type
- assignment by subscription or range subscription
- a length operator which returns the number of items in the sequence

Thus, to a programmer already familiar with Python, an `NVector` behaves exactly like a Python list. Examples of these syntactic differences can be found in Listings 2.2 and 2.4.

Just as `N_Vector`s can be expressed as Python lists, the various matrix types in SUNDIALS, dense and banded, can be thought of in the abstract sense as lists of lists. This introduces the familiar problem of choosing a majority form (row or column) for the matrix, determining which list (inner or outer) represents what. SUNDIALS attempts to conform to the mathematical convention of row majority, but implementation details in the memory layout of banded matrices make this impossible. SUNDIALS provides a remedy in the form of macros that provide row major accessors. Macros are a language feature of the C language not present in Python. PySUNDIALS instead provides Python classes for the various types of matrices, each of which implements the functionality of the C macros in the overloading of the subscription operators for the class (See Listing 2.1), providing a consistent row major matrix set, that acts identically to a Python list of lists.

Listing 2.1: Illustrative example of matrix access using PySUNDIALS

```
1 >>> from pysundials import cvode
2 >>> Md = cvode.DenseMat(3,3)
3 >>> for r in range(3):
4 ...     for c in range(3):
5 ...             Md[r][c] = r*c
6 ...
7 >>> Md
8 0.000000 0.000000 0.000000
```

```
0.000000 1.000000 2.000000
0.000000 2.000000 4.000000
```

Although SUNDIALS provides a comprehensive set of vector operations for the `N_Vector` type, *inter alia*: vector addition, inner product, scalar product; these operations are provided as functions, and not operators. The C language does not allow operator overloading. PySUNDIALS provides the same operations using operator overloading, yielding a syntax more familiar both to mathematicians and Python programmers. These overloaded operators call the underlying SUNDIALS functions rather than duplicating their functionality in pure Python, hence maintaining performance. Where a SUNDIALS vector operator function might require the prior declaration of an auxiliary `N_Vector` for the purposes of temporary storage of the result of the operation, PySUNDIALS automates this process by creating new `NVector` objects within the overloaded operator wrapper function, which are then returned as the result. This has the additional advantage of conforming to the Python language convention of having operators return newly created objects, to be assigned to a variable or discarded according to the programmer's wishes.

Listing 2.2: Illustrative example of vector operations in C using SUNDIALS

```c
#include <stdio.h>
#include <nvector/nvector_serial.h>

int main(int argc, char **argv) {
        N_Vector u, v, w;

        u = N_VNew_Serial(5);
        NV_Ith_S(u,0) = 0;
        NV_Ith_S(u,1) = 1;
        NV_Ith_S(u,2) = 2;
        NV_Ith_S(u,3) = 3;
        NV_Ith_S(u,4) = 4;

        v = N_VNew_Serial(5);
        NV_Ith_S(v,0) = 1;
        NV_Ith_S(v,1) = -1;
        NV_Ith_S(v,2) = 1;
        NV_Ith_S(v,3) = -1;
        NV_Ith_S(v,4) = 1;

        w = N_VNew_Serial(5);

        N_VScale_Serial(-1,v,w); // unary negation
        printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
                NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
                NV_Ith_S(w,3), NV_Ith_S(w,4));

        N_VAddConst_Serial(v,1,w); //scalar addition
        printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
                NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
                NV_Ith_S(w,3), NV_Ith_S(w,4));

        N_VLinearSum_Serial(1,u,1,v,w); //vector addition
        printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
                NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
                NV_Ith_S(w,3), NV_Ith_S(w,4));
```

```
37
38          N_VAddConst_Serial(v,-1,w); //scalar subtraction
39          printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
40                  NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
41                  NV_Ith_S(w,3), NV_Ith_S(w,4));
42
43          N_VScale_Serial(-1,v,w); // vector subtraction
44          N_VLinearSum_Serial(1,u,1,w,w);
45          printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
46                  NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
47                  NV_Ith_S(w,3), NV_Ith_S(w,4));
48
49          N_VScale_Serial(2,v,w); // scalar multiplication
50          printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
51                  NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
52                  NV_Ith_S(w,3), NV_Ith_S(w,4));
53
54          N_VProd_Serial(u,v,w); // element wise multiplication
55          printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
56                  NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
57                  NV_Ith_S(w,3), NV_Ith_S(w,4));
58
59          N_VScale_Serial(1/2.0,v,w); // scalar division
60          printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
61                  NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
62                  NV_Ith_S(w,3), NV_Ith_S(w,4));
63
64          N_VDiv_Serial(u,v,w); // element wise division
65          printf("%3.1f␣%3.1f␣%3.1f␣%3.1f␣%3.1f\n",
66                  NV_Ith_S(w,0), NV_Ith_S(w,1), NV_Ith_S(w,2),
67                  NV_Ith_S(w,3), NV_Ith_S(w,4));
68
69          printf("%g\n", N_VDotProd_Serial(u, v)); //inner product
70
71          N_VDestroy_Serial(u);
72          N_VDestroy_Serial(v);
73          N_VDestroy_Serial(w);
74
75  }
```

Listing 2.3: Output from Listing 2.2

```
1  -1.0 1.0 -1.0 1.0 -1.0
2  2.0 0.0 2.0 0.0 2.0
3  1.0 0.0 3.0 2.0 5.0
4  0.0 -2.0 0.0 -2.0 0.0
5  -1.0 2.0 1.0 4.0 3.0
6  2.0 -2.0 2.0 -2.0 2.0
7  0.0 -1.0 2.0 -3.0 4.0
8  0.5 -0.5 0.5 -0.5 0.5
9  0.0 -1.0 2.0 -3.0 4.0
10 2
```

Listing 2.4: Illustrative example of vector operations using PySUNDIALS

```
1  >>> from pysundials import nvecserial
2  >>>
3  >>> u = nvecserial.NVector([0,1,2,3,4])
4  >>> v = nvecserial.NVector([1,-1,1,-1,1])
5  >>>
6  >>> -v                    # unary negation
7  [-1.0, 1.0, -1.0, 1.0, -1.0]
```

```
8  >>>
9  >>> v+1                  # scalar addition
10 [2.0, 0.0, 2.0, 0.0, 2.0]
11 >>>
12 >>> w = u+v              # vector addition
13 >>> w
14 [1.0, 0.0, 3.0, 2.0, 5.0]
15 >>> w is u               # w is newly created NVector
16 False
17 >>> w is v               # w is newly created NVector
18 False
19 >>>
20 >>> v-1                  # scalar subtraction
21 [0.0, -2.0, 0.0, -2.0, 0.0]
22 >>>
23 >>> u-v                  # vector subtraction
24 [-1.0, 2.0, 1.0, 4.0, 3.0]
25 >>>
26 >>> v*2                  # scalar multiplication
27 [2.0, -2.0, 2.0, -2.0, 2.0]
28 >>>
29 >>> u*v                  # element wise multiplication
30 [0.0, -1.0, 2.0, -3.0, 4.0]
31 >>>
32 >>> v/2                  # scalar division
33 [0.5, -0.5, 0.5, -0.5, 0.5]
34 >>>
35 >>> u/(2*v)              # element wise division
36 [0.0, -0.5, 1.0, -1.5, 2.0]
37 >>>
38 >>> u.dotproduct(v)      # inner product
39 2.0
```

2.4.2 Callback functions

Because C is a statically typed language, its functions are also typed. A function's type is determined by the type and order of parameters it takes. Python, being a dynamically typed language, does not enforce the number or type of parameters passed to a function, and within the interpreter itself, Python functions all have a single argument, which is a Python tuple of arguments, and potentially an additional dictionary if the function takes keyword arguments. This makes Python functions unsuitable as callback functions without some form of wrapping. Using ctypes, the function type of the callback must first be defined in Python, specifying the return type of the function, and the types and order of parameters it expects (See Listing 2.5, lines 1–5). Defining a function type performs two tasks. Firstly, a type class is created, whose objects are instantiated as wrappers to pure Python functions, and have methods to translate the C parameters received into a Python tuple, then passing said tuple on to the actual Python function, and similarly translating the returned Python object into a C typed value. Secondly, the type can be used to cast any Python function into a C function pointer of appropriate function type at the time of nomination of the function as a callback.

Listing 2.5: Declaration and nomination of a callback function using ctypes

```
1  CVRhsFn = ctypes.CFUNCTYPE(ctypes.c_int,
2          realtype,
3          ctypes.POINTER(nvecserial._NVector),
4          ctypes.POINTER(nvecserial._NVector),
5          ctypes.c_void_p)
6
7  def f(t, Cy, Cydot, f_data):
8          y = cvode.NVector(Cy)
9          ydot = cvode.NVector(Cydot)
10         ydot[S2] = R2(y) - R1(y)
11         ydot[S1] = R1(y) - R3(y) - R4(y)
12
13         return 0
14
15 .
16 .
17 .
18
19 cvode.CVodeMalloc(
20         cvode_mem,
21         CVRhsFn(f),
22         0.0,
23         y,
24         cvode.CV_SS,
25         1.0e-8,
26         1.0e-12
27 )
```

The nominated Python function must still be able to deal with C typed parameters (Listing 2.5, lines 8–9[1]), expect where ctypes is capable of performing automatic type translation, which is only the case for integers, floating point numbers, and strings. In the case of PySUNDIALS, users will be writing these callback functions. It is unreasonable to expect the user to have to deal with type translation of complex compound data types such as `N_Vector`s, or various matrix types, in their callback functions. Instead, PySUNDIALS creates anonymous functions that create an additional wrapper around the ctypes created wrapper, which automate the process of these additional type translations. `N_Vector`s, and the various matrix types used by SUNDIALS are wrapped in PySUNDIALS classes of appropriate type, providing manipulation of the underlying data as described in Section 2.4.1.

Those PySUNDIALS wrapper functions that require a function as a parameter create an anonymous wrapper function of appropriate type, making this

[1] `Cy` and `Cydot` are in this case crude ctypes mappings to the underlying C `N_Vector` structure. Lines 8 and 9 also use the PySUNDIALS `NVector` class constructor in a non-intuitive way, constructing the class around an already existing ctypes data structure, rather than creating a new class. When the objects `y` and `ydot` go out of scope the `NVector`s are garbage collected, and one would expect this to deallocate the memory storage of the underlying C `N_Vector` structure. However, this is not the case. Where an `NVector` is constructed around an already extant `N_Vector`, only the `NVector` class is destroyed, not deallocating the C data storage. This behaviour is intended, but since it is inconsistent with the behaviour expected from `NVector`s operating under normal circumstances, it is hidden from the user.

process transparent to the user. Thus a user need simply specify a Python function, and all the rest is done automatically.

Listing 2.6: Declaration and nomination of a callback function using PySUNDIALS

```
1  def f(t, y, ydot, f_data):
2          ydot[S2] = R2(y) - R1(y)
3          ydot[S1] = R1(y) - R3(y) - R4(y)
4
5          return 0
6
7  .
8  .
9  .
10
11 cvode.CVodeMalloc(
12         cvode_mem,
13         f,
14         0.0,
15         y,
16         cvode.CV_SS,
17         1.0e-8,
18         1.0e-12
19 )
```

2.4.3 Memory management

In SUNDIALS, data used by the solver, `N_Vector`s and all matrices must be dynamically allocated. Not being a garbage collected language, C leaves the programmer to allocate and deallocate this memory. While SUNDIALS provides a set of functions for allocation and deallocation which enforce the usage of correct types, the SUNDIALS programmer must still write the code to manage memory (Listing 2.2, lines 5, 7, 14, 21, and 71–73).

Python is an object oriented language, and PySUNDIALS wraps each SUNDIALS C structure which requires memory management with a class, allowing us to use the classes constructor and finalizer to manage memory. Constructors are called whenever a Python object is instantiated, allowing the object to call the underlying SUNDIALS allocation function to acquire the requisite memory. Similarly, when objects in Python are garbage collected, their finalizers are called, allowing any required deallocation functions to be called. As object instantiation in Python is as simple as assigning a variable a value, the entire memory management process of underlying SUNDIALS memory structures is made transparent to the user, reducing the potential for memory leaks or similar bugs.

2.4.4 Error handling

The C language has no error handling mechanism built in. When an error state is encountered, a C programmer is free to ignore it, or flag the particular state in any way they see fit. There are two conventions for error handling in

C. Functions that encounter an error state will usually return a value indicating error, usually being a value that is not in the valid range of the function. For those functions whose valid range matches or exceeds the range of the function's return type, a particular value from the return type's range is reserved and this value is returned as an error flag whilst a global variable is simultaneously set to a value indicating the exact nature of the error. This means that a C programmer must be prepared to check the return values of each function call that might encounter an error condition, and write code to handle the error value each time this is done. SUNDIALS tends to use the first convention, whereby any SUNDIALS function that could encounter an error will indicate this in its return value. A SUNDIALS user must capture each function call's return value and check it for an error condition. To complicate matters, some errors are recoverable, while others are not. SUNDIALS functions indicate success by returning 0, recoverable errors by returning a positive value, and fatal errors using a negative value.

Python uses exceptions to handle errors. When an exception is raised, if not specifically handled by the programmer, it will halt the program, as opposed to C programs continuing blithely on. PySUNDIALS captures the results of SUNDIALS functions in their respective wrapper functions and translates fatal error results into exceptions, returning all other results as normal. The use of the word "error" to describe recoverable errors is a misnomer in the case of a number of SUNDIALS functions, and these would better be described as "situations of interest", e.g. the root of a specified function was found prior to the specified output time, or the system reached a steady state. Hence, the PySUNDIALS user remains responsible for checking the return value for recoverable errors, and handling them appropriately. Regardless, the burden of error checking is reduced, as it is only required for those functions from which situations of interest may arise, and not for every function which might cause an error.

2.5 Difficulties

2.5.1 The assignment idiom "gotcha"

2.5.1.1 The C data model

Being both a statically typed and compiled language, C requires that variables be declared in advance of their usage. The declaration serves to inform the compiler of the scope, type, and hence size, of the variable. It also reserves space in memory for the variable, and binds the name of the variable to that particular reserved address in memory. Assignments to the variable alter the contents of memory at the bound address.

2.5.1.2 The Python data model

Python uses a fundamentally different data model to C. Python maintains a mapping between variable names and values, each kept in separate tables. When a variable is assigned a value in Python, an object representing that value is first created in the values table, then the variable name is created in the names table, and associated with the value object. If a reassignment occurs, the new value is created as an object in the values table, and the original variable name simply associated with the new value. If no variable names are associated with the old value anymore, or more specifically, nothing references that value, it is discarded. It is important to note that the new value assigned is a different object using different memory space, and that in the course of reassignment the association to the original value's object is lost and potentially cannot be reacquired (See Figure 2.1).

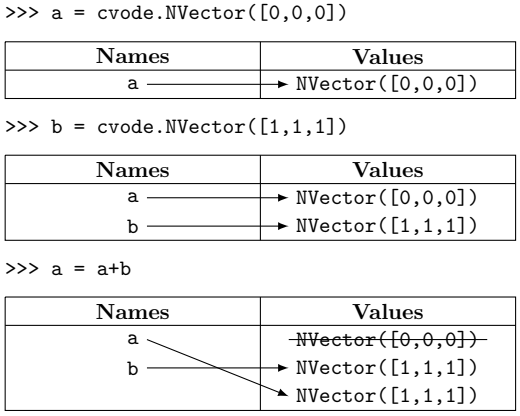

Figure 2.1: *An illustration of the Python data model*: Note that `a`'s original object is destroyed on reassignment (`a = a+b`), and that `a`'s new object and `b`'s object are not the same object in memory, despite having equivalent value.

2.5.1.3 The problem

Each of the SUNDIALS modules work by integrating or solving a given right hand side function. SUNDIALS cannot not know in advance what this function will be, as it is supplied by the user. The user must therefor *nominate* a particular function, i.e. they must inform SUNDIALS of the name of the function. The function nominated must be of a particular type. SUNDIALS passes two `N_Vectors`, amongst other variables, as parameters to the nominated function. The first represents the current state of the system in time, and the second is a container vector into which the new state of the system should be placed. Strictly speaking, the SUNDIALS `N_Vector` is a pointer to data structure, and

thus modification to the passed parameters via indirection modifies the contents of the data structure pointed to. If the pointer were to be reassigned a different value, the nominated function would no longer modify the `N_Vector` SUNDIALS uses, but the C language syntax is clear as to which is being modified, the pointer or the pointed to structure. Similarly, the strict typing of the C language would prevent the assignment of an entire array of values to the pointer itself, only allowing the assignment to the indirected pointer.

However in Python, this type protection does not exist. Consider what would seem the intuitive syntax for assignment to the result vector (`ydot`) in function `f1` of Listing 2.7. This assignment creates a new `NVector`, discarding the reference to the original `ydot` without altering its value.

Often this is not a problem, e.g., if the callback function explicitly describes the system by assigning to individual elements of the `NVector` using subscription, as in function `f2`. The use of subscription in the assignment maintains the reference to the original `ydot NVector`, modifying its value.

Listing 2.7: The assignment "gotcha" in callback functions

```
1  def f1(t, y, ydot, f_data):
2          ydot = cvode.Nvector([R2(y) - R1(y), R1(y) - R3(y) - R4(y)])
3          return 0
4
5  def f2(t, y, ydot, f_data):
6          ydot[S2] = R2(y) - R1(y)
7          ydot[S1] = R1(y) - R3(y) - R4(y)
8          return 0
9
10 def f3(t, y, ydot, f_data):
11         ydot[:] = cvode.Nvector([R2(y) - R1(y), R1(y) - R3(y) - R4(y)])
12         return 0
```

The alternative to assigning each element is to assign to the entire `NVector` using slice notation, which acts in the same way as element-wise assignment (function `f3`). This is potentially counter-intuitive to the Python programmer, and could be a source of many difficult to diagnose bugs in user programs. Indeed, this issue was a major source of bugs during the development and testing of PySUNDIALS. However, due to the nature of the Python data model, the only solution is *clear* documentation explaining the problem and the solution.

2.5.2 Garbage collection of callback wrapper functions

As explained in Section 2.4.2, functions nominated as callback functions have dynamically generated wrapper functions which perform complex type translation. This generation takes place inside the wrapper function around the SUNDIALS function in which callback nomination takes place. Dynamically generated type translation wrappers hence exist in the local scope, meaning they are garbage collected upon return from the function which performs callback nomination. This means the type translation wrapper function may no longer be available for execution at the time the callback is actually called.

The solution is to keep at least one reference to these dynamically generated functions in a list in the module scope, which will prevent their garbage collection. Ideally, when the function being wrapped goes out of scope, the dynamically generated functions should be discarded too, i.e. their references should be removed from the list, but the Python language provides no way to hook the destruction of a function, as functions are not objects and have no finalizers.

Python objects can be made to act like functions, becoming callable objects, and this led us to consider requiring users to implement their callback functions as callable objects derived from a common base class whose finalizer would handle the removal of the reference to the appropriate wrapper. It was decided, however, that this would make matters more complex for users of PySUNDIALS. The majority of callback functions would be in the global scope regardless, meaning they are only discarded at the end of the program, obviating the need to monitor function destruction. The overhead of references being kept beyond their necessary lifespan is minimal, meaning this method is only a needless addition of complexity, and hence was not implemented.

2.5.3 Determining the size of `realtype`

SUNDIALS can be compiled to use one of three precision levels, namely single, double or extended, referring to the native C floating point types. The specific type chosen is aliased within the SUNDIALS code as `realtype`. When ctypes loads the shared library, it must map the `realtype` type to a specific C type, i.e. one of single, double or extended. The ctypes module does not provide a mechanism for mapping the extended type, but more importantly, ctypes has no way of determining what size `realtype` was chosen. The size cannot be determined prior to the mapping, and the mapping cannot take place prior to the size determination. PySUNDIALS provides an auxiliary shared library written in C code which includes the SUNDIALS header files, providing a compile time determination of the size of the `realtype`. When PySUNDIALS is installed, this library is compiled against the SUNDIALS shared libraries and provides a single function that returns the size of the `realtype`.

2.5.4 Precision differences

Whilst the output of PySUNDIALS is nearly identical to SUNDIALS, precision differences do exist. These differences arise from the calculations performed in callback functions, i.e. those calculations performed in Python. Although ctypes uses the same number of bits to store a floating point number as C would, a compiled C program will often perform a series of calculations on a particular value and leave that value in a hardware register. Depending on the hardware, the register may be wider than the C standards definition for the width of the appropriate type of floating point number, hence rounding error is

reduced (Monniaux, 2008). When performing the same series of calculations in Python however, even if performed in a C function called via ctypes, the result is likely to be truncated on function return as it passes out of the hardware register and into RAM. This is guaranteed to happen if the value is stored in a Python variable before being passed on to the next calculation. Hence PySUNDIALS demonstrates more rounding error than, and hence a slight precision difference to, SUNDIALS.

