Systems Biology: Modeling with Python

First Edition

Herbert M. Sauro
University of Washington
Seattle, WA

Ambrosius Publishing

First Edition, version 1.00
Published by Ambrosius Publishing and Future Skill Software
www.analogmachine.org

Typeset using LaTeX 2_ε, TikZ, PGFPlots, WinEdt, InkScape, and
11pt Math Time Professional 2 Fonts

ISBN 13: xxxx (paperback)
ISBN-10: xxxxx (paperback)

Printed in the United States of America.

Mosaic image modified from Daniel Steger's Tikz image (http://www.texample.net/tikz/examples/mosaic-from-pompeii/

Contents

1 Kinetics in a Nutshell **1**

1.1 Introduction 1

1.2 Definitions 1

1.3 Elementary Mass-Action Kinetics 3

1.4 Chemical Equilibrium 3

1.5 Mass-action and Disequilibrium Ratio 5

1.6 Modified Mass-Action Rate Laws 5

Further Reading 6

1.7 Exercises 7

2 Enzyme Kinetics in a Nutshell **9**

2.1 Michaelis-Menten Kinetics 9

2.2 Reversible Rate laws 10

2.3 Haldane Relationship 11

2.4 Competitive Inhibition 12

2.5 Cooperativity 13

2.6 Allostery 15

2.7 Elasticities 17

Further Reading 18

2.8 Exercises 18

3 Quick Introduction to Python **19**

3.1 Introduction to Python 20

3.1.1 Running Commands from the Console 21

3.1.2 Repeating Calculations 23

3.1.3 Making Decisions: Conditionals 25

3.1.4 Creating Functions in Python . . . 25
3.2 Brief Introduction to Numpy . . . 27
3.2.1 Creating Arrays . . . 28
3.2.2 Indexing Elements . . . 28
3.2.3 Extracting Rows and Columns . . . 28
3.2.4 Stacking Arrays . . . 29
3.3 Plotting Graphs . . . 30
3.3.1 Basic Plotting . . . 30
3.3.2 Changing Markers and Line Styles . . . 31
3.3.3 Changing Colors . . . 32
3.3.4 Plotting Functions . . . 33
3.3.5 Arithmetic using Numpy . . . 34
3.4 Exercises . . . 34

4 Networks in a Nutshell 37
4.1 Mass Conservation . . . 37
4.2 Exercise . . . 38
4.3 Modeling Gene Networks . . . 39
4.4 Exercises . . . 41

5 Entering Models 43
5.1 Describing Reaction Networks using Antimony . . . 43
5.1.1 Initializing of Model Values . . . 46
5.1.2 Setting up Compartments . . . 46
5.2 Including Additional Equations with the Model . . . 47
5.3 Loading and Running Models in Python . . . 48
5.3.1 Loading Antimony Models . . . 49
5.4 Models as Differential Equations . . . 49
5.5 Exercises . . . 50

6 Sharing Models 51
6.1 Sharing Models . . . 51
6.1.1 Loading SBML Models . . . 53
6.1.2 Loading Models from Biomodels . . . 54

6.2 Generating SBML and Matlab Files 54
6.2.1 Exporting SBML 54
6.2.2 Exporting Matlab 54
6.3 Test models 55
6.4 Exercises 55

7 Running a Simulation **57**
7.1 Time Course Simulation 57
7.1.1 Plotting Simulation Results 58
7.1.2 Setting and Getting Values 59
7.1.3 Selecting Output from a Simulation 60
7.1.4 Resetting Simulations 61
7.1.5 Access to Fluxes or Rates of Reaction 61
7.1.6 Access to Rates of Change 62
7.1.7 Applying Perturbations to a Simulation 63
7.1.8 Using Antimony to Implement Events 64
7.2 Perturbations and Events 65
7.3 Other Model Properties of Interest 65
7.4 More on Plotting 66
7.4.1 Controlling Axes etc 67
7.5 Exercises 68

8 Steady State Analysis **71**
8.1 Steady State 71
8.1.1 Stability Analysis 72
8.2 Metabolic Control Analysis 73
8.2.1 Control Coefficients 74
8.2.2 Elasticity Coefficients 74
8.3 Exercises 75

9 Running Parameter Scans **77**
9.1 Introduction 77

10 Model Fitting **81**

10.1 Introduction 81
10.2 Fitting Models 81
10.3 Optimization Algorithms 84

11 Stoichiometric Analysis **87**
11.1 Stoichiometric Analysis 87
11.1.1 Stoichiometry Matrix 87
11.1.2 Conservation Matrix 88

12 Projects **89**

References **91**

History **93**

1 Kinetics in a Nutshell

1.1 Introduction

Understanding chemical kinetics is at the heart of building biochemical models. This chapter gives a minimal introduction to some of the essential concepts of elementary chemical kinetics. A fuller account is given in the companion book, 'Enzyme Kinetics for Systems Biology'. This chapter may be omitted by those already familiar with this topic.