2.6 Integration with NumPy

With NumPy being the standard for numeric computation in Python, it is envisaged that many users of PySUNDIALS will want to use NumPy's extensive features to manipulate their `NVector`s. The `NVector` class offers a method, `asarray`, which yields a NumPy array, of appropriate shape and data type, that is constructed around the underlying C data storage. The returned NumPy array and the `NVector` both share the underlying memory area, meaning modifications to one are reflected in the other. This allows `NVector`s to be used by NumPy and SciPy, with the simple use of their `asarray` methods.

The reverse operation, that of constructing an `NVector` around a NumPy array, is as simple as instantiating the `NVector` using the array as a parameter. It should be noted that the instantiated `NVector` is a copy of the array, and does not share the same memory. This behaviour was deliberately implemented as it conforms to the pythonic behaviour of that syntax where other types or classes are used.

Listing 2.8: NumPy integration demonstration

```
1  Python 2.5.2 (r252:60911, Oct 31 2008, 13:49:23)
2  [GCC 4.1.2 (Gentoo 4.1.2 p1.0.2)] on linux2
3  Type "help", "copyright", "credits" or "license" for more information.
4  >>> from pysundials import nvecserial
5  >>> v = nvecserial.NVector([0,1,2,3,4])
6  >>> a = v.asarray()
7  >>> a                                    #a is a numpy array
8  array([ 0.,  1.,  2.,  3.,  4.])
9  >>> a[2] = a[2]*2                        #a is mutable
10 >>> a
11 array([ 0.,  1.,  4.,  3.,  4.])
12 >>> v                                    #v reflects changes to a
13 [0.0, 1.0, 4.0, 3.0, 4.0]
14 >>> v[0] = -1
15 >>> a                                    #a reflects changes to v
16 array([-1.,  1.,  4.,  3.,  4.])
17 >>>
```

2.7 Availability of PySUNDIALS

PySUNDIALS is available for download as a Python source distribution, and a Windows 32-bit binary distribution from the PySUNDIALS homepage at

`http://pysundials.sourceforge.net`. The package is released under an MIT style licence as open source software. PySUNDIALS has been successfully compiled and run on Windows and various flavours of Linux, including Gentoo (32 and 64-bit), Mandriva, and Ubuntu (32 and 64-bit).

PySUNDIALS is still considered to be in beta testing phase, although, at the time of writing, the first release candidate is available. PySUNDIALS version numbers may seem at first misleading, considering the project is still in beta. PySUNDIALS major versions are tied to the SUNDIALS version whose header files they match. Currently, only one major version exists, namely PySUNDIALS 2.3.0, which wraps the SUNDIALS 2.3.0 shared libraries. Minor version numbers are used to track modifications to PySUNDIALS without changes in the underlying SUNDIALS code. An example of a complete version specification would be `pysundials-2.3.0-rc1`.

Complete documentation is distributed with both the source and binary distributions, and is also available online at the project homepage. It should be pointed out that the PySUNDIALS documentation is best used in conjunction with the original SUNDIALS documentation. The PySUNDIALS documentation highlights the differences between SUNDIALS and PySUNDIALS, and contains a complete function reference. Mathematical considerations and general help on usage of the SUNDIALS modules as a framework are covered thoroughly only in the original SUNDIALS users' guides. The PySUNDIALS function reference is available in a running Python interpreter via the Python docstring mechanism.

Chapter 3

Results

In this chapter we present a selection of simple usage examples of PySUNDIALS, chosen to illustrate the features of SUNDIALS and demonstrate the ease of use of PySUNDIALS. We show at least one example of each SUNDIALS module, after which we discuss the performance and coding differences between SUNDIALS and PySUNDIALS. The SUNDIALS source package contains a comprehensive set of example scripts, which have been reimplemented in Python using PySUNDIALS, both as code examples for users of PySUNDIALS and as a testing and benchmarking suite. The PySUNDIALS examples produce identical output to their SUNDIALS equivalents, bar precision differences, verifying correctness of the PySUNDIALS code. The differences discussed between SUNDIALS and PySUNDIALS are based on comparisons from the example sets, as they offer a sufficiently large sample size from which to work. The PySUNDIALS versions of the SUNDIALS examples are available as part of the source distribution of PySUNDIALS, and the source code for the examples presented in this chapter can be found in the appendices.

3.1 CVODE examples

3.1.1 A simple test model

Example 1 uses CVODE to solve a simple reaction network containing a branch and a moiety-conservation cycle, taken from Hofmeyr (2001). A schematic representation of the model is given in Figure 3.1. This model will be used as an example for each of the other SUNDIALS modules as a basis for comparison between modules and verification that results produced are correct. We will first enumerate the equations and parameter values for the model, before demonstrating that the results obtained using CVODE are identical to those obtained using LSODA in PySCeS.

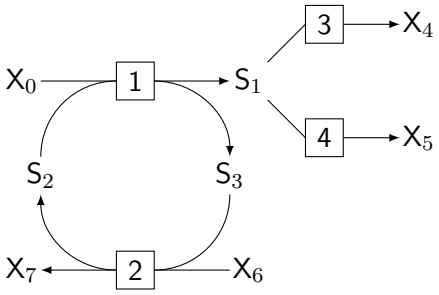

Figure 3.1: *Example 1 - A simple model with a moiety conservation cycle and a branch.*

We introduce the model's kinetics with the following four rate equations.

$$v_1 = \frac{\frac{V_1}{K_{X_0,1}K_{S_2,1}}([X_0][S_2])}{1 + \frac{[X_0]}{K_{X_0,1}} + \frac{[S_2]}{K_{S_2,1}} + \frac{[X_0][S_2]}{K_{X_0,1}K_{S_2,1}}} \tag{3.1}$$

$$v_2 = \frac{\frac{V_2}{K_{S_3,2}K_{X_6,2}}([S_3][X_6])}{1 + \frac{[S_3]}{K_{S_3,2}} + \frac{[X_6]}{K_{X_6,2}} + \frac{[S_3][X_6]}{K_{S_3,2}K_{X_6,2}}} \tag{3.2}$$

$$v_3 = \frac{V_3[S_1]}{[S_1] + K_{S_1,3}} \tag{3.3}$$

$$v_4 = \frac{V_4[S_1]}{[S_1] + K_{S_1,4}} \tag{3.4}$$

The model consists of three ODEs, two of which are independent, and one dependent.

$$\frac{dS_1}{dt} = v_1 - v_3 - v_4 \tag{3.5}$$

$$\frac{dS_2}{dt} = v_2 - v_1 \tag{3.6}$$

$$\frac{dS_3}{dt} = v_1 - v_2 \tag{3.7}$$

Parameter	Value
V_1	1
V_2	10
V_3	1
V_4	1
$K_{X_0,1}$	1
$K_{S_2,1}$	1
$K_{S_3,2}$	1
$K_{X_6,2}$	1
$K_{S_1,3}$	1
$K_{S_1,4}$	1
$[X_0]$	1
$[X_4]$	1×10^{-12}
$[X_5]$	1×10^{-12}
$[X_6]$	1
$[X_7]$	1×10^{-12}
c	$[S_2]_{init} + [S_3]_{init}$

Table 3.1: *Parameter values used for example 1.*

We will choose Equation 3.7 as our dependent ODE. This means we can substitute $S_3 = c - S_2$ (where c is a constant equal to the sum of the initial values of S_2 and S_3) into Equation 3.2 yielding

$$v_2 = \frac{\frac{V_2}{K_{S_3,2}K_{X_6,2}}((c-[S_2])[X_6])}{1 + \frac{(c-[S_2])}{K_{S_3,2}} + \frac{[X_6]}{K_{X_6,2}} + \frac{(c-[S_2])[X_6]}{K_{S_3,2}K_{X_6,2}}} \tag{3.8}$$

Figure 3.2 shows a side by side comparison of the plots of the concentrations of the three species over time, using LSODA and CVODE, initialised with concentrations of 1, 0.7 and 0.3 for S_1, S_2, and S_3 respectively. Clearly, the results are identical, which is a reassuring confirmation that the PySUNDIALS CVODE implementation is working correctly. Source code listings of the programs that produced these two images can be examined in the Appendices.

3.1.2 A more complex model including events

Example 2 demonstrates the modeling of discontinuous events, with possibly delayed actions, using the CVODE root finding routines. Again, the reaction network is a simple one, consisting of three species (Figure 3.3). The system is described with the following equations:

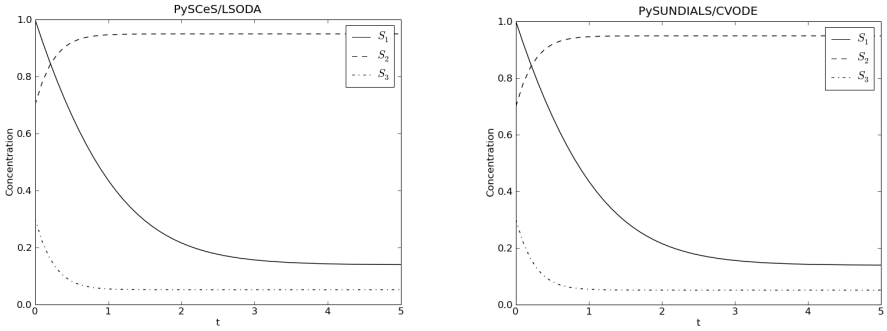

Figure 3.2: *A comparison of results between LSODA and CVODE for example 1.* LSODA results were produced using PySCeS, from Listings A.1 and A.2. CVODE results produced from Listing A.3

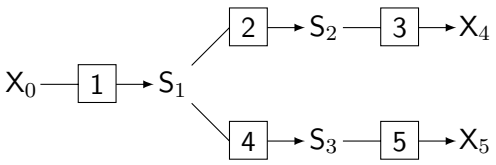

Figure 3.3: *Example 2 - A branched chain.*

$$v_1 = \frac{\frac{V_1}{K_{X_0,1}}\left([X_0] - \frac{[S_1]}{K_{eq,1}}\right)}{1 + \frac{[X_0]}{K_{X_0,1}} + \frac{[S_1]}{K_{S_1,1}}} \tag{3.9}$$

$$v_2 = \frac{\frac{V_2}{K_{S_1,2}}\left([S_1] - \frac{[S_2]}{K_{eq,2}}\right)}{1 + \frac{[S_1]}{K_{S_1,2}} + \frac{[S_2]}{K_{S_2,2}}} \tag{3.10}$$

$$v_3 = \frac{\frac{V_3}{K_{S_2,3}}\left([S_2] - \frac{[X_4]}{K_{eq,3}}\right)}{1 + \frac{[S_2]}{K_{S_2,3}} + \frac{[X_4]}{K_{X_4,3}}} \tag{3.11}$$

$$v_4 = \frac{\frac{V_4}{K_{S_1,4}}\left([S_1] - \frac{[S_3]}{K_{eq,4}}\right)}{1 + \frac{[S_1]}{K_{S_1,4}} + \frac{[S_3]}{K_{S_3,4}}} \tag{3.12}$$

Parameter	Value
V_1	8.0
V_2	2.0
V_3	2.0
V_4	4.0
V_5	4.0
$K_{X_0,1}$	1.0
$K_{S_1,1}$	1.0
$K_{S_1,2}$	1.0
$K_{S_2,2}$	1.0
$K_{S_2,3}$	1.0
$K_{X_4,3}$	1.0
$K_{S_1,4}$	1.0
$K_{S_3,4}$	1.0
$K_{X_6,5}$	1.0
$K_{S_3,5}$	1.0
$K_{eq,1}$	1.0
$K_{eq,2}$	1.0
$K_{eq,3}$	1.0
$K_{eq,4}$	1.0
$K_{eq,5}$	1.0
$[X_0]$	5.0
$[X_4]$	0
$[X_6]$	0

Table 3.2: *Parameter values used for example 2.*,

$$v_5 = \frac{\frac{V_5}{K_{S_3,5}}\left([S_3] - \frac{[X_5]}{K_{eq,5}}\right)}{1 + \frac{[S_3]}{K_{S_3,5}} + \frac{[X_5]}{K_{X_5,5}}} \tag{3.13}$$

$$\frac{dS_1}{dt} = v_1 - v_2 - v_4 \tag{3.14}$$

$$\frac{dS_2}{dt} = v_2 - v_3 \tag{3.15}$$

$$\frac{dS_3}{dt} = v_4 - v_5 \tag{3.16}$$

Starting with the parameters listed in Table 3.2, the committing step of the X_5 (lower) branch has a higher rate (v_4) than that of the X_4 upper branch (v_2). At $t = 2$ two events trigger. The first event sets both V_2 and V_3 to 4, which quickly equalises v_2 and v_4. The second event is delayed by a second before it takes effect, and at $t = 3$, V_4 and V_5 are both set to 2. The resulting concentration and rate plots can be seen in Figure 3.4.

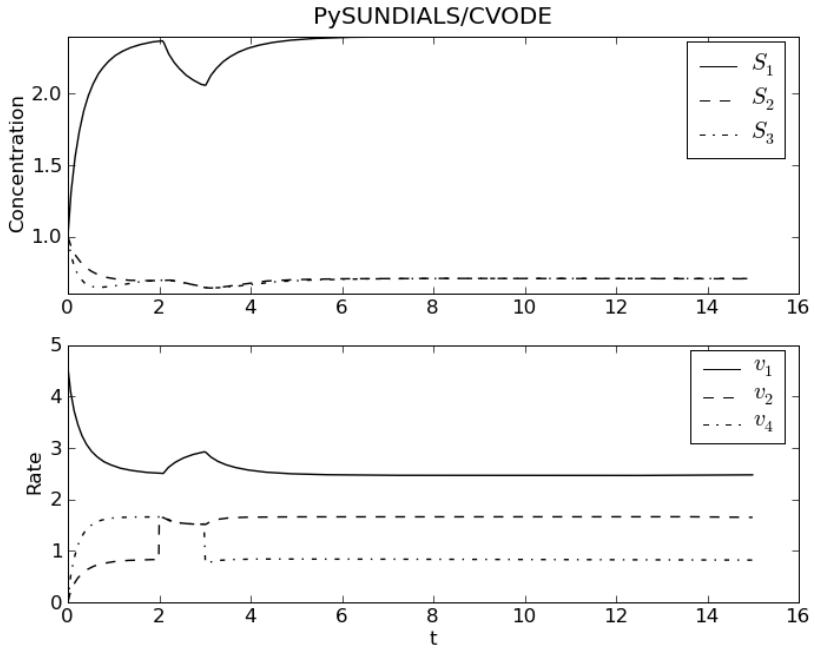

Figure 3.4: *Concentration and rates over time for example 2.* Results produced by Listing A.4.

3.1.3 A real world biological model

Example 3 is model number 96 from the biomodels database (Le Novre et al., 2006). It is a model of the circadian clock of Arabidopsis (Zeilinger et al., 2006). A PySCeS model file was created and run. This is our first example of using PySUNDIALS from within PySCeS. PySCeS detects the presence of events in the model file, and automatically selects CVODE as the back end integrator. The biomodels reference image was created using MathSBML. The results were compared to the reference results on biomodels, which can be seen at `http://www.ebi.ac.uk/biomodels-main/simulation-result.do?uri=publ-model.do&mid=BIOMD0000000096`. For convenience the reference image is given in Figure 3.5, alongside the results from PySCeS, although the image quality is quite poor. The oscillation evident is caused by two cycling events. Event one triggers when time (in hours) exceeds the number of hours at the end of the current day, i.e. 24 for day 1, 48 for day 2. Event one advances the current day by one, and turns on light. Event two triggers when time is between "midday" and "midnight", i.e. time modulus 12 is between 12 and 24. Event two simply turns off light.

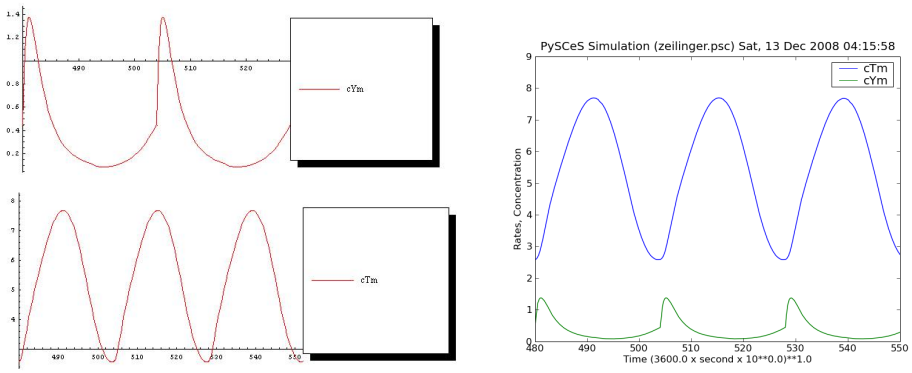

Figure 3.5: *Comparing biomodels reference image and PySCeS for example 3.* The biomodels reference result is in the left pane, and the PySCeS result is in the right pane. The PySCeS/PySUNDIALS results match the reference results. Right pane results produced from Listings A.5 and A.6.

3.2 Using CVODES to analyse changes in sensitivities in transient states

The CVODES module adds sensitivity analysis capabilities to CVODE, and we use it to analyse the sensitivities of two systems.

3.2.1 Analysing sensitivities prior to the achievement of a steady state

We use example 1 described in Section 3.1.1 for our first demonstration of sensitivity analysis. We plot the sensitivities of S_1 and S_2 to changes in the parameter V_2. Parameters remain identical to those in Table 3.1. Results can be seen in Figure 3.6.

3.2.2 Analysing sensitivities in oscillatory systems

To further illustrate the convenience and power of CVODES, we will examine an autocatalytic, oscillating chemical reaction. *Example 4*, the "Brusselator", is a well studied, classic oscillatory system (Lefever et al., 1988). A schematic representation of the system can be seen in Figure 3.7, and it is mathematically described with the following set of equations:

$$v_1 = k_1 * A \tag{3.17}$$

$$v_2 = k_2 * [X]^2 * [Y] \tag{3.18}$$

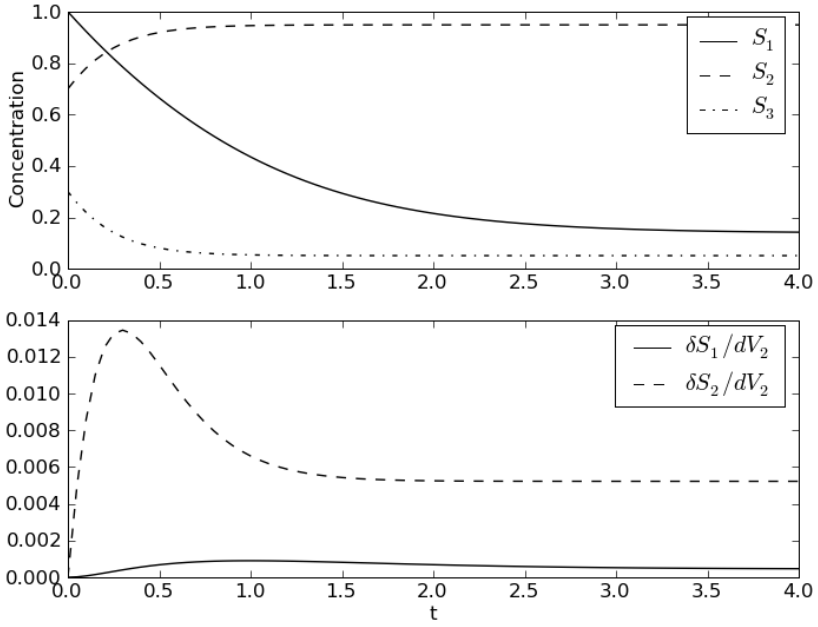

Figure 3.6: *Species sensitivities to variation in V_2 against time for example 1.* Results produced from Listing A.7.

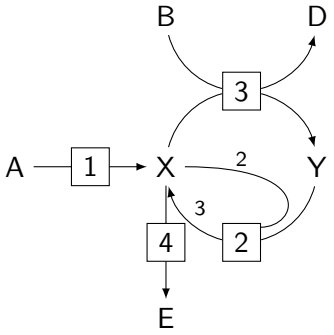

Figure 3.7: *Example 4 - The brusselator.*

$$v_3 = k_3 * [X] * B \tag{3.19}$$

$$v_4 = k_4 * [X] \tag{3.20}$$

$$\frac{dX}{dt} = v_1 + v_2 - v_3 - v4 \tag{3.21}$$

$$\frac{dY}{dt} = v_3 - v_2 \tag{3.22}$$

A, B, D, and E are constant concentrations and are set at 0.5, 3.0, 0, and 0 respectively. The results shown in Figure 3.8 were produced with k_1 through k_4 at values of 1.

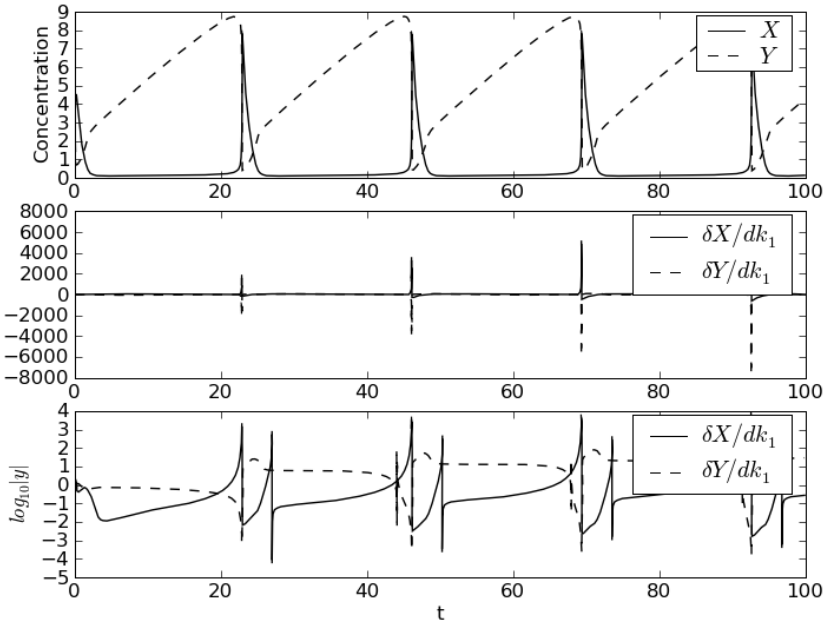

Figure 3.8: *Species sensitivities to variation in k_1 against time in example 3.* The top panel depicts concentration time courses. The middle panel shows sensitivities of concentrations over time to variation in k_1. The bottom panel depicts the same sensitivities on a $log_{10}|y|$ axis. Values on this axis have their absolute values logged, and then their original sign restored. Using this transformation allows us to visualise the trends of small scale changes in sensitivities otherwise masked by the extreme peaks and troughs of the middle panel. Results produced by Listing A.8.

The period of oscillation is dependent on k_1. This explains the slow increase in sensitivities over time, as the two systems (each with a different value for k_1) diverge. Our results demonstrate the usefulness of time dependent sensitivity analysis as a tool for the analysis for systems which do not reach a steady state, especially oscillatory systems where period is influenced by parameters.

3.3 IDA examples

3.3.1 Using IDA to solve a system with a moiety-conserved cycle

To illustrate the use of IDA for solving DAE systems where moiety conservations are represented by algebraic equations, we once again turn to example 1. However, we use a three species system and Equation 3.2 instead of Equation 3.8. We also add the implicit algebraic equation specifying S_3

$$0 = S_2 + S_3 - c \tag{3.23}$$

IDA requires that all equations, including the ODEs, be specified implicitly. Hence, Equations 3.5 and 3.6 therefore become

$$0 = v_1 - v_3 - v_4 - \frac{dS_1}{dt} \tag{3.24}$$

$$0 = v_2 - v_1 - \frac{dS_2}{dt} \tag{3.25}$$

Figure 3.9 shows identical results to our initial CVODE example.

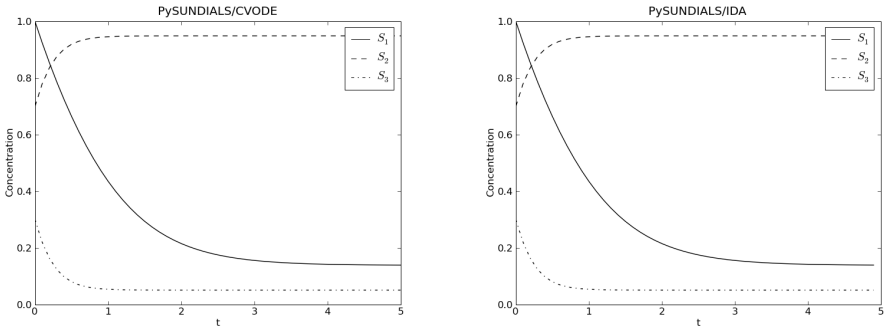

Figure 3.9: *Comparison of CVODE and IDA concentrations over time for example 1.* IDA produces identical results to CVODE with a reduced ODE system.

3.3.2 Using IDA to solve systems with equilibrium blocks

For the second IDA example (*Example 5*) we introduce a new model consisting of a simple three species linear reaction chain (See Figure 3.10). Reactions one and four use reversible Michaelis-Menten kinetics, but reactions two and three form an equilibrium block. This forms a system of differential and algebraic equations, described as follows:

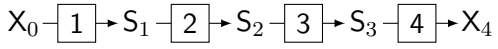

Figure 3.10: *Example 5 - A linear chain with an equilibrium block formed from reactions 2 and 3.*

Parameter	Value
V_1	1.0
$K_{X_0,1}$	1.0
$K_{S_1,1}$	1.0
$K_{eq,1}$	1.0
V_4	1.0
$K_{S_3,4}$	1.0
$K_{X_4,4}$	1.0
$K_{eq,4}$	1.0
X_0	1.0
X_4	1×10^{12}
$K_{eq,2}$	0.9
$K_{eq,3}$	0.75

Table 3.3: *Parameter values used for example 5.*

$$v_1 = \frac{\frac{V_1}{K_{X_0,1}}\left([X_0] - \frac{[S_1]}{K_{eq,1}}\right)}{1 + \frac{[X_0]}{K_{X_0,1}} + \frac{[S_1]}{K_{S_1,1}}} \tag{3.26}$$

$$v_4 = \frac{\frac{V_4}{K_{S_3,4}}\left([S_3] - \frac{[X_4]}{K_{eq,4}}\right)}{1 + \frac{[S_3]}{K_{S_3,4}} + \frac{[X_4]}{K_{X_4,4}}} \tag{3.27}$$

$$0 = v_1 - v_4 - \frac{dS_1}{dt} \tag{3.28}$$

$$0 = S_1 \times K_{eq,2} - S_2 \tag{3.29}$$

$$0 = S_2 \times K_{eq,3} - S_3 \tag{3.30}$$

Figure 3.11 shows the concentration levels over time for example 5. For the purposes of comparison we also present the same system as run in CVODE, but with the algebraic equations removed, and used only post solution to generate

values for S_2 and S_3. This reduced system consists of the same single ODE, but Equation 3.27 is changed to

$$v_4 = \frac{\frac{V_4}{K_{S_3,4}}\left([S_1]K_{eq,2}K_{eq,3} - \frac{[X_4]}{K_{eq,4}}\right)}{1 + \frac{[S_1]K_{eq,2}K_{eq,3}}{K_{S_3,4}} + \frac{[X_4]}{K_{X_4,4}}} \tag{3.31}$$

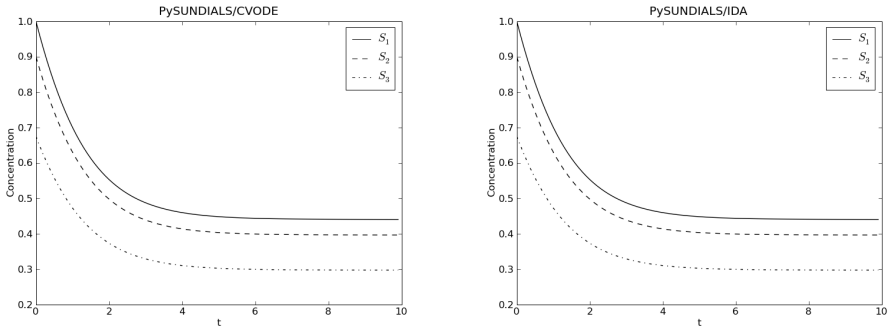

Figure 3.11: *Comparison of time courses for example 5,* demonstrating identical results between CVODE and IDA.

The results are as expected. The IDA results match the CVODE results. It can clearly be seen that S_2 and S_3 maintain their respective equilibrium relationships throughout the time course.

While the demonstration of identical results between the CVODE and IDA solutions might lead one to think IDA is unnecessary, it should be noted that the recasting of a system to remove algebraic equations is not necessarily trivial, or always possible to automate. Methods exist to detect conservation relationships, but these methods do not identify equilibrium relationships. It may be easy for the human eye to differentiate an algebraic equation from a differential, but this is not the case for a computer program. Thus IDA is appealing, in that it obviates the need for the human modeler to recast their models for use without algebraic equations.

3.4 A KINSOL example

Following the pattern established, we use *Example 1* as our first KINSOL example. Identical parameters are used. Being a nonlinear solver, the resulting output is a listing of vector of steady state concentrations and derived fluxes. We compare the results of using KINSOL (Listing 3.1), to the results obtained

by using HYBRD (Listing 3.2[1]), the current default nonlinear solver of PySCeS. Code listings for this example can be found in Listings A.12 and A.13.

Listing 3.1: Output of moiety_branch-kinsol.py

```
Steady-state species concentrations
S2_ss = 9.4882e-01
S1_ss = 1.3858e-01
S3_ss = 5.1179e-02

Steady-state fluxes
J_R1 = 2.4343e-01
J_R2 = 2.4343e-01
J_R3 = 1.2172e-01
J_R4 = 1.2172e-01
```

Listing 3.2: Output of moiety_branch-hybrd.py

```
Steady-state species concentrations
S2_ss = 9.4882e-01
S1_ss = 1.3859e-01
S3_ss = 5.1179e-02

Steady-state fluxes
J_R1 = 2.4343e-01
J_R2 = 2.4343e-01
J_R3 = 1.2172e-01
J_R4 = 1.2172e-01
```

Modifying Listing A.12 by adding a simple loop and some plotting code we can perform parameter scans, the results of which are shown in Figure 3.12.

3.5 Benchmark comparisons

The SUNDIALS source is distributed with a comprehensive selection of example programs for parallel, serial and FORTRAN implementations. Thirty-three of these examples have been translated into Python for use with PySUNDIALS, and they collectively serve as both a test suite for PySUNDIALS and as example code for potential PySUNDIALS users to learn from, or adapt to their own uses. The thirty-three examples that have been translated include every SUNDIALS example using the serial `N_Vector` and written exclusively in C. The FORTRAN and mixed C and FORTRAN examples were not translated; the FORTRAN interface of SUNDIALS is not wrapped by PySUNDIALS, and the underlying C code suffices. The parallel `N_Vector` based examples will be translated when the parallel `N_Vector` has been successfully wrapped. We have used these translated examples as a benchmark suite as they provide a comprehensive usage of SUNDIALS features, and are cross-disciplinary in nature. They also form a sample set large enough to allow us to draw conclusions about behaviour and performance.

[1]Only pertinent output has been listed, output generated by the line "`import pysces`" has been removed

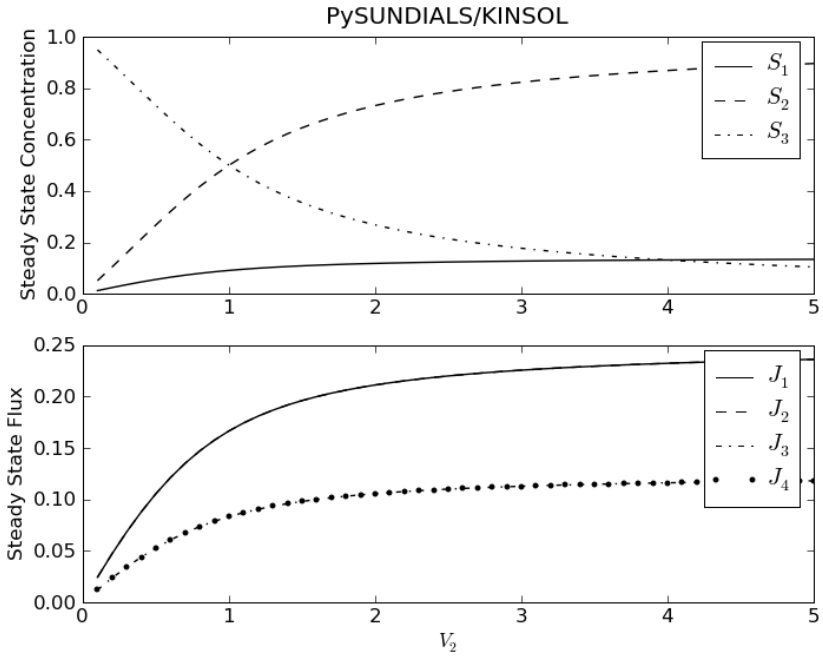

Figure 3.12: *Parameter scan of steady state concentrations and fluxes of example 1 varying V_2 from 0 to 5. Results produced by Listing A.14*

3.5.1 Size and performance comparison of C and Python examples

Each example program was run as both the compiled C version and the Python version, and their execution timed. Each example was run in series on the same machine. The machine used for benchmarking was an AMD Opteron Dual Core 64-bit CPU with 1Gb memory, running at 2.2Ghz per core. The machine was left undisturbed throughout the benchmarking process, which used only one core, leaving system and background services to use the second core, so as to interfere as little as possible with the process. Despite these efforts, execution times presented are meant to be taken in a relative context, and are not necessarily indicative of execution times on any other system. The results of the benchmarking are presented in Table 3.4.

From these results it seems that the C source code examples are on average three times longer than their Python equivalents but run nearly 300 times faster, and in the worst cases a whole three orders of magnitude faster. Closer inspection reveals a different story. The ratios of code length form a reasonably normal distribution, so we can accept the mean reduction in code length when coding in Python, but the same cannot be said of the ratios of execution

	Execution Time (s)			Code Length (lines)		
	C	Python	Python/C	C	Python	C/Python
cvbanx	0.019	0.332	17.474	438	139	3.15
cvdenx	0.016	0.245	15.312	370	73	5.07
cvdenx_uw	0.016	0.263	16.438	393	84	4.68
cvdirectdem	0.046	1.431	31.109	777	352	2.21
cvkrydem_lin	0.308	27.07	87.890	749	339	2.21
cvkrydem_pre	0.632	151.308	239.411	1136	578	1.97
cvkryx	0.101	7.175	71.040	534	300	1.78
cvkryx_bp	0.217	12.01	55.346	660	227	2.91
cvsadjbanx	0.723	6.287	8.696	551	231	2.39
cvsadjdenx	0.032	0.549	17.156	662	200	3.31
cvsadjkryx_int	14.899	5531.52	371.268	1348	605	2.23
cvsadjkryx_pnt	17.936	2497.13	139.225	1346	611	2.20
cvsbanx	0.02	0.333	16.650	438	135	3.24
cvsdenx	0.017	0.251	14.765	370	73	5.07
cvsdenx_uw	0.017	0.271	15.941	393	84	4.68
cvsdirectdem	0.045	1.48	32.889	777	352	2.21
cvsfwddenx	0.021	0.565	26.905	622	235	2.65
cvsfwdkryx	0.779	76.511	98.217	922	415	2.22
cvsfwdnonx	0.04	1.61	40.250	517	209	2.47
cvskrydem_lin	0.309	26.216	84.841	750	339	2.21
cvskrydem_pre	0.632	148.111	234.353	1136	578	1.97
cvskryx	0.102	7.197	70.559	534	301	1.77
cvskryx_bp	0.22	11.17	50.773	660	227	2.91
idabanx1	0.023	0.941	40.913	404	135	2.99
idabanx2	1.612	153.451	95.193	671	245	2.74
idadenx	0.016	0.265	16.562	399	98	4.07
idakrydem_lin	0.027	1.348	49.926	586	193	3.04
idakryx	0.021	0.765	36.429	547	186	2.94
kinbanx	0.04	1.171	29.275	394	112	3.52
kindenx1	0.014	0.215	15.357	492	168	2.93
kindenx2	0.014	0.2	14.286	434	162	2.68
kinkrydem_lin	0.073	271.4	3717.808	892	330	2.70
kinkryx	0.083	265.326	3196.699	791	294	2.69
			mean = 271.786			mean = 2.903

Table 3.4: *Comparative execution times and code lengths between C and Python.*

times. There are some clear outliers in this set. Glancing over the table, we notice that any example using Krylov techniques appears to have a high execution times in Python. As the C and Python versions of the example algorithms are identical, this is quite surprising, and indeed it is not the use of Krylov techniques that causes such excessive execution times, but rather the complexity of the callback functions. Specifically, those examples that perform additional function calls, or looping within their right-hand side or preconditioning callback functions suffer heavily. This is due in large part to Python's inherent cost in calling functions or using variables outside the local scope, both of which cause a non-constant time lookup. For simple cases, the performance difference is more in the region of a single order of magnitude, rather than the two to three orders of magnitude for average to worst case. Still, the performance penalty of using PySUNDIALS remains a real concern, and solutions need to be sought.

3.5.2 Contrasting the use of CVODE and LSODA in PySCeS

PySCeS, up to version 0.6.9, uses the SciPy interface to LSODA as its default ODE solver. PySCeS has a framework that allows new solvers to be plugged in with relative ease, and PySUNDIALS CVODE has been successfully incorporated. We will compare LSODA to CVODE in the specific context of being used as PySCeS back end solvers.

Both LSODA and CVODE, along with the other modules in ODEPACK and SUNDIALS respectively, are executed in a similar manner. They both take a nominated function and evaluate it as the right hand side of an ODE system. This is done within the scaffolding of a user implemented driving loop, which steps the solver from initial state to the next desired output point along the domain of the independent variable, usually time. This method of execution allows the solvers to exit prior to the desired output point being reached in the case of error, or other notification, such as the roots of a specified function being found. The user is then able to attempt recovery, in the case of an error, or otherwise handle any notifications and proceed normally.

The SciPy interface to LSODA encapsulates the driving loop mechanism within a single function, removing this control from the user, or in the case of PySCeS, from the controlling program. While the SciPy interface allows a set of desired output points to be specified, no advanced functionality, such as root finding or error recovery can be implemented via this interface. One additional implication of this interface revolves around required maximum step sizes. If a single step in a particular solution requires an increase in step size, due to failure to converge, it only becomes apparent when the entire solution fails. At this point, the maximum step size must be increased for *all steps*, and the solution from scratch. This incurs a heavy performance penalty.

PySUNDIALS leaves the driving loop to the user or controlling program to construct and execute. This allows access to features such as root finding, and

the adjustment of step size for a single time step, without having to restart the solution entirely. Once difficult steps have been passed, maximum step size can again be reduced to optimal levels, speeding up the solution again.

3.6 The Whole Cell Model: A complex example

As our last example of CVODE use, we will reproduce the minimal whole cell model by Surovstev et al. (2007) (See Figure 3.13 for a schematic representation). This model is an attempt to simulate cell division as an internally produced mechanism, as opposed to modelling cell triggered by an event set by the modeller.

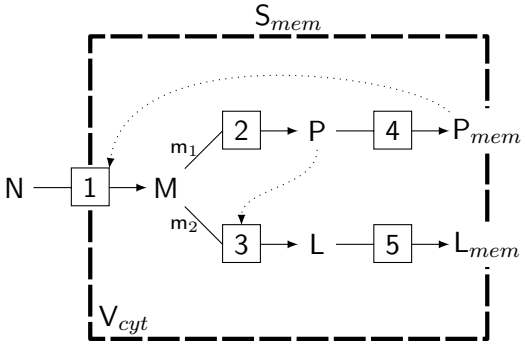

Figure 3.13: *The minimal whole cell model.* N represents nutrients external to the cell. M is the metabolome, P is the proteome and L is the lipidome. P_{mem} and L_{mem} are those parts of the proteome and lipidome embedded in the cell membrane. V_{cyt} is the cytosolic volume. S_{mem} is the surface area of the cell membrane.

At first we attempt to reproduce the results using PySCeS and LSODA (See Listings A.16 and A.15). LSODA fails. Though rare, there are some models which LSODA cannot handle, but CVODE can, and this appears to be one of them. Next the model was recast into an amount based system, rather than the original concentration based system, and was hand coded using the PySUNDIALS CVODE module in an attempt to reproduce the results of Surovstev et al. (2007). We refer to figures 4a and 4c of the Surovstev paper as a means to provide a rapid visual method of comparison between results. Our results are presented in Figures 3.15 and 3.17 respectively. Parameters, initial values and equations can be found in Listing A.17. A complete set of parameters could not be found in the paper itself, or the supplementary materials, as a significant portion of the paper is dedicated to parameter discovery and optimisation. The parameters used were were obtained by communication with the authors.