1.2 Definitions

Reaction kinetics is the study of how fast chemical reactions take place, what factors influence the rate of reaction, and what mechanisms are responsible.

Stoichiometric Amount

The **stoichiometric amount** is the number of molecules for a particular reactant or products takes part in a given reaction reaction. For example:

$$2A + 3B \rightarrow A + 3C$$

In the above example the stoichiometric amount for reactant A is 2 and for B is 3. The stoichiometric amount for product A is 1 and for C is 3.

Depicting Reactions

$$aA + bB + \ldots \rightarrow +pP + qQ + \ldots$$

where $a, b, \ldots, p, q, \ldots$ are stoichiometric amounts.

Rates of Change

The rate of change is defined as the rate of change in concentration or amount of a designated molecular species.

$$\text{Rate of Change} = \frac{dS}{dt}$$

Stoichiometric coefficients

The **stoichiometric coefficient**, c_i, for a molecular species A_j, is the difference between the molar amount of species, i – also called the **stoichiometric amount** – on the product side and the molar amount of the same species on the reactant side.

$$c_i = \text{Molar Amount of Product} - \text{Molar Amount of Reactant}$$

In the reaction, $2A \longrightarrow B$, the molar amount of A on the product side is zero while on the reactant size it is two. Therefore the stoichiometric coefficient of A is given by $0-2 = -2$. In many cases a particular species will only occur on the reactant or product side but it is not uncommon to find situations where a species occurs simultaneously as a product and a reactant. As a result, reactant stoichiometric coefficients tend to be *negative* while product stoichiometric coefficients tend to be *positive*.

Reaction Rates

The **reaction rate**, often denoted by the symbol v, is measured with respect to a given molecular species normalized by the species stoichiometric coefficient. This definition ensures that no matter which molecular species in a reaction is measured, the reaction rate is uniquely defined for that reaction. More formally, the reaction rate for the given reaction is:

$$aA + bB + \ldots \rightarrow pP + qQ + \ldots$$

$$v = \frac{1}{c_a}\frac{dA}{dt} = -\frac{1}{c_b}\frac{dB}{dt} \ldots = \frac{1}{c_p}\frac{dP}{dt} = \frac{1}{c_q}\frac{dQ}{dt} \ldots$$

where c_x are the stoichiometric coefficients. Alternatively, we can express the rate of change in terms of the reaction rate as:

$$\frac{dA}{dt} = c_a v \tag{1.1}$$

1.3 Elementary Mass-Action Kinetics

An elementary reaction is one that cannot be broken down into simpler reactions. Such reactions will often display simple kinetics called mass-action kinetics. For a reaction of the form:

$$aA + bB + \ldots \rightarrow +pP + qQ + \ldots$$

the mass-action kinetic rate law is given by:

$$v = k_1 A^a B^b \ldots - k_2 P^p Q^q \ldots$$

where k_1 and k_2 are the forward and reverse rate constants, respectively. In the case of reaction:

$$2\text{ADP} \rightarrow \text{ATP} + \text{AMP} \tag{1.2}$$

The reversible mass-action rate law would be written as:

$$v = k_1 \text{ ADP}^{\,2} - k_2 \text{ ATP AMP}$$

In all mass-action rate laws, the units for the reactant and product terms must be expressed in concentration. The units for the rate constants, k will depend on the exact form of the rate law but must be set to ensure that the rate of reaction is expressed in units moles $L^{-1}t^{-1}$.

1.4 Chemical Equilibrium

In principle, all reactions are reversible, meaning transformations can occur from reactant to product or product to reactant. The net rate of a reversible reaction is the difference between the forward and reverse rates. Given a reversible reaction such as:

$$A \rightleftharpoons B$$

we can observe the concentrations of A and B approach equilibrium (Figure 1.3).