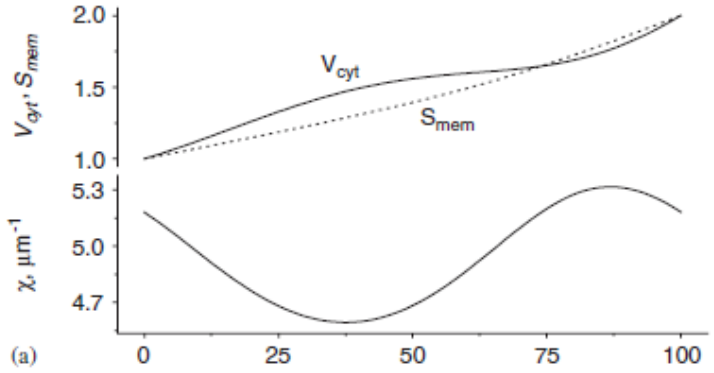

Figure 3.14: *Reference figure 4a from Surovstev et al. (2007).*

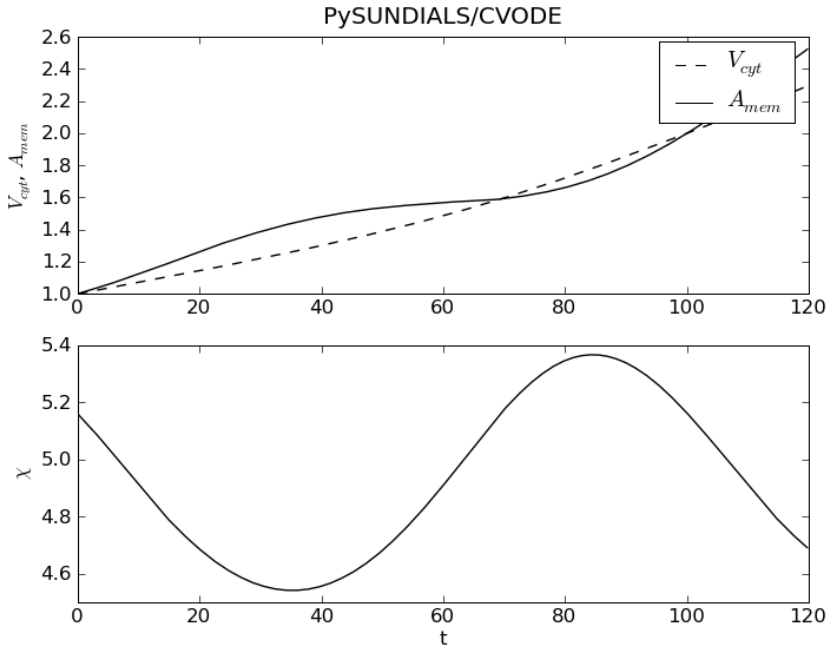

Figure 3.15: *V_{cyt}, S_{mem}, and χ displaying an oscillatory growth pattern.* This figure is a reproduction of Figure 3.14. Cytosolic volume (V_{cyt}) and Cell membrane (S_{mem}) surface area are scaled against their initial values, hence their increasing tendency over time. $V_{cyt} = 1$ represents the unitary volume of a single cell. χ is the ratio between V_{cyt} and S_{mem}. Results produced by Listing A.17.

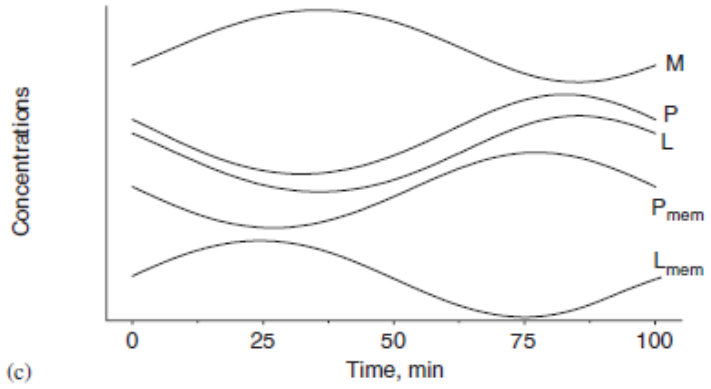

Figure 3.16: *Reference figure 4c from Surovstev et al. (2007).* Each plot is on its own y-axis.

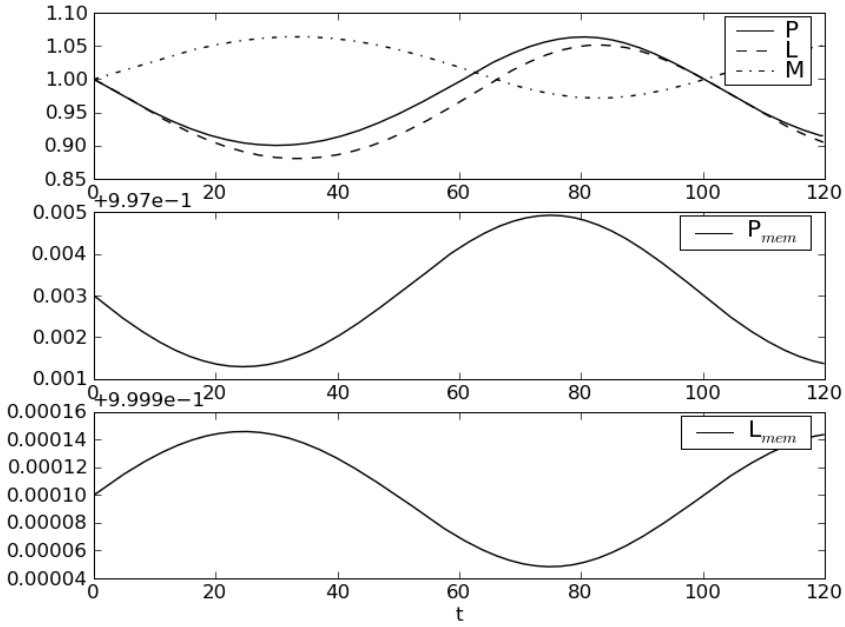

Figure 3.17: *Oscillatory growth shown in P, L, M, P_{mem} and L_{mem}.* This figure is a reproduction of Figure 3.16 using multiple three separate y-axes, which is sufficient to visually depict oscillation in all five species. Results produced by Listing A.17.

Chapter 4

Discussion and conclusion

4.1 The integration of SUNDIALS into PySCeS

The integration of PySUNDIALS into PySCeS has already started. As of PySCeS release 0.6.9, users can select CVODE as a back end integrator in preference to LSODA, although LSODA remains the default choice, except where models specify events, multiple compartments or rate rules.

Listing 4.1: Using CVODE in PySCeS

```
>>> import pysces
.
.
.
>>> m = pysces.model('branched_moiety')
>>> m.doLoad()

Parsing file: /home/sirlark/Pysces/psc/moiety_branch.psc
Info: "X4" has been initialised but does not occur in a rate equation
Info: "X5" has been initialised but does not occur in a rate equation
Info: "X7" has been initialised but does not occur in a rate equation

 Calculating L matrix . . . . . .  done.
 Calculating K matrix . . . . . .  done.


>>> m.mode_integrator
'LSODA'
>>> m.mode_integrator = 'CVODE'
>>> m.doSimPlot()
CVODE time for 21 points: 0.0830161571503
ylist empty plotting all data vs (0)

>>>
```

The PySCeS input file syntax has been extended to support time dependent event specification, following the framework described in the SMBL L2V1 specification. Each event must be given an identifier, a trigger condition, a delay time, and specify what should be done, in the form of a set of assignments. The assignments are executed when the trigger condition is true, or

in the case of a nonzero delay, when the specified amount of time has elapsed since the trigger condition was true.

Listing 4.2: Event syntax in PySCeS model files

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

Presently the PySCeS developers are working on the inclusion of IDA, with KINSOL and CVODES integration to follow. Although PySCeS already has mechanisms for handling systems of differential and algebraic equations, the integration of the PySUNDIALS IDA module offers the potential to make dealing with systems much simpler. Presently, moieties are identified, and only independent ODEs are used during system solution. For equilibrium blocks, forcing functions must be used, and PySCeS does not currently support SBML constraints. The use of a DAE solver such as IDA would obviate the need for these workarounds.

4.2 Future work

4.2.1 Exploring various technologies as optimization techniques

The poor performance of PySUNDIALS needs to be addressed. Profiling the slower running PySUNDIALS examples indicates that the `NVector` class would be a good candidate for optimization, especially the subscription methods, as most PySUNDIALS programs spend a significant amount of time manipulating `NVector`s. The subscription methods are already as optimal as they can be made in pure Python, and the next step would involve refactoring the `NVector` class into a Python extension module coded in pure C. The PySUNDIALS examples make no use of NumPy though, which distorts the profile results somewhat. It is expected that PySUNDIALS will be used in conjunction with NumPy more often than not. Using NumPy means that getting and setting individual elements of `NVector`s via their subscription methods occurs far less often, most operations being performed at the whole vector or matrix level. A NumPy matrix multiplication into a NumPy array sharing memory with an `NVector` is far faster than assigning each element of the `NVector` a value in turn.

Still, the major areas of concern lie not in PySUNDIALS itself, but rather in the user defined callback functions. This leaves much of the burden of optimization on the user, although a number of technologies exists that might ease this burden.

4.2.1.1 The psyco module

Of most interest is the Python module psyco (`http://psyco.sourceforge.net`), which functions somewhat like a just-in-time (JIT) compiler for Python code. psyco can be used to selectively optimize blocks of code, as opposed to the entire program. With only a few extra lines of Python code, approximately one per function (or code block) to be optimized, standard Python code can be modified to use psyco. Often optimizing an entire program creates so much overhead, that the performance gains are obviated. The performance gains from doing this are highly dependent on the nature of the code being optimized, as some code constructs are more easily optimized than others, and some not at all. The psyco documentation claims performance gains are in the range of one to two orders of magnitude (`http://psyco.sourceforge.net/introduction.html`), although in our experience, albeit limited to very cursory investigation, performance gains are more often in the range of 3 to 8 fold. The psyco module can also function in tandem with a Python profiler, dynamically identifying code bottlenecks during runtime, and selecting those code blocks as candidates for optimization. We have not tested this technique yet. Of course optimization always comes at a trade off, and in psyco's case, the trade off is speed for memory. Psyco's memory overhead is admittedly high, proportional to the amount of Python code being rewritten by psyco. Further disadvantages are that pscyo only works on intel i386 or compatible CPU's, and that it is not a standard Python module. The nature of psyco and Python however makes it possible for PySUNDIALS to use psyco if available, with relatively minimal code modification.

4.2.1.2 PyInline, weave, and instant

PyInline (`http://pyinline.sourceforge.net`), weave and instant all work in a similar manner, facilitating the inclusion of small sections of code written in other languages, into a Python program. As such, they all suffer from the drawback that their use requires the user of PySUNDIALS to know C. Avoiding the need to code in C was one of the reasons PySUNDIALS was developed in the first place, and so we have a catch-22 situation. Additionally, PyInline's last release was in 2001 and it seems to be a dead project, whilst instant requires SWIG. Of these three options, weave, being part of SciPy, remains the most attractive. Despite the requirement of coding in C using one of these technologies being seen as a drawback, the amount of code which would be required in C would be so small that they would be likely to avoid the more complicated parts of C coding, namely dynamic memory management and error handling. Coding a simple loop in C is not significantly more complex than the same in Python, and if called often can provide a significant performance boost.

4.2.1.3 Pure Python optimization techniques

Ultimately, the user of PySUNDIALS is left to make sure their callback functions run optimally. Guido van Rossum, Python's creator, has written a short but very helpful essay (`http://www.python.org/doc/essays/list2str`) covering optimization techniques in the Python language specifically. It is likely that many of these pointers will be absorbed into the PySUNDIALS documentation, after being recast into examples more specific to PySUNDIALS. As always, the general guidelines for optimization are language independent and work in Python too:

- Don't optimize unless you know you need to, and where you need to.
- A better algorithm will always outperform an optimized slower algorithm.
- Unroll your loops and inline your code

The last is especially helpful in Python, as function calls are very expensive due to the function name lookup cost.

4.2.2 Wrapping the parallel `N_Vector` structure

A portion of SUNDIALS remains unwrapped. The SUNDIALS `N_Vector` exists in both serial and parallel form, both of which implement the exact same computational functionality. Despite the lack of a parallel `N_Vector` wrapper, the serial wrapper provides the complete set of SUNDIALS features. The SUNDIALS parallel implementation uses the Message Passing Interface (MPI) (`http://www.mpi-forum.org/`). Many Python MPI implementations exist, including:

MPI 4 Python: MPI 4 Python is part of the SciPy project, and offers a complete MPI implementation, featuring optimized support for transfer of NumPy arrays. Any picklable Python object can be sent or received. (`http://mpi4py.scipy.org`)

pypar: At first look pypar seems very simple and easy to use. It is available as a Python extension module. It is capable of passing any Python object as a message, and it handles any marshalling or serialisation required automatically. It is not a complete MPI implementation though. (`http://datamining.anu.edu.au/~ole/pypar`)

pyMPI: MPI Python, or pyMPI, is a standalone Python interpreter which incorporates "a large subset of MPI functions". This functionality is accessible through a built in module, `mpi`. (`http://pympi.sourceforge.net`)

The successful wrapping of the SUNDIALS parallel `N_Vector` requires a thorough investigation into the various merits and deficiencies of each Python MPI implementation. Once an appropriate implementation has been selected, the actual wrapper code needs to be written. One additional complexity remains, however. If two compatible `NVector` modules exist in PySUNDIALS, a method must be provided for the user to choose which one should be used. Python does not allow parameters to be passed on module import, and modules do not have access to a program's global scope, so no method exists of informing one of the higher level modules (`cvode`, `cvodes`, `ida`, or `kinsol`) which `NVector` module to use. Although not ideal, one way around this problem is to require that the higher level modules are initialised for parallel use, which would serve the dual purpose of switching from the serial `NVector` module to the parallel one, and initialising MPI.

4.3 Summary

We have argued that a Python interface to SUNDIALS is necessary, on account of SUNDIALS, and especially the CVODE module, becoming a *de facto* standard in scientific computing, gradually replacing ODEPACK and LSODA respectively. SUNDIALS is the more advanced suite, and provides the major advantages of allowing user intervention during the integration process and a better adaptive step size routine. The suite provides a variety of modules geared towards the solution of different problems but remains accessible by providing a consistently organised collection of functions and data structures. A Python interface to SUNDIALS is desirable because the C language is time consuming to code in, has a much steeper learning curve, and incurs much code overhead in the form of memory management and error handling.

PySUNDIALS provides the desired Python interface, not only to CVODE, but to the entire suite. It facilitates reduced coding times of, on average, two thirds. In addition to bare bindings, PySUNDIALS implements an interface consisting of a set of classes that act like similar Python built in objects. `NVector`s behave like lists, and matrix objects like lists of lists. PySUNDIALS is currently at the end of its beta development phase, with release candidate 1 being available for download, and is confirmed to run on both windows and unix derivative operating systems. Testing on the Apple Macintosh has been delayed pending hardware availability. PySUNDIALS will be moved out of beta once evidence of community testing and involvement becomes available, and any bugs found are resolved. At this point, it is hoped that PySUNDIALS can become a module of SciPy.

Interoperability with SciPy and NumPy is an important concern, considering that these packages are the mainstay of scientific computing in Python. PySUNDIALS provides convenient and easy interoperability mechanisms, as an `NVector` can be used as a drop in replacement for a NumPy array by the

use of its `asarray` method. NumPy arrays can be converted to `NVector`s with equal ease. As NumPy arrays are the foundational data type of SciPy, these two conversion mechanisms provide all that is required to use SciPy, NumPy and PySUNDIALS in conjunction with one another.

Using PySUNDIALS, the capabilities of PySCeS have been extended, and they now include SMBL events. With the integration of the PySUNDIALS IDA module, differential and algebraic equation systems and SMBL constraints will become features of PySCeS, leading to a more feature rich, robust software tool for computational systems biology. The integration of the rest of PySUNDIALS will provide time dependent sensitivity analysis capabilities through CVODES and an additional robust nonlinear solver in the form of KINSOL.

Appendix A

Code listings

Note that some listings may have been altered to fit print format. Where line continuations were required, trailing backslashes are used. In Python code this does not affect the running of the code, but in PySCeS model files it may.

Listing A.1: moiety_branch.psc

```
#Small system with a branch and a moiety using simplified kinetics
#From Hofmeyr (2001)

FIX: X0 X4 X5 X6 X7

R1:
    X0 + S2 = S1 + S3
    V1*X0*S2/K1_X0*K1_S2/(1 + X0/K1_X0 + S2/K1_S2 + X0*S2/K1_X0*K1_S2)

R2:
    X6 + S3 = X7 + {1.0}S2
    V2*S3*X6/K2_S3*K2_X6/(1 + S3/K2_S3 + X6/K2_X6 + S3*X6/K2_S3*K2_X6)

R3:
    S1 = X4
    V3*S1/(S1 + K3_S1)


R4:
    {1.0}S1 = X5
    V4*S1/(S1 + K4_S1)

#External Metabolites
X0 = 1.0
X4 = 0.0
X5 = 0.0
X6 = 1.0
X7 = 0.0

#System Paramters
V1 = 1.0
V2 = 10.0
V3 = 1.0
V4 = 1.0
K1_X0 = 1.0
K1_S2 = 1.0
K2_S3 = 1.0
```

```
38 K2_X6 = 1.0
39 K3_S1 = 1.0
40 K4_S1 = 1.0
41
42 #Initial Concentrations
43 S1 = 1.0
44 S2 = 0.7
45 S3 = 0.3
```

Listing A.2: moiety_branch-lsoda-independent.py

```
 1 import pysces
 2 from matplotlib import pyplot
 3
 4 m = pysces.model('moiety_branch')
 5 m.doLoad()
 6 m.doSim(end=5, points=51)
 7
 8 pyplot.title('PySCeS/LSODA')
 9 pyplot.xlabel('t')
10 pyplot.ylabel('Concentration')
11 results = m.data_sim.species.transpose()
12 pyplot.plot(
13         m.sim_time, results[1], 'k-',
14         m.sim_time, results[0], 'k--',
15         m.sim_time, results[2], 'k-.'
16 )
17 pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
18 pyplot.show()
```

Listing A.3: moiety_branch-cvode-independent.py

```
from pysundials import cvode
import ctypes
from matplotlib import pyplot

V1 = 1.0
V2 = 10.0
V3 = 1.0
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
X0 = 1.0
X4 = 1.0e-12
X5 = 1.0e-12
X6 = 1.0
X7 = 1.0e-12

S1 = 0
S2 = 1
#S3 = 2

def R1(y):
        return (V1/(K1_X0*K1_S2)*(X0*y[S2]))/\
                (1 + X0/K1_X0 + y[S2]/K1_S2 +\
                (X0*y[S2])/(K1_X0*K1_S2))

def R2(y):
        return (V2/(K2_S3*K2_X6))*((icsum-y[S2])*X6)/\
                (1 + (icsum-y[S2])/K2_S3 + X6/K2_X6\
                + (((icsum-y[S2])*X6)/(K2_S3*K2_X6)))

def R3(y):
        return (V3*y[S1])/(y[S1] + K3_S1)

def R4(y):
        return (V4*y[S1])/(y[S1] + K4_S1)

def f(t, y, ydot, f_data):
        ydot[S2] = R2(y) - R1(y)
        ydot[S1] = R1(y) - R3(y) - R4(y)

        return 0

y = cvode.NVector([1.0, 0.7])
icsum = 0.7 + 0.3 #setting the initial sum of the conservation

cvode_mem = cvode.CVodeCreate(cvode.CV_BDF, cvode.CV_NEWTON)
cvode.CVodeMalloc(cvode_mem, f, 0.0, y, cvode.CV_SS, 1.0e-8, 1.0e-12)
cvode.CVDense(cvode_mem, 2)

results = ([], [], [], [])

t = cvode.realtype(0.0)
results[0].append(t.value)
results[1].append(y[S1])
results[2].append(y[S2])
results[3].append((icsum-y[S2]))

```

```
62  iout = 1
63  tout = 0.1
64  while iout <= 50:
65          ret = cvode.CVode(
66                  cvode_mem,
67                  tout,
68                  y,
69                  ctypes.byref(t),
70                  cvode.CV_NORMAL
71          )
72          if ret != 0:
73                  print "CVODE␣Error:␣%i"%(ret)
74                  break
75          results[0].append(tout)
76          results[1].append(y[S1])
77          results[2].append(y[S2])
78          results[3].append((icsum-y[S2]))
79
80          iout += 1
81          tout += 0.1
82
83  pyplot.title('PySUNDIALS/CVODE')
84  pyplot.xlabel('t')
85  pyplot.ylabel('Concentration')
86  pyplot.plot(
87          results[0], results[1], 'k-',
88          results[0], results[2], 'k--',
89          results[0], results[3], 'k-.'
90  )
91  pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
92  pyplot.show()
```

Listing A.4: branched_events-cvode.py

```
from pysundials import cvode
import ctypes
from matplotlib import pyplot

V1 = 8.0
K1_X0 = 1.0
K1eq = 1.0
K1_S1 = 1.0

V2 = 2.0
K2_S1 = 1.0
K2eq = 1.0
K2_S2 = 1.0

V3 = 2.0
K3_S2 = 1.0
K3eq = 1.0
K3_X4 = 1.0

V4 = 4.0
K4_S1 = 1.0
K4eq = 1.0
K4_S3 = 1.0

V5 = 4.0
K5_S3 = 1.0
K5eq = 1.0
K5_X6 = 1.0

E1_Trigger_t = 2
E2_Trigger_t = 2
E2_Delay_t = 1
E2_Activate_t = None

X0 = 5.0
X4 = 0
X6 = 0

S1 = 0
S2 = 1
S3 = 2

def R1(y):
        return (V1/K1_X0)*(X0-(y[S1]/K1eq))/\
                (1+(X0/K1_X0)+(y[S1]/K1_S1))

def R2(y):
        return (V2/K2_S1)*(y[S1]-(y[S2]/K2eq))/\
                (1+(y[S1]/K2_S1)+(y[S2]/K2_S2))

def R3(y):
        return (V3/K3_S2)*(y[S2]-(X4/K3eq))/\
                (1+(y[S2]/K3_S2)+(X4/K3_X4))

def R4(y):
        return (V4/K4_S1)*(y[S1]-(y[S3]/K4eq))/\
                (1+(y[S1]/K4_S1)+(y[S3]/K4_S3))

def R5(y):
        return (V4/K5_S3)*(y[S3]-(X6/K5eq))/\
                (1+(y[S3]/K5_S3)+(X6/K5_X6))
```

```
62
63  def f(t, y, ydot, f_data):
64          ydot[S1] = R1(y)-R2(y)-R4(y)
65          ydot[S2] = R2(y)-R3(y)
66          ydot[S3] = R4(y)-R5(y)
67
68          return 0
69
70  def g(t, y, gout, g_data):
71          gout[0] = t - E1_Trigger_t
72          gout[1] = t - E2_Trigger_t
73          if E2_Activate_t is not None:
74                  gout[2] = t - E2_Activate_t
75          else:
76                  gout[2] = 1
77          return 0
78
79  y = cvode.NVector([1.0, 1.0, 1.0])
80
81  cvode_mem = cvode.CVodeCreate(cvode.CV_BDF, cvode.CV_NEWTON)
82  cvode.CVodeMalloc(cvode_mem, f, 0.0, y, cvode.CV_SS, 1.0e-6, 1.0e-8)
83  cvode.CVodeRootInit(cvode_mem, 3, g, None)
84  cvode.CVDense(cvode_mem, 3)
85
86  results = ([], [], [], [], [], [], [])
87
88  t = cvode.realtype(0.0)
89  results[0].append(t.value)
90  results[1].append(y[S1])
91  results[2].append(y[S2])
92  results[3].append(y[S3])
93  results[4].append(R1(y))
94  results[5].append(R2(y))
95  results[6].append(R4(y))
96
97  iout = 1
98  tout = 0.01
99  while iout <= 1500:
100         ret = cvode.CVode(
101                 cvode_mem,
102                 tout,
103                 y,
104                 ctypes.byref(t),
105                 cvode.CV_NORMAL
106         )
107         if ret == cvode.CV_ROOT_RETURN:
108                 roots = cvode.CVodeGetRootInfo(cvode_mem, 3)
109                 if roots[0] == 1:
110                         V2 = 4
111                         V3 = 4
112                 if roots[1] == 1:
113                         E2_Activate_t = t.value + E2_Delay_t
114                 if roots[2] == 1:
115                         V4 = 2
116                         V5 = 2
117         elif ret != 0:
118                 print "CVODE␣Error:␣%i"%(ret)
119                 break
120         results[0].append(tout)
121         results[1].append(y[S1])
122         results[2].append(y[S2])
123         results[3].append(y[S3])
```

```
124             results[4].append(R1(y))
125             results[5].append(R2(y))
126             results[6].append(R4(y))
127 
128             iout += 1
129             tout += 0.01
130 
131     pyplot.figure(1)
132     pyplot.subplot(211)
133     pyplot.title('PySUNDIALS/CVODE')
134     pyplot.ylabel('Concentration')
135     pyplot.plot(
136             results[0], results[1], 'k-',
137             results[0], results[2], 'k--',
138             results[0], results[3], 'k-.'
139     )
140     pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
141 
142     pyplot.subplot(212)
143     pyplot.xlabel('t')
144     pyplot.ylabel('Rate')
145     pyplot.plot(
146             results[0], results[4], 'k-',
147             results[0], results[5], 'k--',
148             results[0], results[6], 'k-.'
149     )
150     pyplot.legend(('$v_1$', '$v_2$', '$v_4$'))
151     pyplot.show()
```

Listing A.5: zeilinger.psc

```
# Generated by PySCeS 0.6.9 (2008-12-03 07:52)

# Keywords
Description: Zeilinger2006_PRR7-PRR9light-Y
Modelname: Zeilinger2006_PRR7_PRR9light_Y
Output_In_Conc: True
Species_In_Conc: True

# GlobalUnitDefinitions
UnitVolume: litre, 1.0, 0, 1
UnitLength: metre, 1.0, 0, 1
UnitSubstance: mole, 1.0, -9, 1
UnitArea: metre, 1.0, 0, 2
UnitTime: second, 3600.0, 0, 1

# Compartments
Compartment: cytoplasm, 1.0, 3
Compartment: nucleus, 1.0, 3

# Reactions
R16:
    $pool > cXm
    nucleus*(n3*pow(cTn,d)/(pow(g4,d)+pow(cTn,d)))
# R16 has modifier(s): cTn

R17:
    cXm > $pool
    nucleus*(m9*cXm/(k7+cXm))

R14:
    cTc > $pool
    cytoplasm*((lmax-ld)*m5+m6)*(cTc/(k5+cTc))

R15:
    cTn > $pool
    nucleus*((lmax-ld)*m7+m8)*(cTn/(k6+cTn))

R12:
    cTc > cTn
    cytoplasm*r3*cTc

R13:
    cTn > cTc
    nucleus*r4*cTn

R10:
    cTm > $pool
    nucleus*m4*cTm/(k4+cTm)

R11:
    $pool > cTc
    cytoplasm*p2*cTm
# R11 has modifier(s): cTm

R18:
    $pool > cXc
    cytoplasm*p3*cXm
# R18 has modifier(s): cXm

R19:
    cXc > cXn
```

```
62      cytoplasm*r5*cXc
63
64  R38:
65      cP7n > $pool
66      nucleus*m18*cP7n/(k16+cP7n)
67
68  R39:
69      $pool > cP9m
70      (ld*q4*cPn+n7*ld+n8)*pow(cLn,k)/(pow(g10,k)+pow(cLn,k))
71  # R39 has modifier(s): cPn cLn
72
73  R34:
74      $pool > cP7c
75      cytoplasm*p6*cP7m
76  # R34 has modifier(s): cP7m
77
78  R35:
79      cP7c > cP7n
80      cytoplasm*r9*cP7c
81
82  R36:
83      cP7n > cP7c
84      nucleus*r10*cP7n
85
86  R37:
87      cP7c > $pool
88      cytoplasm*m17*cP7c/(k15+cP7c)
89
90  R30:
91      $pool > cPn
92      nucleus*(lmax-ld)*p5
93
94  R32:
95      $pool > cP7m
96      nucleus*n6*pow(cLn,j)/(pow(g9,j)+pow(cLn,j))
97  # R32 has modifier(s): cLn
98
99  R33:
100     cP7m > $pool
101     nucleus*m16*cP7m/(k14+cP7m)
102
103 R4:
104     $pool > cLc
105     cytoplasm*p1*cLm
106 # R4 has modifier(s): cLm
107
108 R5:
109     cLc > cLn
110     cytoplasm*r1*cLc
111
112 R6:
113     cLn > cLc
114     nucleus*r2*cLn
115
116 R7:
117     cLc > $pool
118     cytoplasm*m2*cLc/(k2+cLc)
119
120 R1:
121     $pool > cLm
122     nucleus*ld*q1*cPn
123 # R1 has modifier(s): cPn
```

```
124
125 R2:
126     $pool > cLm
127     nucleus*(n1*pow(cXn,a)/(pow(g1,a)+pow(cXn,a)))*
128     (pow(g7,h)/(pow(g7,h)+pow(cP7n,h)))*
129     (pow(g8,i)/(pow(g8,i)+pow(cP9n,i)))
130 # R2 has modifier(s): cXn cP7n cP9n
131
132 R3:
133     cLm > $pool
134     nucleus*m1*cLm/(k1+cLm)
135
136 R42:
137     cP9c > cP9n
138     cytoplasm*r11*cP9c
139
140 R8:
141     cLn > $pool
142     nucleus*m3*cLn/(k3+cLn)
143
144 R9:
145     $pool > cTm
146     nucleus*(n2*pow(cYn,b)/(pow(g2,b)+pow(cYn,b)))
147     *(pow(g3,c)/(pow(g3,c)+pow(cLn,c)))
148 # R9 has modifier(s): cYn cLn
149
150 R43:
151     cP9n > cP9c
152     nucleus*r12*cP9n
153
154 R31b:
155     cPn > $pool
156     nucleus*q3*ld*cPn
157
158 R31a:
159     cPn > $pool
160     nucleus*m15*cPn/(k13+cPn)
161
162 R45:
163     cP9n > $pool
164     nucleus*m21*cP9n/(k19+cP9n)
165
166 R44:
167     cP9c > $pool
168     cytoplasm*m20*cP9c/(k18+cP9c)
169
170 R41:
171     $pool > cP9c
172     cytoplasm*p7*cP9m
173 # R41 has modifier(s): cP9m
174
175 R40:
176     cP9m > $pool
177     nucleus*m19*cP9m/(k17+cP9m)
178
179 R29:
180     cYn > $pool
181     nucleus*m14*cYn/(k12+cYn)
182
183 R28:
184     cYc > $pool
185     cytoplasm*m13*cYc/(k11+cYc)
```

```
186
187 R27:
188     cYn > cYc
189     nucleus*r8*cYn
190
191 R26:
192     cYc > cYn
193     cytoplasm*r7*cYc
194
195 R25:
196     $pool > cYc
197     cytoplasm*p4*cYm
198 # R25 has modifier(s): cYm
199
200 R24:
201     cYm > $pool
202     nucleus*m12*cYm/(k10+cYm)
203
204 R23:
205     $pool > cYm
206     nucleus*(ld*q2*cPn+(ld*n4+n5)*pow(g5,e)/
207     (pow(g5,e)+pow(cTn,e)))*(pow(g6,f)/(pow(g6,f)+pow(cLn,f)))
208 # R23 has modifier(s): cTn cLn cPn
209
210 R22:
211     cXn > $pool
212     nucleus*m11*cXn/(k9+cXn)
213
214 R21:
215     cXc > $pool
216     cytoplasm*m10*cXc/(k8+cXc)
217
218 R20:
219     cXn > cXc
220     nucleus*r6*cXn
221
222 # Event definitions
223 Event: event_0000001, operator.le(Day_in_hours-_TIME_,0), 0.0
224 {
225 ld = 1
226 Day_in_hours = Day_in_hours + 24
227 }
228 Event: event_0000002, (operator.le(Day_in_hours-_TIME_,12) \
229 and operator.gt(Day_in_hours-_TIME_,0)), 0.0
230 {
231 ld = 0
232 }
233
234 # Fixed species
235
236 # Variable species
237 cXm@nucleus = 0.652
238 cLm@nucleus = 0.1114
239 cLn@nucleus = 0.2366
240 cPn@nucleus = 0.0
241 cYm@nucleus = 0.2992
242 cYn@nucleus = 17.4355
243 cXc@cytoplasm = 2.4188
244 cLc@cytoplasm = 0.0731
245 cYc@cytoplasm = 49.2611
246 cP9c@cytoplasm = 0.734
247 cP7c@cytoplasm = 0.0266
```

```
cXn@nucleus = 14.7289
cTc@cytoplasm = 5.2235
cTm@nucleus = 3.6732
cTn@nucleus = 4.5333
cP9n@nucleus = 1.1162
cP7m@nucleus = 0.0204
cP7n@nucleus = 1.5103
cP9m@nucleus = 0.002

# Parameters
n3 = 0.6703
d = 1.0164
g4 = 11.3625
m9 = 2.6345
k7 = 8.6873
lmax = 1.0
ld = 1.0
m5 = 9.3024
m6 = 10.899
k5 = 16.9133
m7 = 0.7527
m8 = 13.7459
k6 = 43.7049
r3 = 29.4222
r4 = 33.6178
m4 = 8.5185
k4 = 4.0551
p2 = 1.0494
p3 = 8.583
r5 = 27.818
m18 = 8.671
k16 = 42.4837
q4 = 7.4548
n7 = 0.0833
n8 = 2.0738
k = 3.3953
g10 = 5.6855
p6 = 6.7738
r9 = 31.0318
r10 = 0.4557
m17 = 5.4062
k15 = 49.4094
p5 = 0.5
n6 = 11.3117
j = 2.5579
g9 = 14.5219
m16 = 9.531
k14 = 50.9418
p1 = 1.2294
r1 = 31.5166
r2 = 9.1138
m2 = 10.4609
k2 = 32.7881
q1 = 7.9798
n1 = 2.3023
a = 2.2802
g1 = 16.3389
g7 = 0.4444
h = 2.2116
g8 = 11.0459
i = 1.1065
m1 = 8.0568
```

```
310 k1 = 22.3951
311 r11 = 34.6266
312 m3 = 12.7853
313 k3 = 29.0823
314 n2 = 7.5433
315 b = 3.1075
316 g2 = 16.7487
317 g3 = 11.5922
318 c = 1.6808
319 r12 = 22.838
320 q3 = 1.0
321 m15 = 1.2
322 k13 = 1.2
323 m21 = 0.028
324 k19 = 26.5795
325 m20 = 3.4152
326 k18 = 16.2407
327 p7 = 10.4532
328 m19 = 6.1155
329 k17 = 18.6089
330 m14 = 3.2581
331 k12 = 23.2876
332 m13 = 6.8544
333 k11 = 48.5862
334 r8 = 25.8963
335 r7 = 9.1917
336 p4 = 14.6828
337 m12 = 8.4753
338 k10 = 16.1162
339 q2 = 2.5505
340 n4 = 1.5293
341 n5 = 2.6296
342 g5 = 0.5061
343 e = 1.4943
344 g6 = 7.8469
345 f = 1.9491
346 m11 = 7.9066
347 k9 = 14.605
348 m10 = 9.2511
349 k8 = 13.4324
350 r6 = 4.2863
351 Day_in_hours = 24.0
```

Listing A.6: zeilinger.py

```
1 mod = pysces.model('zeilinger')
2 mod.doLoad()
3 mod.doSimPlot(550,5500,plot=['cTm','cYm'])
4 pysces.plt.setRange('x',480,550)
```

Listing A.7: moiety_branch-cvodes-independent.py

```
from pysundials import cvodes
from pysundials import nvecserial
import ctypes
from matplotlib import pyplot

#V1 = 1.0
#V2 = 10.0
#V3 = 1.0
#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
X0 = 1.0
X4 = 1.0e-12
X5 = 1.0e-12
X6 = 1.0
X7 = 1.0e-12

S1 = 0
S2 = 1
V1 = 0
V2 = 1
V3 = 2
V4 = 3

class UserData(ctypes.Structure):
        _fields_ = [
                ('p', cvodes.realtype*4)
        ]
PUserData = ctypes.POINTER(UserData)

def R1(y, p):
        return (p[V1]/(K1_X0*K1_S2)*(X0*y[S2]))/\
                (1 + X0/K1_X0 + y[S2]/K1_S2 + (X0*y[S2])/(K1_X0*K1_S2))

def R2(y, p):
        return (p[V2]/(K2_S3*K2_X6))*((icsum-y[S2])*X6)/\
                (1 + (icsum-y[S2])/K2_S3 + X6/K2_X6\
                + (((icsum-y[S2])*X6)/(K2_S3*K2_X6)))

def R3(y, p):
        return (p[V3]*y[S1])/(y[S1] + K3_S1)

def R4(y, p):
        return (p[V4]*y[S1])/(y[S1] + K4_S1)

def f(t, y, ydot, f_data):
        data = ctypes.cast(f_data, PUserData).contents

        ydot[S2] = R2(y, data.p) - R1(y, data.p)
        ydot[S1] = R1(y, data.p) - R3(y, data.p) - R4(y, data.p)

        return 0

data = UserData()
data.p[V1] = 1.0
data.p[V2] = 10.0
data.p[V3] = 1.0
```

```
62  data.p[V4] = 1.0
63
64  y = cvodes.NVector([1.0, 0.7])
65  icsum = 0.7 + 0.3 #setting the initial sum of the conservation
66
67  cvodes_mem = cvodes.CVodeCreate(cvodes.CV_BDF, cvodes.CV_NEWTON)
68  cvodes.CVodeMalloc(
69          cvodes_mem,
70          f,
71          0.0,
72          y,
73          cvodes.CV_SS,
74          1.0e-8,
75          1.0e-12
76  )
77  cvodes.CVodeSetFdata(cvodes_mem, ctypes.pointer(data))
78  cvodes.CVDense(cvodes_mem, 2)
79
80  yS = nvecserial.NVectorArray([([0]*2)]*4)
81
82  cvodes.CVodeSensMalloc(cvodes_mem, 4, cvodes.CV_SIMULTANEOUS, yS)
83
84  #So this bit below might seem like magic, but it's fairly straight
85  #forward.  This particular line doesn't look like the example
86  #cvsfwddenx.py because we are not suplying our own sensitivity
87  #function.
88
89  #CVodeSetSensParams expects four parameters
90  #(for more detail see p. 111 of the CVODES user guide)
91  # 1. the cvodes memory object
92  # 2. a pointer to the array of parameter values which MUST be passed
93  #    through the user data structure (so CVODES knows where the values
94  #    are and can peturb them, presumably)
95  # 3. an array (i.e. list) of scaling factors, one for each parameter
96  #    for which sensitivies are to be determined
97  # 4. an array of integers (either 1 or 0), where a 1 indicates the
98  #    respective paramter value should be used in estimating
99  #    sensistivities
100
101 cvodes.CVodeSetSensParams(cvodes_mem,
102         data.p, #we have four system parameters (The four VMax's)
103         [1]*4, #they should all be scaled by 1, i.e. unscaled,
104         [1]*4 #they all contribute to the estimation of sensitivities
105 )
106
107 results = ([], [], [], [], [], [])
108
109 t = cvodes.realtype(0.0)
110 results[0].append(t.value)
111 results[1].append(y[S1])
112 results[2].append(y[S2])
113 results[3].append((icsum-y[S2]))
114 results[4].append(yS[V2][S1])
115 results[5].append(yS[V2][S2])
116
117 iout = 1
118 tout = 0.05
119 while iout <= 80:
120         ret = cvodes.CVode(cvodes_mem,
121                 tout,
122                 y,
123                 ctypes.byref(t),
```

```
124                     cvodes.CV_NORMAL
125             )
126             cvodes.CVodeGetSens(cvodes_mem, t, yS)
127             if ret != 0:
128                     print "CVODE␣Error:␣%i"%(ret)
129                     break
130             results[0].append(tout)
131             results[1].append(y[S1])
132             results[2].append(y[S2])
133             results[3].append((icsum-y[S2]))
134             results[4].append(yS[V2][S1])
135             results[5].append(yS[V2][S2])
136
137             iout += 1
138             tout += 0.05
139
140     pyplot.figure(1)
141     pyplot.subplot(211)
142     pyplot.ylabel('Concentration')
143     pyplot.plot(
144             results[0], results[1], 'k-',
145             results[0], results[2], 'k--',
146             results[0], results[3], 'k-.'
147     )
148     pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
149
150     pyplot.subplot(212)
151     pyplot.xlabel('t')
152     pyplot.plot(
153             results[0], results[4], 'k-',
154             results[0], results[5], 'k--'
155     )
156     pyplot.legend(('$\delta␣S_1/dV_2$', '$\delta␣S_2/dV_2$'))
157
158     pyplot.show()
```

Listing A.8: brusselator-cvodes.py

```
from pysundials import cvodes
from pysundials import nvecserial
import ctypes
import math
from matplotlib import pyplot

A = 0.5
B = 3.0
D = 0.0
E = 0.0

class UserData(ctypes.Structure):
        _fields_ = [
                ('p', cvodes.realtype*4)
        ]
PUserData = ctypes.POINTER(UserData)

def R1(y, k):
    return k[0]*A

def R2(y, k):
    return k[1]*y[0]*y[0]*y[1]

def R3(y, k):
    return k[2]*y[0]*B

def R4(y, k):
    return k[3]*y[0]

def f(t, y, ydot, f_data):
        data = ctypes.cast(f_data, PUserData).contents

        ydot[0] = R1(y,data.p)+R2(y,data.p)-R3(y,data.p)-R4(y,data.p)
        ydot[1] = R3(y,data.p)-R2(y,data.p)

        return 0


data = UserData()
data.p[0] = 1.0
data.p[1] = 1.0
data.p[2] = 1.0
data.p[3] = 1.0

y = cvodes.NVector([3.0, 3.0])

cvodes_mem = cvodes.CVodeCreate(cvodes.CV_BDF, cvodes.CV_NEWTON)
cvodes.CVodeMalloc(
        cvodes_mem,
        f,
        0.0,
        y,
        cvodes.CV_SS,
        1.0e-8,
        1.0e-12
)
cvodes.CVodeSetFdata(cvodes_mem, ctypes.pointer(data))
cvodes.CVDense(cvodes_mem, 2)

yStmp = [cvodes.NVector([0,0,0,0]),
        cvodes.NVector([0,0,0,0]),
```

```
62          cvodes.NVector([0,0,0,0]),
63          cvodes.NVector([0,0,0,0])
64  ]
65  yS = nvecserial.NVectorArray([([0]*2)]*4)
66
67  cvodes.CVodeSensMalloc(cvodes_mem, 4, cvodes.CV_SIMULTANEOUS, yS)
68
69  #So this bit below might seem like magic, but it's fairly straight
70  #forward.  This particular line doesn't look like the example
71  #cvsfwddenx.py because we are not suplying our own sensitivity
72  #function.
73
74  #CVodeSetSensParams expects four parameters
75  #(for more detail see p. 111 of the CVODES user guide)
76  # 1. the cvodes memory object
77  # 2. a pointer to the array of parameter values which MUST be passed
78  #    through the user data structure (so CVODES knows where the values
79  #    are and can peturb them, presumably)
80  # 3. an array (i.e. list) of scaling factors, one for each parameter
81  #    for which sensitivies are to be determined
82  # 4. an array of integers (either 1 or 0), where a 1 indicates the
83  #    respective paramter value should be used in estimating
84  #    sensistivities
85
86  cvodes.CVodeSetSensParams(cvodes_mem,
87          data.p, #we have four system parameters (The four k's)
88          [1]*4, #they should all be scaled by 1, i.e. unscaled,
89          [1]*4 #they all contribute to the estimation of sensitivities
90  )
91
92  results = ([], [], [], [], [], [], [])
93
94  t = cvodes.realtype(0.0)
95  results[0].append(t.value)
96  results[1].append(y[0])
97  results[2].append(y[1])
98  results[3].append(yS[0][0])
99  results[4].append(yS[0][1])
100 results[5].append(yS[0][0])
101 results[6].append(yS[0][1])
102
103 iout = 1
104 tout = 0.01
105 while iout <= 9999:
106         ret = cvodes.CVode(cvodes_mem,
107                 tout,
108                 y,
109                 ctypes.byref(t),
110                 cvodes.CV_NORMAL
111         )
112         cvodes.CVodeGetSens(cvodes_mem, t, yS)
113         if ret != 0:
114                 print "CVODE␣Error:␣%i"%(ret)
115                 break
116         results[0].append(t.value)
117         results[1].append(y[0])
118         results[2].append(y[1])
119         results[3].append(yS[0][0])
120         results[4].append(yS[0][1])
121         if yS[0][0] < 0:
122                 results[5].append(-math.log10(abs(yS[0][0])))
123         else:
```

```
124                 results[5].append(math.log10(abs(yS[0][0])))
125         if yS[0][1] < 0:
126                 results[6].append(-math.log10(abs(yS[0][1])))
127         else:
128                 results[6].append(math.log10(abs(yS[0][1])))
129
130         iout += 1
131         tout += 0.01
132
133 pyplot.figure(1)
134 pyplot.subplot(311)
135 pyplot.ylabel('Concentration')
136 pyplot.plot(
137         results[0], results[1], 'k-',
138         results[0], results[2], 'k--',
139 )
140 pyplot.legend(('$X$', '$Y$'))
141
142 pyplot.subplot(312)
143 pyplot.plot(
144         results[0], results[3], 'k-',
145         results[0], results[4], 'k--',
146 )
147 pyplot.legend(('$\delta␣X/dk_1$', '$\delta␣Y/dk_1$'))
148
149 pyplot.subplot(313)
150 pyplot.ylabel('$log_{10}|y|$')
151 pyplot.xlabel('t')
152 pyplot.plot(
153         results[0], results[5], 'k-',
154         results[0], results[6], 'k--',
155 )
156 pyplot.legend(('$\delta␣X/dk_1$', '$\delta␣Y/dk_1$'))
157
158 pyplot.show()
```

Listing A.9: moiety_branch-ida.py

```
from pysundials import ida
import ctypes
from matplotlib import pyplot

V1 = 1.0
V2 = 10.0
V3 = 1.0
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
X0 = 1.0
X4 = 0
X5 = 0
X6 = 1.0
X7 = 0

S1 = 0
S2 = 1
S3 = 2

def R1(y):
        return (V1/(K1_X0*K1_S2)*(X0*y[S2]))/\
                (1+X0/K1_X0+y[S2]/K1_S2+(X0*y[S2])/(K1_X0*K1_S2))

def R2(y):
        return (V2/(K2_S3*K2_X6))*(y[S3]*X6)/\
                (1+y[S3]/K2_S3+X6/K2_X6\
                + ((y[S3]*X6)/(K2_S3*K2_X6)))

def R3(y):
        return (V3*y[S1])/(y[S1]+K3_S1)

def R4(y):
        return (V4*y[S1])/(y[S1]+K4_S1)

def f(t, yy, yp, rr, data):
        rr[0] = R1(yy)-R3(yy)-R4(yy)-yp[0]
        rr[1] = R2(yy)-R1(yy)-yp[1]
        rr[2] = yy[1]+yy[2]-1

        return 0

yy = ida.NVector([1.0, 0.7, 0.3])
yp = ida.NVector([0.1, 0.9, 0.05])

mem = ida.IDACreate()
ida.IDAMalloc(mem, f, 0.0, yy, yp, ida.IDA_SS, 1.0e-8, 1.0e-12)
ida.IDADense(mem, 3)

results = ([], [], [], [])

t = ida.realtype(0.0)
results[0].append(t.value)
results[1].append(yy[S1])
results[2].append(yy[S2])
results[3].append(yy[S3])

```

```
62  iout = 1
63  tout = 0.1
64  while iout < 50:
65          ret = ida.IDASolve(
66                  mem,
67                  tout,
68                  ctypes.byref(t),
69                  yy,
70                  yp,
71                  ida.IDA_NORMAL
72          )
73          if ret != 0:
74                  print "IDA␣Error:␣%i"%(ret)
75                  break
76          results[0].append(t.value)
77          results[1].append(yy[S1])
78          results[2].append(yy[S2])
79          results[3].append(yy[S3])
80          iout += 1
81          tout += 0.1
82
83  pyplot.title('PySUNDIALS/IDA')
84  pyplot.xlabel('t')
85  pyplot.ylabel('Concentration')
86  pyplot.plot(
87          results[0], results[1], 'k-',
88          results[0], results[2], 'k--',
89          results[0], results[3], 'k-.'
90  )
91  pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
92  pyplot.show()
```

Listing A.10: linear_equilibrium-cvode-independent.py

```
from pysundials import cvode
import ctypes
from matplotlib import pyplot
import sys

V1 = 1.0
K1_X0 = 1.0
K1_S1 = 1.0
K1eq = 1.0

K2 = 0.9
K3 = 0.75

V4 = 1.0
K4_S3 = 1.0
K4_X4 = 1.0
K4eq = 1.0

X0 = 1.0
X4 = 1.0e-12

S1 = 0
S2 = 1
S3 = 2

def R1(y):
        return (V1/K1_X0)*(X0-(y[S1]/K1eq))/\
                (1+(X0/K1_X0)+(y[S1]/K1_S1))

def R4(y):
        return (V4/K4_S3)*(y[S1]*K2*K3-(X4/K4eq))/\
                (1+(y[S1]*K2*K3/K4_S3)+(X4/K4_X4))

def f(t, y, ydot, f_data):
        ydot[0] = R1(y)-R4(y)
        return 0

y = cvode.NVector([1])

cvode_mem = cvode.CVodeCreate(cvode.CV_BDF, cvode.CV_NEWTON)
cvode.CVodeMalloc(
        cvode_mem,
        f,
        0.0,
        y,
        cvode.CV_SS,
        1.0e-8,
        1.0e-12
)
cvode.CVDense(cvode_mem, 1)

results = ([], [], [], [])

t = cvode.realtype(0.0)
results[0].append(t.value)
results[1].append(y[S1])
results[2].append(y[S1]*K2)
results[3].append(y[S1]*K2*K3)

iout = 1
tout = 0.1
```

```
62  while iout < 100:
63          ret = cvode.CVode(
64                  cvode_mem,
65                  tout,
66                  y,
67                  ctypes.byref(t),
68                  cvode.CV_NORMAL
69          )
70          if ret != 0:
71                  print "CVODE␣Error:␣%i"%(ret)
72                  break
73          results[0].append(t.value)
74          results[1].append(y[S1])
75          results[2].append(y[S1]*K2)
76          results[3].append(y[S1]*K2*K3)
77
78          iout += 1
79          tout += 0.1
80
81  pyplot.title('PySUNDIALS/CVODE')
82  pyplot.xlabel('t')
83  pyplot.ylabel('Concentration')
84  pyplot.plot(
85          results[0], results[1], 'k-',
86          results[0], results[2], 'k--',
87          results[0], results[3], 'k-.'
88  )
89  pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
90  pyplot.show()
```

Listing A.11: linear_equilibrium-ida.py

```
from pysundials import ida
import ctypes
from matplotlib import pyplot
import sys

V1 = 1.0
K1_X0 = 1.0
K1_S1 = 1.0
K1eq = 1.0

K2 = 0.9
K3 = 0.75

V4 = 1.0
K4_S3 = 1.0
K4_X4 = 1.0
K4eq = 1.0

X0 = 1.0
X4 = 1.0e-12

S1 = 0
S2 = 1
S3 = 2

def R1(y):
        return (V1/K1_X0)*(X0-(y[S1]/K1eq))/(1+(X0/K1_X0)+(y[S1]/K1_S1))

def R4(y):
        return (V4/K4_S3)*(y[S3]-(X4/K4eq))/(1+(y[S3]/K4_S3)+(X4/K4_X4))

def f(t, yy, yp, rr, data):
        rr[0] = R1(yy)-R4(yy)-yp[0]
        rr[1] = yy[0]*K2-yy[1]
        rr[2] = yy[1]*K3-yy[2]
        return 0

yy = ida.NVector([1, 0, 0])
yp = ida.NVector([0, 0, 0])
f(0, yy, yp, yp, None)

mem = ida.IDACreate()
ida.IDAMalloc(mem, f, 0.0, yy, yp, ida.IDA_SS, 1.0e-8, 1.0e-12)
ida.IDADense(mem, 3)

ida.IDASetId(mem, ida.NVector([1,0,0]))
ida.IDACalcIC(mem, ida.IDA_YA_YDP_INIT, 0.1)

results = ([], [], [], [])

t = ida.realtype(0.0)
results[0].append(t.value)
results[1].append(yy[S1])
results[2].append(yy[S1]*K2)
results[3].append(yy[S1]*K2*K3)

iout = 1
tout = 0.1
while iout < 100:
        ret = ida.IDASolve(mem,
                tout,
```

```
62                  ctypes.byref(t),
63                  yy,
64                  yp,
65                  ida.IDA_NORMAL
66          )
67          if ret != 0:
68                  print "IDA␣Error:␣%i"%(ret)
69                  break
70          results[0].append(t.value)
71          results[1].append(yy[S1])
72          results[2].append(yy[S2])
73          results[3].append(yy[S3])
74          iout += 1
75          tout += 0.1
76
77  pyplot.title('PySUNDIALS/IDA')
78  pyplot.xlabel('t')
79  pyplot.ylabel('Concentration')
80  pyplot.plot(
81          results[0], results[1], 'k-',
82          results[0], results[2], 'k--',
83          results[0], results[3], 'k-.'
84  )
85  pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
86  pyplot.show()
```

Listing A.12: moiety_branch-kinsol.py

```
from pysundials import kinsol
import ctypes
from matplotlib import pyplot

V1 = 1.0
V2 = 10.0
V3 = 1.0
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
X0 = 1.0
X4 = 1.0e-12
X5 = 1.0e-12
X6 = 1.0
X7 = 1.0e-12

S1 = 0
S2 = 1
#S3 = 2

def R1(y):
        return (V1/(K1_X0*K1_S2)*(X0*y[S2]))/\
                (1 + X0/K1_X0 + y[S2]/K1_S2 + (X0*y[S2])/(K1_X0*K1_S2))

def R2(y):
        return (V2/(K2_S3*K2_X6))*((icsum-y[S2])*X6)/\
                (1 + (icsum-y[S2])/K2_S3 + X6/K2_X6\
                + (((icsum-y[S2])*X6)/(K2_S3*K2_X6)))

def R3(y):
        return (V3*y[S1])/(y[S1] + K3_S1)

def R4(y):
        return (V4*y[S1])/(y[S1] + K4_S1)

def f(u, fval, f_data):
        fval[S2] = R2(u) - R1(u)
        fval[S1] = R1(u) - R3(u) - R4(u)

        return 0

u = kinsol.NVector([1.0, 0.7])
template = kinsol.NVector([0, 0])
s = kinsol.NVector([1, 1])
icsum = 0.7 + 0.3 #setting the initial sum of the conservation

kin_mem = kinsol.KINCreate()
kinsol.KINMalloc(kin_mem, f, template)
kinsol.KINDense(kin_mem, 2)
kinsol.KINSol(kin_mem, u, kinsol.KIN_LINESEARCH, s, s)

print "Steady-state␣species␣concentrations"
print "S2_ss␣=␣%.4e"%(u[1])
print "S1_ss␣=␣%.4e"%(u[0])
print "S3_ss␣=␣%.4e"%(icsum-u[1])
print
print "Steady-state␣fluxes"
```

```
62  print "J_R1␣=␣%.4e"%(R1(u))
63  print "J_R2␣=␣%.4e"%(R2(u))
64  print "J_R3␣=␣%.4e"%(R3(u))
65  print "J_R4␣=␣%.4e"%(R4(u))
```

Listing A.13: moiety_branch-hybrd.py

```
1 import pysces
2 m = pysces.model('moiety_branch')
3 m.doLoad()
4 m.doStateShow()
```

Listing A.14: moiety_branch-kinsol-paramscan.py

```
from pysundials import kinsol
import ctypes
from matplotlib import pyplot

V1 = 1.0
V2 = 0.1
V3 = 1.0
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
X0 = 1.0
X4 = 1.0e-12
X5 = 1.0e-12
X6 = 1.0
X7 = 1.0e-12

S1 = 0
S2 = 1
#S3 = 2

def R1(y):
        return (V1/(K1_X0*K1_S2)*(X0*y[S2]))/\
                (1 + X0/K1_X0 + y[S2]/K1_S2 + (X0*y[S2])/(K1_X0*K1_S2))

def R2(y):
        return (V2/(K2_S3*K2_X6))*((icsum-y[S2])*X6)/\
                (1 + (icsum-y[S2])/K2_S3 + X6/K2_X6\
                + (((icsum-y[S2])*X6)/(K2_S3*K2_X6)))

def R3(y):
        return (V3*y[S1])/(y[S1] + K3_S1)

def R4(y):
        return (V4*y[S1])/(y[S1] + K4_S1)

def f(u, fval, f_data):
        fval[S2] = R2(u) - R1(u)
        fval[S1] = R1(u) - R3(u) - R4(u)

        return 0

u = kinsol.NVector([1.0, 0.7])
template = kinsol.NVector([0, 0])
s = kinsol.NVector([1, 1])
icsum = 0.7 + 0.3 #setting the initial sum of the conservation

kin_mem = kinsol.KINCreate()
kinsol.KINMalloc(kin_mem, f, template)
kinsol.KINDense(kin_mem, 2)

results = ([], [], [], [], [], [], [], [])

while V2 <= 5:
        kinsol.KINSol(kin_mem, u, kinsol.KIN_LINESEARCH, s, s)
        results[0].append(V2)
        results[1].append(u[0])
        results[2].append(u[1])
```

```
62          results[3].append(icsum-u[1])
63          results[4].append(R1(u))
64          results[5].append(R2(u))
65          results[6].append(R3(u))
66          results[7].append(R4(u))
67          V2 += 0.1
68
69  pyplot.figure(1)
70  pyplot.subplot(211)
71  pyplot.title('PySUNDIALS/KINSOL')
72  pyplot.ylabel('Steady␣State␣Concentration')
73  pyplot.plot(
74          results[0], results[1], 'k-',
75          results[0], results[2], 'k--',
76          results[0], results[3], 'k-.'
77  )
78  pyplot.legend(('$S_1$', '$S_2$', '$S_3$'))
79
80  pyplot.subplot(212)
81  pyplot.xlabel('$V_2$')
82  pyplot.ylabel('Steady␣State␣Flux')
83  pyplot.plot(
84          results[0], results[4], 'k-',
85          results[0], results[5], 'k--',
86          results[0], results[6], 'k-.',
87          results[0], results[7], 'k.'
88  )
89  pyplot.legend(('$J_1$', '$J_2$', '$J_3$', '$J_4$'))
90
91  pyplot.show()
```

Listing A.15: MWC_wholecell_2b.psc

```
Modelname: MWC_wholecell2b
Description: Surovtsev whole cell model

Species_In_Conc: True
Output_In_Conc: True

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

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

FIX: N

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

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

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

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

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

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

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

# tracking functions
!F S_V_Ratio = Mem_Area/Vcyt
!F Mconc = (M)/M_init
!F Lconc = (L)/L_init
!F Pconc = (P)/P_init
!F Mem_Area_scaled = Mem_Area/Mem_Area_init
!F Vcyt_scaled = Vcyt/Vcyt_init
!F sigma_test = sigma_P*Pmem + sigma_L*Lmem
!F Co_test = M + L + P

"""
As it turns out if one wants to use <thing>_init in a rule and have this
translated via Core2 to SBML or PSC, you *have* to explicitly initialise
it as a paramter. This also goes for the parameter you are setting with
the rule.

NOTE: PySCeS will work as it generated the init attributes autoamtically
but then they should be regarded as 'internal' attributes that are not
```

```
62  exported.
63  """
64
65  M_init = 199693.0
66  L_init = 102004
67  P_init = 5303
68  Vcyt_init = 1.0
69  Mem_Area_init = 5.15898
70  Mconc = 1.0
71  Lconc = 1.0
72  Pconc = 1.0
73  Mem_Area_scaled = 1.0
74  Vcyt_scaled = 1.0
75  S_V_Ratio = 5.15898
76  sigma_test = 1.0
77  Co_test = 3.07e5
78
79  N@Vout = 3.07e5
80
81  Pmem@Mem_Area = 37.38415
82  Lmem@Mem_Area = 8291.2350678770199
83
84  M@Vcyt = 199693.0
85  L@Vcyt = 102004
86  P@Vcyt = 5303
87
88  k1 = 0.00089709
89  k2 = 0.000182027
90  k3 = 1.7539e-010
91  k4 = 5.0072346e-005
92  k5 = 0.000574507164
```

Listing A.16: MWC_LSODA.py

```
1  import pysces
2  m = pysces.model('MWC_wholecell_2b')
3  m.doLoad()
4  m.mode_integrator='LSODA'
5  m.doSimPlot(end=100, points=1001)
```

Listing A.17: beta5R.py

```
from pysundials import cvode
import ctypes
from matplotlib import pyplot

#Constants
k1 = 0.89709e-3
k2 = 0.182027e-3
k3 = 0.17539e-9
k4 = 0.50072346e-4
k5 = 0.574507164e-3
N0 = 307000
Ccyt0 = 307000
sigmaL = 120e-6
m1 = 244
m2 = 42
sigmaP = (sigmaL*m1)/m2

#indices into vector
P = 0
L = 1
M = 2
Pmem = 3
Lmem = 4
V = 5
A = 6

def R1(y):
        return y[A]*k1*N0*y[Pmem]/y[A]

def R2(y):
        return y[V]*k2*y[M]/y[V]

def R3(y):
        return y[V]*k3*(y[M]/y[V])*(y[P]/y[V])**2

def R4(y):
        return y[A]*k4*y[P]/y[V]

def R5(y):
        return y[A]*k5*y[L]/y[V]

def f(t, y, ydot, f_data):
        ydot[P] = R2(y)-R4(y)
        ydot[L] = R3(y)-R5(y)
        ydot[M] = R1(y)-m1*R2(y)-m2*R3(y)
        ydot[Pmem] = R4(y)
        ydot[Lmem] = R5(y)
        ydot[V] = (R1(y)-(m1-1)*R2(y)-(m2-1)*R3(y)-R4(y)-R5(y))/Ccyt0
        ydot[A] = sigmaL*R5(y)+sigmaP*R4(y)

        return 0

t = cvode.realtype(0)

#initial conditions
Pinit = 5303
Linit = 102004
Minit = 199693
Pmeminit = 192.86408217
Lmeminit = 41871.0515
Ainit = 5.15898
```

```
62  Vinit = 1
63
64  y = cvode.NVector([
65          Pinit,
66          Linit,
67          Minit,
68          Pmeminit,
69          Lmeminit,
70          Vinit,
71          Ainit
72  ])
73
74  cvode_mem = cvode.CVodeCreate(cvode.CV_BDF, cvode.CV_NEWTON)
75  cvode.CVodeMalloc(cvode_mem, f, 0.0, y, cvode.CV_SS, 1.0e-8, 1.0e-12)
76  cvode.CVDense(cvode_mem, len(y))
77
78  results = ([], [], [], [], [], [], [], [], [])
79
80  t = cvode.realtype(0.0)
81  results[0].append(t.value)
82  results[1].append(y[P]/(Pinit*y[V]))
83  results[2].append(y[L]/(Linit*y[V]))
84  results[3].append(y[M]/(Minit*y[V]))
85  results[4].append(y[Pmem]*Ainit/(Pmeminit*y[A]))
86  results[5].append(y[Lmem]*Ainit/(Lmeminit*y[A]))
87  results[6].append(y[A]/y[V])
88  results[7].append(y[A]/Ainit)
89  results[8].append(y[V]/Vinit)
90
91  tstop = 120.0
92  npoints = 1200
93  iout = 1
94  tout = tstop/npoints
95  while iout < npoints:
96          ret = cvode.CVode(
97                  cvode_mem,
98                  tout,
99                  y,
100                 ctypes.byref(t),
101                 cvode.CV_NORMAL
102         )
103         if ret != 0:
104                 print "CVODE␣Error:␣%i"%(ret)
105                 break
106
107         results[0].append(t.value)
108         results[1].append(y[P]/(Pinit*y[V]))
109         results[2].append(y[L]/(Linit*y[V]))
110         results[3].append(y[M]/(Minit*y[V]))
111         results[4].append(y[Pmem]*Ainit/(Pmeminit*y[A]))
112         results[5].append(y[Lmem]*Ainit/(Lmeminit*y[A]))
113         results[6].append(y[A]/y[V])
114         results[7].append(y[A]/Ainit)
115         results[8].append(y[V]/Vinit)
116
117         iout += 1
118         tout = iout * tstop / npoints
119
120 pyplot.figure(1)
121 pyplot.subplot(211)
122 pyplot.title('PySUNDIALS/CVODE')
123 pyplot.ylabel('$V_{cyt}$,␣$A_{mem}$')
```

```
124 pyplot.plot(
125         results[0], results[7], 'k--',
126         results[0], results[8], 'k-',
127 )
128 pyplot.legend(('$V_{cyt}$', '$A_{mem}$'))
129
130 pyplot.subplot(212)
131 pyplot.ylabel('$\chi$')
132 pyplot.xlabel('t')
133 pyplot.plot(
134         results[0], results[6], 'k-',
135 )
136
137 pyplot.figure(2)
138 pyplot.subplot(311)
139 pyplot.plot(
140         results[0], results[1], 'k-',
141         results[0], results[2], 'k--',
142         results[0], results[3], 'k-.'
143 )
144 pyplot.legend(('P', 'L', 'M'))
145
146 pyplot.subplot(312)
147 pyplot.plot(
148         results[0], results[4], 'k-'
149 )
150 pyplot.legend(('P$_{mem}$',))
151
152 pyplot.subplot(313)
153 pyplot.xlabel('t')
154 pyplot.plot(
155         results[0], results[5], 'k-'
156 )
157 pyplot.legend(('L$_{mem}$',))
158 pyplot.show()
```

Appendix B

Abridged PySUNDIALS Documentation

B.1 Introduction

PySUNDIALS is a python package providing python bindings for the SUNDIALS suite of solvers. It is being developed by the triple 'J' group at Stellenbosch University, South Africa. While python bindings for SUNDIALS will hopefully be generally useful in the computational scientific community, they are being developed with the specific aim of providing a robust underlying numerical solver capable of implementing models conforming to the Systems Biology Markup Language specification (version 2), including triggers, events, and delays. As such the development process is partially driven by the continuing parallel development of PySCeS.

This documentation serves to introduce and act as a reference for PySUNDIALS. As such it focuses on the differences in behaviour between PySUNDIALS and SUNDIALS. If you wish to know more about the underlying mathematical considerations, or wish a more in depth discussion of how to go about using SUNDIALS in general, the SUNDIALS documentation is the place to start. We assume a basic knowledge of initial value problems and their computational solutions on the part of the reader.

B.2 Installation

B.2.1 Prerequisites

PySUNDIALS requires the following:

- a working copy of SUNDIALS installed with shared libraries compiled
- Python v2.4 with the `ctypes` module, OR Python v2.5 or higher

The python packages `numpy` and `scipy` are recommended but not necessary.

B.2.2 From Source

The latest release version of PySUNDIALS can be download at `http://sourceforge.net/projects/pysundials`, or alternatively, the very latest (potentially non-functional) version via anonymous svn using the command:

```
$ svn co https://pysundials.svn.sourceforge.net/svnroot\
/pysundials pysundials
```

B.2.2.1 On Linux/BSD or other POSIX

- Download and untar the complete SUNDIALS suite.
- `$ ./configure --enable-shared`
- `$ make && make install`
- Download and untar PySUNDIALS
- Change to the directory where you unpacked PySUNDIALS
- `$ python setup.py install`

B.2.3 On Windows using MSys/MinGW

- Download and untar the complete SUNDIALS suite somewhere inside MSys.
- Do *not* run aclocal, autoconf, or autoheader, or the makefiles will break!
- `$ ./configure --enable-shared`
- `$ make && make install`
- Download and untar PySUNDIALS
- Change to the directory where you unpacked PySUNDIALS
- `$ python setup.py -c mingw32 install`

If you receive a compile time error when executing the final command, which complains about missing `sundials_conf.h`, or other missing `.h` files, set the `CPATH` environment variable to point to the directory containing the sundials include directories, for example:

```
$ CPATH=/local/include python setup.py -c mingw32 install
```

Note that using MSys presents unique problems being a hybrid environment. We recommend using MSys only to compile and install PySUNDIALS/SUNDIALS, not for use as an environment in which to run PySUNDIALS. Having followed the above instructions, the SUNDIALS shared libs will be in `%MSYSROOT%/usr/local/lib/`, and end with `-<digit>.exe`. On older versions of MSys/MinGW, the `.exe` extension may be left off.

B.3 Configuration

PySUNDIALS uses `ctypes` to link the required SUNDIALS libraries directly into the running python process. In order to do this, it needs to know where to find those shared libraries. On Linux/BSD systems, this is usually auto detected, as library locations are standard, however on windows systems, the final location and even naming convention of the shared library files is compiler dependent. If PySUNDIALS cannot find the SUNDIALS libraries, please locate them yourself, and specify their locations in a file named `~/.pysundials/config` (Linux/BSD), or `%HOMEPATH%\pysundials\config` (Windows). If PySUNDIALS cannot find this configuration file in your home directory, it will seek it in the same directory it was installed in, i.e. `$PYTHONROOT/site-packages/pysundials/config`.

In the "config" file, anything following a hash (including the hash itself) is considered a comment. Each line specifies the location of a required library in the form:

```
library = path
```

Where *library* is one of `c`, `aux`, `nvecserial`, `nvecparallel`, `cvode`, `cvodes`, `ida`, or `kinsol`, and *path* is the complete path of the appropriate library file. `c`, `aux`, and `nvecparallel` are optional; the first two being generally autodetected, and the second only required if you will be using PySUNDIALS in parallel with MPI.

You can find a sample config file for both posix and mingw32 systems in the doc subdirectory of the PySUNDIALS distribution.

B.4 Using PySUNDIALS

B.4.1 Importing PySUNDIALS modules

There are five PySUNDIALS modules available for general use; One for each of the SUNDIALS modules, namely `cvode`, `cvodes`, `ida`, and `kinsol`. The fifth is the `nvecserial` module, which you may wish to use in conjunction with CVODES for it's convenience functions for dealing with sensitivity analysis data structures. To import one of these modules use:

```
from pysundials import module_name
```

where *module_name* is the name of the module you wish to use.

B.4.2 NVectors

The fundamental data type of SUNDIALS, and hence PySUNDIALS is the `NVector`. PySUNDIALS implements the `NVector` class in a manner that closely resembles a python list. Creating an `NVector` is a simple case of class instantiation:

```
>>> from pysundials import nvecserial
>>> v = nvecserial.NVector([1, 2, 3])
>>> v
[1.0, 2.0, 3.0]
```

When instantiating an `NVector`, simply pass a sequence (tuple, list, numpy array, another `NVector`, etc...) to the constructor. The contents of the sequence determine the length of the `NVector` (which remains immutable for its lifetime) and the initial value of the `NVector`. `NVector` objects can be subscripted or sliced like normal python lists.:

```
>>> v[1]
2.0
>>> v[2] = 4
>>> v
[1.0, 2.0, 4.0]
>>> v[0:2]
[1.0, 2.0]
>>> v[0:2] = (-1, -2)
>>> v
[-1.0, -2.0, 4.0]
```

Operators act intuitively on `NVector`s too,

- `+` and `-` perform scalar or vector addition or subtraction respectively, depending on operands.:

  ```
  >>> w = nvecserial.NVector([1,2,-4])
  >>> v+1
  [0.0, -1.0, 5.0]
  >>> v+w
  [0.0, 0.0, 0.0]
  >>> v-w
  [-2.0, -4.0, 8.0]
  ```

- `*` performs scalar or element-wise multiplication depending on operands.:

```
>>> v*2
[-2.0, -4.0, 8.0]
>>> v*w
[-1.0, -4.0, -16.0]
```

- `/` performs scalar or element-wise division depending on operands.:

```
>>> v/2
[-0.5, -1.0, 2.0]
>>> v/w
[-1.0, -1.0, -1.0]
>>> v/v
[1.0, 1.0, 1.0]
```

- and a variety of object methods perform more complex operations including dot product and various norms. See the reference section for a complete list

An `NVector` object can be used as a numpy array by using its `asarray` method. Note how changes to the array affect the `NVector` and *vice versa*.:

```
>>> import numpy
>>> a = v.asarray()
>>> a
array([-1., -2.,  4.])]
>>> a[0] = 0
>>> a
array([ 0., -2.,  4.])]
>>> v
[0.0, -2.0, 4.0]
>>> v[1] = 0
>>> a
array([ 0.,  0.,  4.])]
```

The `NVector` class is exported to each of the main PySUNDIALS modules, so there is rarely a need to import `nvecserial`.:

```
>>> from pysundials import cvode
>>> v = cvode.NVector([1,2,3])
>>> v
[1.0, 2.0, 3.0]
```

B.4.3 CVODE

Programs using CVODE will generally conform to a certain skeleton layout. The example used here serves to illustrate this skeleton layout, and is neither complete, nor representative of SUNDIALS/PySUNDIALS complete set of capabilities. Please see the function reference or SUNDIALS documentation for more information.

1. Import the cvode and ctypes modules.:

   ```
   from pysundials import cvode
   import ctypes
   ```

2. Define your right-hand side function. This function must take exactly four parameters. The first parameter will be the current value of the independent variable (usually time). The second parameter will be an `NVector` containing the current values of the dependent variables. The third parameter is an `NVector` whose elements must be filled with the new values of the dependent variables. The fourth parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system. For example, a simple problem consisting of three variables and having the following ODEs:

   - $v_1 = r_2 - r_1$
   - $v_2 = r_1 - r_2$
   - $v_3 = r_1 - r_3 - r_4$

   (where r_i is a function of the independent variable and the current values of the dependent variables) would have the following RHS function:

   ```
   def f(t, y, ydot, f_data):
       ydot[0] = r2(t,y) - r1(t,y)
       ydot[1] = r1(t,y) - r2(t,y)
       ydot[2] = r1(t,y) - r3(t,y) - r4(t,y)
       return 0
   ```

3. Define any optional functions such as a Jacobian approximation, error weight and/or root finding functions. See function reference for details on parameters and returns.:

   ```
   def rootfind(t, y, gout, g_data):
       gout[0] = y[0] - 0.5
       gout[1] = y[1] - 0.5
       return 0
   ```

4. Initialise an `NVector` with the initial conditions.:

   ```
   y = cvode.NVector([0.7, 0.3, 0.0])
   ```

5. Create a CVODE object.:

   ```
   cvode_mem = cvode.CVodeCreate(lmm, iter)
   ```

   (where *lmm* is on of `cvode.CV_ADAMS` or `cvode.CV_BDF`, and *iter* is one of `cvode.CV_NEWTON` or `cvode.CV_FUNCTIONAL`)

6. Allocate integrator memory, set the initial value of the independent variable, and set tolerances. Absolute tolerances may be an `NVector` of the same size as `y` in which case `cvode.CV_SV` should be used, or a scalar value applying to all (`cvode.CV_SS`).:

   ```
   abstol = cvode.NVector([1.0e-8, 1.0e-14, 1.0e-6])
   reltol = cvode.realtype(1.0e-4)
   cvode.CVodeMalloc(cvode_mem, f, 0.0, y, cvode.CV_SV,
   reltol, abstol)
   ```

7. Set any optional inputs using `CVSet*()`.

8. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `CVDense`, `CVBand`, `CVDiag`, `CVSpgmr`, `CVSpbcg`, and `CVSptfqmr`.:

   ```
   cvode.CVDense(cvode_mem, 3)
   ```

9. Set any optional linear solver inputs using `cvode.CV<solver>Set*`.

10. Optionally initialise root finding passing the CVODE object, the number of roots to find, a vector of size equal to the number of roots, and a pointer to any optional user data you want available in your root finding function. The root finding function should populate a vector of root values, generally using implicit algebraic equations. If any of those values are zero the integrator pauses, returning `cvode.CV_ROOT_RETURN` to indicate that at least one root has been found.:

    ```
    cvode.CVodeRootInit(cvode_mem, 2, rootfind, None)
    ```

11. Advance the solution in time, calling `cvode.CVode` for each desired output time step. Each call to `cvode.CVode` specifies the desired time for the next stop (`tout`) and the current conditions (`y`). On return, `y` will contain the new conditions, and `t` will contain the time at which the integrator stopped. `t`, which must be of type `realtype` and passed into `cvode.CVode` by reference, can be different from `tout` if roots are found, or errors encountered. The last parameter specifies how CVODE should step. See the SUNDIALS documentation for more details.:

```
t = cvode.realtype(0)
tout = 0.4
while tout < 0.4*(10**12):
    flag = cvode.CVode(cvode_mem,
        tout,
        y,
        ctypes.byref(t),
        cvode.CV_NORMAL
    )
    print (t, y)
    if flag == cvode.CV_ROOT_RETURN:
        rootsfound = cvode.CVodeGetRootInfo(cvode_mem, 2)
        print rootsfound
    elseif flag == cvode.CV_SUCCESS:
        tout *= 10
    else:
        break
```

B.4.4 CVODES

Programs using CVODES will generally conform to a certain skeleton layout very similar to that of CVODE. Our layout here provides an example for simple calculation of sensitivities using forward sensitivity analysis. CVODES is capable of adjoint sensitivity analysis to. See the function reference or the SUNDIALS documentation for information of how to uses these alternative methods.

1. Import the cvodes module, the nvecserial module, and the ctypes module:

```
from pysundials import cvodes
import nvecserial
import ctypes
```

2. Define a structure to hold your parameters for which you wish to calculate sensitivities as well as any optional user data.:

```
class UserData(ctypes.Structure):
    _fields_ = [
        ('p', cvodes.realtype*4)
    ]
PUserData = ctypes.POINTER(UserData)
```

3. Define your right-hand side function. This function must take exactly four parameters. The first parameter will be the current value of the independent variable (usually time). The second parameter will be an `NVector` containing the current values of the dependent variables. The third parameter is an `NVector` whose elements must be filled with the new values of the dependent variables. The fourth parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system. For example, a simple problem consisting of three variables and having the following ODES:

   - $v_1 = r_2 - r_1$
   - $v_2 = r_1 - r_2$
   - $v_3 = r_1 - r_3 - r_4$

   (where r_i is a function of the independent variable, the current values of the dependent variables and the parameter set) would have the following RHS function.:

```
def f(t, y, ydot, f_data):
    data = ctypes.cast(f_data, PUserData).contents
    ydot[0] = r2(t,y,data.p) - r1(t,y,data.p)
    ydot[1] = r1(t,y,data.p) - r2(t,y,data.p)
    ydot[2] = r1(t,y,data.p) - r3(t,y,data.p)\
       - r4(t,y,data.p)
    return 0
```

4. Define any optional functions such as a Jacobian approximation, error weight and/or root finding functions. See function reference for details on parameters and returns.:

```
def rootfind(t, y, gout, g_data):
    gout[0] = y[0] - 0.5
    gout[1] = y[1] - 0.5
    return 0
```

5. Initialise an `NVector` with the initial conditions.:

```
y = cvodes.NVector([0.7, 0.3, 0.0])
```

6. Create a CVODE object.:

   ```
   cvode_mem = cvodes.CVodeCreate(lmm, iter)
   ```

   (where *lmm* is on of `cvodes.CV_ADAMS` or `cvodes.CV_BDF`, and *iter* is one of `cvodes.CV_NEWTON` or `cvodes.CV_FUNCTIONAL`)

7. Allocate integrator memory, set the initial value of the independent variable, and set tolerances. Absolute tolerances may be an `NVector` of the same size as `y` in which case `cvodes.CV_SV` should be used, or a scalar value applying to all (`cvodes.CV_SS`).:

   ```
   abstol = cvodes.NVector([1.0e-8, 1.0e-14, 1.0e-6])
   reltol = cvodes.realtype(1.0e-4)
   cvodes.CVodeMalloc(cvode_mem, f, 0.0, y, cvodes.CV_SV,
       reltol, abstol)
   ```

8. Set any optional inputs using `CVSet*()`.:

   ```
   cvodes.CVodeSetFdata(cvode_mem, ctypes.pointer(data))
   ```

9. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `CVDense`, `CVBand`, `CVDiag`, `CVSpgmr`, `CVSpbcg`, and `CVSptfqmr`.:

   ```
   cvodes.CVDense(cvode_mem, 3)
   ```

10. Set sensitivity system options by first creating an NVectorArray of dimensions v by p, where v is the number of variables, and p is the number of parameters for which sensitivities will be calculated.:

    ```
    yS = nvecserial.NVectorArray([([0]*2)]*4)
    ```

    Next call `cvodes.CVodeSensMalloc` to allocate and initialise required memory for sensitivity analysis, passing the CVODE object, the number of parameters, the desired method (`cvodes.CV_SIMULTANEOUS`, `cvodes.CV_STAGGERED`, or `cvodes.CV_STAGGERED1`), and the NVectorArray.:

    ```
    cvodes.CVodeSensMalloc(cvodes_mem, 4,
        cvodes.CV_SIMULTANEOUS, yS)
    ```

Next we have to inform CVODES which parameters are going to be used for sensitivity calculations by calling `cvodes.CVodeSetSensParams`, which expects four parameters (for more detail see p. 111 of the CVODES user guide).

a) the CVODES memory object

b) a pointer to the array of parameter values which MUST be passed through the user data structure (so CVODES knows where the values are and can perturb them, presumably)

c) an array (i.e. list) of scaling factors, one for each parameter for which sensitivities are to be determined

d) an array of integers (either 1 or 0), where a 1 indicates the respective parameter value should be used in estimating sensitivities

for example:

```
cvodes.CVodeSetSensParams(cvodes_mem,
    data.p, #we have four system parameters (The four VMax's)
    [1]*4, #all are scaled by 1, i.e. unscaled,
    [1]*4 #all contribute to estimation of sensitivities
)
```

11. Set any optional linear solver inputs using `cvodes.CV<solver>Set*`.

12. Optionally initialise root finding passing the CVODE object, the number of roots to find, a vector of size equal to the number of roots, and a pointer to any optional user data you want available in your root finding function. The root finding function should populate a vector of root values, generally using implicit algebraic equations. If any of those values are zero the integrator pauses, returning `cvodes.CV_ROOT_RETURN` to indicate that at least one root has been found.:

```
cvodes.CVodeRootInit(cvode_mem, 2, g, None)
```

13. Advance the solution in time, calling `cvodes.CVode` for each desired output time step. Each call to `cvodes.CVode` specifies the desired time for the next stop (`tout`) and the current conditions (`y`). On return, `y` will contain the new conditions, and `t` will contain the time at which the integrator stopped. `t`, which must be of type `realtype` and passed into `cvodes.CVode` by reference, can be different from `tout` if roots are found, or errors encountered. The last parameter specifies how CVODE should step. See the SUNDIALS documentation for more details.:

```
t = cvodes.realtype(0)
tout = 0.4
while tout < 0.4*(10**12):
    flag = cvodes.CVode(
        cvode_mem,
        tout,
        y,
        ctypes.byref(t),
        cvodes.CV_NORMAL
    )
    cvodes.CVodeGetSens(cvodes_mem, t, yS)
    print (t, y, yS)
    if flag == cvodes.CV_ROOT_RETURN:
        rootsfound = cvodes.CVodeGetRootInfo(cvode_mem, 2)
        print rootsfound
    elseif flag == cvodes.CV_SUCCESS:
        tout *= 10
    else:
        break
```

B.4.5 IDA

Programs using IDA will generally conform to a certain skeleton layout. The example used here serves to illustrate this skeleton layout, and is neither complete, nor representative of SUNDIALS/PySUNDIALS complete set of capabilities. Please see the function reference or SUNDIALS documentation for more information.

1. Import the ida and ctypes modules.:

   ```
   from pysundials import ida
   import ctypes
   ```

2. Define your right-hand side function. This function must take exactly five parameters. The first parameter will be the current value of the independent variable (usually time). The second parameter will be an `NVector` containing the current values of the dependent variables. The third parameter will be an `NVector` containing *dy/dt*. The fourth parameter is an `NVector` whose elements must be filled with the new values of the dependent variables. The fifth parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system, and must do so in implicit form for both differential and algebraic equations. Additionally, those variables

determined by algebraic relations should appear strictly after those determined by differential equations in the dependent variable vector. For example, a simple problem consisting of three variables and having the following ODES (note the rearrangement of the order of differential and algebraic equations compared to previous examples):

- $v_1 = r_1 - r_3 - r_4$
- $v_2 = r_2 - r_1$
- $v_3 = r_1 - r_2$

(where r_i is a function of the independent variable and the current values of the dependent variables) would have the following RHS function:

```
def f(t, yy, yp, rr, data):
    rr[0] = r1(yy)-r3(yy)-r4(yy)-yp[0]
    rr[1] = r2(yy)-r1(yy)-yp[1]
    rr[2] = yy[1]+yy[2]-1
    return 0
```

3. Define any optional functions such as a Jacobian approximation, error weight and/or root finding functions. See function reference for details on parameters and returns.:

```
def rootfind(t, y, gout, g_data):
    gout[0] = y[0] - 0.5
    gout[1] = y[1] - 0.5
    return 0
```

4. Initialise an `NVector` with the initial conditions.:

```
yy = ida.NVector([0.7, 0.3, 0.0])
```

5. Initialise another `NVector` with the initial derivative conditions.:

```
yp = ida.NVector([r1(yy)-r3(yy)-r4(yy),r2(yy)-r1(yy),1-yy[1]])
```

6. Create an IDA object.:

```
ida_mem = ida.IDACreate()
```

7. Allocate integrator memory, set the initial value of the independent variable, and set tolerances. Absolute tolerances may be an `NVector` of the same size as `y` in which case `ida.IDA_SV` should be used, or a scalar value applying to all (`ida.IDA_SS`).:

```
abstol = ida.NVector([1.0e-8, 1.0e-14, 1.0e-6])
reltol = ida.realtype(1.0e-4)
ida.IDAMalloc(ida_mem, f, 0.0, yy, yp, ida.IDA_SV,
    reltol, abstol)
```

8. Set any optional inputs using `IDASet*()`.

9. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `IDADense`, `IDABand`, `IDASpgmr`, `IDASpbcg`, and `IDASptfqmr`.:

```
ida.IDADense(ida_mem, 3)
```

10. Set any optional linear solver inputs using `ida.IDA<solver>Set*`.

11. Optionally initialise root finding passing the IDA object, the number of roots to find, a vector of size equal to the number of roots, and a pointer to any optional user data you want available in your root finding function. The root finding function should populate a vector of root values, generally using implicit algebraic equations. If any of those values are zero the integrator pauses, returning `ida.IDA_ROOT_RETURN` to indicate that at least one root has been found.:

```
ida.IDARootInit(ida_mem, 2, rootfind, None)
```

12. Advance the solution in time, calling `ida.IDASolve` for each desired output time step. Each call to `ida.IDASolve` specifies the desired time for the next stop (`tout`) and the current conditions (`y`). On return, `y` will contain the new conditions, and `t` will contain the time at which the integrator stopped. `t`, which must be of type `realtype` and passed into `ida.IDASolve` by reference, can be different from `tout` if roots are found, or errors encountered. The last parameter specifies how IDA should step. See the SUNDIALS documentation for more details.:

```
t = ida.realtype(0)
tout = 0.4
while tout < 0.4*(10**12):
    flag = ida.IDASolve(
        ida_mem,
        tout,
        ctypes.byref(t),
        yy,
        yp,
```

```
        ida.IDA_NORMAL
    )
    print (t, yy)
    if flag == ida.IDA_ROOT_RETURN:
        rootsfound = ida.IDAGetRootInfo(ida_mem, 2)
        print rootsfound
    elseif flag == ida.IDA_SUCCESS:
        tout *= 10
    else:
        break
```

B.4.6 KINSOL

Programs using KINSOL will generally conform to a certain skeleton layout. The example used here serves to illustrate this skeleton layout, and is neither complete, nor representative of SUNDIALS/PySUNDIALS complete set of capabilities. Please see the function reference or SUNDIALS documentation for more information.

1. Import the kinsol and ctypes modules.:

   ```
   from pysundials import kinsol
   import ctypes
   ```

2. Define your right-hand side function. This function must take exactly three parameters. The first parameter will be an `NVector` containing the current values of the dependent variables. The second parameter is an `NVector` whose elements must be filled with the new values of the dependent variables. The third parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system and must do so using strictly linearly independent equations. For example, a simple problem consisting of three variables and having the following ODES:

   - $v_1 = r_1 - r_3 - r_4$
   - $v_2 = r_2 - r_1$
   - $v_3 = r_1 - r_2$

   (where r_i is a function of the independent variable and the current values of the dependent variables) would have the following RHS function, ignoring v_3 because it is linearly dependent with v_2:

   ```
   def f(u, fval, f_data):
   ```

```
    fval[S2] = R2(u) - R1(u)
    fval[S1] = R1(u) - R3(u) - R4(u)
    return 0
```

3. Define any optional functions such as a Jacobian approximation, and/or error weight functions. See function reference for details on parameters and returns.

4. Initialise an `NVector` with an initial guess.:

   ```
   u = kinsol.NVector([1.0, 0.7])
   ```

5. Initialise a template `NVector` of the same size as your dependent variable vector.:

   ```
   template = kinsol.NVector([0, 0])
   ```

6. Initialise a scaling vector or vectors as necessary. See function reference for `kinsol.KINSol` for more details:

   ```
   s = kinsol.NVector([1, 1])
   ```

7. Create a KINSOL object.:

   ```
   kin_mem = kinsol.KINCreate()
   ```

8. Allocate solver memory, and set the RHS function and size of the system using the template vector.:

   ```
   kinsol.KINMalloc(kin_mem, f, template)
   ```

9. Set any optional inputs using `KINSet*()`.

10. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `KINDense`, `KINBand`, `KINSpgmr`, `KINSpbcg`, and `KINSptfqmr`.:

    ```
    kinsol.KINDense(kin_mem, 2)
    ```

11. Set any optional linear solver inputs using `kinsol.KIN<solver>Set*`.

12. Solve the problem by calling `kinsol.KINSol`, passing the KINSOL memory object, the vector with the initial guess, the globalisation strategy (one of `kinsol.KIN_NONE` or `kinsol.KIN_LINESEARCH`), and two scaling vectors, *u_scale* and *f_scale*. In our case no scaling is applied via the repeated use of the scaling vector `s` (*[1,1]*):

    ```
    kinsol.KINSol(kin_mem, u, kinsol.KIN_LINESEARCH, s, s)
    ```

Bibliography

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D., 1999. LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd Edition. 15

Backus, J., Beeber, R., Best, S., Goldberg, R., Haibt, L., Herrick, H., Nelson, R., Sayre, D., Sheridan, P., Stern, H., Ziller, I., Hughs, R., Nutt, R., 1957. Fortran automatic coding system. In: Proceedings of the Western Joint Computer Conference. Los Angeles, CA. 19

Boderke, P., Schittkowski, K., Wolf, M., Merkle, H. P., 2000. Modeling of diffusion and concurrent metabolism in cutaneous tissue. J Theor Biol 204 (3), 393–407. 4

Brown, P. N., Saad, Y., 1990. Hybrid Krylov methods for nonlinear systems of equations. SIAM J Sci Stat Comp 11, 450. 24

Cleland, W. W., 1967. The statistical analysis of enzyme kinetic data. Advan Enzymol Relat Areas Mol 29, 1–32. 7

Cohen, S. D., Hindmarsh, A. C., 1996. CVODE, a stiff/nonstiff ODE solver in C. Comput Phys 10 (2), 138–143. 23

Cornish-Bowden, A., Hofmeyr, J.-H. S., 1991. MetaModel: a program for modelling and control analysis of metabolic pathways on the IBM PC and compatibles. Comput Appl Biosci 7 (1), 89–93. 16

Curtis, A. R., 1976. FACSIMILE–a computer program for simulation and optimization. Biochem Soc T 4 (2), 364–71. 15

Downey, A. B., Elkner, J., Meyers, C., 2002. How to Think Like a Computer Scientist: Learning with Python. Green Tea Press, http://www.greenteapress.com. 21

Drummond, L. A., Hernandez, V., Marques, O., Roman, J. E., Vidal, V., 2005. A Survey of High-Quality Computational Libraries and Their Impact in Science and Engineering Applications. Vol. 3402 of Lecture Notes in Computer Science. Springer, Berlin, pp. 37–50. 20

Eaton, J. W., 2002. GNU Octave Manual. Network Theory Limited, Bristol. 15

Forster, H., Schrefl, T., Suess, D., Scholz, W., Tsiantos, V., Dittrich, R., Fidler, J., 2002. Domain wall motion in nanowires using moving grids (invited). J Appl Phys 91, 6914. 25

Gauld, A., 2000. Learn to Program Using Python: A Tutorial for Hobbyists, Self-Starters, and All Who Want to Learn the Art of Computer Programming. Addison-Wesley Professional. 21

Gilat, A., 2004. MATLAB: An Introduction with Applications 2nd Edition. Wiley Publishing Inc., New Jersey. 15

Hanekom, A., 2006. Generic kinetic equations for modelling multisubstrate reactions in computational systems biology. Master's thesis, University of Stellenbosch. 7

Heinrich, R., Rapoport, T., 1974. A linear steady-state treatment of enzymatic chains. general properties, control, and effector strength. Eur J Biochem 42, 89–95. 15

Hindmarsh, A., 1983. ODEPACK, A Systematized Collection of ODE Solvers. In: Stepleman, R. e. a. (Ed.), Scientific Computing: Applications of Mathematics and Computing to the Physical Sciences. Vol. 1 of IMACS Transactions on Scientific Computing. North-Holland, Amsterdam, Netherlands; New York, U.S.A., pp. 55–64. 23

Hindmarsh, A. C., 2000. The PVODE and IDA algorithms. Tech. rep., Lawrence Livermore National Lab., California. 24

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., Woodward, C. S., 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM T Math Software 31 (3), 363–396. 20, 23

Hofmeyr, J.-H. S., 2001. Metabolic control analysis in a nutshell. In: Proceedings of the 2nd International Conference on Systems Biology. p. 291300. 15, 44

Hofmeyr, J.-H. S., 2007. The biochemical factory that autonomously fabricates itself: A systems-biological view of the living cell. In: Boogerd, F. C., Bruggeman, F., Hofmeyr, J.-H. S., Westerhoff, H. V. (Eds.), Systems Biology: Philosophical Foundations. Elsevier, Ch. 10, pp. 217–242. 1

Hofmeyr, J.-H. S., Cornish-Bowden, A., 1997. The reversible Hill equation: how to incorporate cooperative enzymes into metabolic models. Comput Appl Biosci 13 (4), 377–385. 7

Hofmeyr, J.-H. S., 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 (4), 243–249. 15

Holzhutter, H. G., Colosimo, A., 1990. SIMFIT: a microcomputer software-toolkit for modelistic studies in biochemistry. Comput Appl Biosci 6 (1), 23–28. 16

Hommes, F., 1962. The integrated Michaelis-Menten equation. Arch Biochem Biophys 96, 28–31. 15

Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., Kummer, U., 2006. COPASI–a COmplex PAthway SImulator. Bioinformatics 22 (24), 3067. 16

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., , the rest of the SBML Forum:, 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. L., 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. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., Wang, J., 2003. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. Bioinformatics 19 (4), 524–531. 18

Ingalls, B. P., 2004. A frequency domain approach to sensitivity analysis of biochemical systems. J Phys Chem B 108, 1143–52. 15

Jones, E., Oliphant, T., Peterson, P., et al., 2001. SciPy: Open source scientific tools for Python.
URL `http://www.scipy.org/` 15, 21

Kacser, H., Burns, J. A., 1973. The control of flux. Symp Soc Exp Biol 32, 65–104. 15

Kernighan, B. W., Ritchie, D. M., 1978. The C programming language, 1st Edition. Prentice-Hall, Englewood Cliffs, NJ. 20

Kholodenko, B., Bruggeman, F., Sauro, H., 2005. Systems Biology. Springer, Berlin, pp. 143–159. 1

King, E. L., Altman, C., 1956. A schematic method of deriving the rate laws for enzyme-catalyzed reactions. J Phys Chem 60 (10), 1375–1378. 7

Kiusalaas, J., 2005. Numerical Methods in Engineering with Python. Cambridge University Press. 21

Kootsey, J., Kohn, M., Feezor, M., Mitchell, G., Fletcher, P., 1986. SCoP: An interactive simulation control program for micro- and minicomputers. B Math Biol 48 (3), 427–441. 15

Langtangen, H. P., 2004. Python Scripting for Computational Science, 1st Edition. Springer, Berlin. 21

Le Novre, N., Bornstein, B., Broicher, A., Courtot, M., Donizelli, M., Dharuri, H., Li, L., Sauro, H., Schilstra, M., Shapiro, B., 2006. BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. Nucleic Acids Res 34, D689–D691. 49

Lefever, R., Nicolis, G., Borckmans, P., 1988. The brusselator: it does oscillate all the same. J Chem Soc Farad T 1 84 (4), 1013–1023. 50

Letellier, T., Reder, C., Mazat, J.-P., 1991. CONTROL - software for the analysis of control of metabolic networks. Comput Appl Biosci 7, 383–390. 16

Loew, L. M., Schaff, J. C., 2001. The Virtual Cell: a software environment for computational cell biology. Trends Biotechnol 19 (10), 401–406. 19

Lutz, M., 2006. Programming Python, 3rd Edition. O'Reilly Media, Inc., California. 21

Martelli, A., 2003. Python in a Nutshell, 1st Edition. O'Reilly Media, Inc., California. 21

Maruch, S., Maruch, A., 2006. Python For Dummies. Wiley Publishing Inc., New Jersey. 21

Mendes, P., 1993. GEPASI: a software package for modelling the dynamics, steady states and control of biochemical and other systems. Bioinformatics 9 (5), 563–571. 16

Michaelis, L., Menten, M., 1913. Der Kinetik der Invertinwirkung. Biochem Z 49, 333–369. 7

Modin, K., Fritzson, D., Fhrer, C., Sderlind, G., 2005. A new class of variable step-size methods for multibody dynamics. In: Multibody Dynamics 2005, ECCOMAS Thematic Conference. Madrid. 25

Monniaux, D., 2008. The pitfalls of verifying floating-point computations. ACM T Progr Lang Syst 30 (3), 1–41. 42

Monod, J., Wyman, J., Changeux, J. P., 1965. On the nature of allosteric transitions: A plausible model. J Mol Biol 12, 88–118. 7

Myers, C., Barker, N., Kuwahara, H., Madsen, C., Nguyen, N., 2008. IBioSim: Myers research group.
URL `http://www.async.ece.utah.edu/iBioSim/` 16

Noble, D., 2006. The Music of Life. 1

Olivier, B. G., Rohwer, J. M., Hofmeyr, J.-H. S., 2005. Modelling cellular systems with PySCeS. Bioinformatics 21 (4), 560–1. 17

Olivier, B. G., Snoep, J. L., 2004. Web-based kinetic modelling using JWS Online. Bioinformatics 20 (13), 2143–4. 17

Pettinen, A., Aho, T., Smolander, O.-P., Manninen, T., Saarinen, A., Taattola, K.-L., Yli-Harja, O., Linne, M.-L., 2005. Simulation tools for biochemical networks: evaluation of performance and usability. Bioinformatics 21 (3), 357–363. 16, 24

Petzold, L., Hindmarsh, A., 1997. LSODA (Livermore Solver of Ordinary Differential Equations). Computing and Mathematics Research Division, Lawrence Livermore National Laboratory, Livermore, CA. 24

Pfeiffer, T., 1999. METATOOL: for studying metabolic networks. Bioinformatics 15 (3), 251–257. 19

Poolman, M. G., 2006. ScrumPy: metabolic modelling with Python. Systems Biol 153 (5), 375–8. 16

Roman, G. C., Garfinkel, D., 1978. BIOSSIM–a structured machine-independent biological simulation language. Comput Biomed Res 11 (1), 3–15. 15

Rudiger, S., Shuai, J. W., Huisinga, W., Nagaiah, C., Warnecke, G., Parker, I., Falcke, M., 2007. Hybrid stochastic and deterministic simulations of calcium blips. Biophys J 93 (6), 1847–1857. 4

Sauer, U., Heinemann, M., Zamboni, N., 2007. GENETICS: Getting closer to the whole picture. Science 316 (5824), 550–551. 1

Sauro, H. M., 1993. SCAMP: a general-purpose simulator and metabolic control analysis program. Bioinformatics 9 (4), 441–450. 16

Sauro, H. M., 2000. Jarnac: A system for interactive metabolic analysis. In: Animating the Cellular Map: Proceedings of the 9th International Meeting on Bio-ThermoKinetics. Stellenbosch University Press. ISBN 0-7972-0776-7. 16

Savageau, M. A., 1969. Biochemical systems analysis. ii. the steady-state solutions for an n-pool system using a power-law approximation. J Theor Biol 25 (3), 370–9. 8

Schmidt, H., Jirstrand, M., 2006. Systems Biology Toolbox for MATLAB: a computational platform for research in systems biology. Vol. 22. Oxford Univ Press. 17

Serban, R., Hindmarsh, A. C., 2005. CVODES, the sensitivity-enabled ODE solver in SUNDIALS. In: Proceedings of IDETC/CIE. 24

Snoep, J., Westerhoff, H., 2005. From isolation to integration, a systems biology approach for building the Silicon Cell. In: Systems Biology. Springer, Berlin, pp. 13–30. 1

Stroustrup, B., 1986. The C++ Programming Language, 3rd Edition. 20

Surovstev, I. V., Morgan, J. J., Lindahl, P. A., 2007. Whole-cell modeling framework in which biochemical dynamics impact aspects of cellular geometry. J Theor Biol 244 (1), 154–166. 60, 61, 62

Urner, K., 2004. Python in the mathematics curriculum. In: PyConDC2004. Washington, D.C. 21

van Rossum, G., 2003a. An Introduction to Python. Network Theory Ltd., Bristol. 21

van Rossum, G., 2003b. The Python Language Reference Manual. Network Theory Ltd., Bristol. 21

Walter, C., 1966. Quasi-steady state in a general enzyme system. J Theor Biol 11 (2), 181–206. 15

Walter, C. F., Morales, M. F., 1964. An analogue computer investigation of certain issues in enzyme kinetics. J Biol Chem 239 (4), 1277–1283. 15

Wikipedia Contributors, 2008. Systems biology. Revision 250433403. URL `http://en.wikipedia.org/w/index.php?title=Systems_biology\&oldid=250433403` 1

Wolfram, S., 1991. Mathematica: a system for doing mathematics by computer. Addison Wesley Professional, Redwood City, CA. 15

Zeilinger, M. N., Farr, E. M., Taylor, S. R., Kay, S. A., III, F. J. D., 2006. A novel computational model of the circadian clock in arabidopsis that incorporates prr7 and prr9. Mol Syst Biol 2 (58). 49

Zelle, J. M., 2003. Python Programming: An Introduction to Computer Science. Franklin Beedle & Associates, Wisonville, OR. 21