At chemical equilibrium the forward and reverse rates are equal and is described by the relation:

$$\frac{k_1}{k_2} = \frac{B}{A} = K_{eq} \tag{1.3}$$

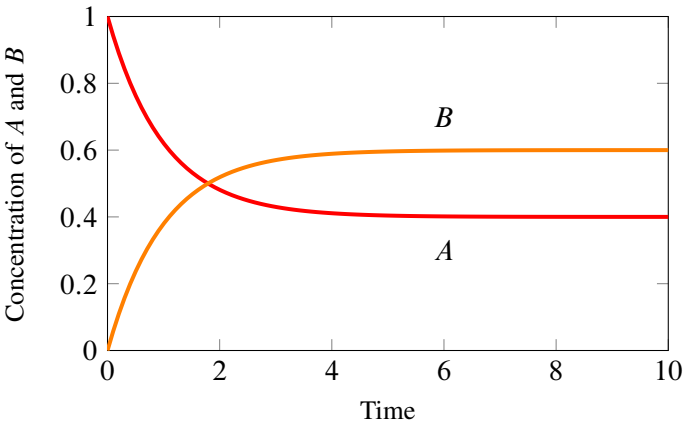


Figure 1.1 Approach to equilibrium for the reaction $A \rightleftharpoons B$, $k_1 = 0.6$, $k_2 = 0.4$, $A(0) = 1$, $B(0) = 0$. Progress curves calculated from the solution to the differential equation $dA/dt = k_2 B - k_1 A$.

This ratio has special significance and is called the **equilibrium constant**, denoted by K_{eq}. The equilibrium constant is also related to the ratio of the rate constants, k_1/k_2. For a general reversible reaction such as:

$$aA + bB + \ldots \rightleftharpoons pP + qQ + \ldots$$

and using arguments similar to those described above, the ratio of the rate constants can be easily shown to be:

$$K_{eq} = \frac{P^p Q^q \ldots}{A^a B^b \ldots} = \frac{k_1}{k_2} \tag{1.4}$$

where the exponents are the stoichiometric *amounts* for each species.

For a bimolecular reaction such as:

$$\mathrm{HA} \rightleftharpoons \mathrm{H} + \mathrm{A}$$

chemists and biochemists will often distinguish between two kinds of equilibrium constants called association and dissociation constants. Thus the equilibrium constant for the above bimolecular reaction is often called the **dissociation constant**, K_d:

$$K_d = \frac{\mathrm{H} \cdot \mathrm{A}}{\mathrm{HA}}$$

to indicate the degree that the complex is dissociated into its component molecules at equilibrium. The **association constant**, K_a, though less commonly used, describes the equilibrium constant for the reverse process $\mathrm{H} + \mathrm{A} \rightleftharpoons \mathrm{HA}$, that is the formation of a complex from component molecules:

$$K_a = \frac{\mathrm{HA}}{\mathrm{H} \cdot \mathrm{A}}$$

It should be evident that:

$$K_d = \frac{1}{K_a} \tag{1.5}$$

The equilibrium constant is also related to the standard free energy change, $DeltaG^o$, such that:

$$\Delta G^o = -RT \ln K_{eq}$$

where R is the gas constant, and T the temperature. Rearranged we can also see that:

$$K_{eq} = e^{-\Delta G^o/RT} \tag{1.6}$$

1.5 Mass-action and Disequilibrium Ratio

Although in closed systems reactions tend to equilibrium, reactions occurring in living cells are generally out of equilibrium and the ratio of the products to the reactants *in vivo* is called the **mass-action ratio**, Γ. The ratio of the mass-action ratio to the equilibrium constant is called the **disequilibrium ratio**:

$$\rho = \frac{\Gamma}{K_{eq}} \tag{1.7}$$

At equilibrium the mass-action ratio will be equal to the equilibrium constant, that is $\rho = 1$. If the reaction is away from equilibrium ($B/A < K_{eq}$), then $\rho < 1$.

For a simple unimolecular reaction it was previously shown that the equilibrium ratio of product to reactant, B/A, is equal to the ratio of the forward and reverse rate constants. Substituting this into the disequilibrium ratio gives:

$$\rho = \Gamma \frac{k_2}{k_1} = \frac{B}{A}\frac{k_2}{k_1}$$

Therefore:

$$\rho = \frac{v_r}{v_f} \tag{1.8}$$

That is, the disequilibrium ratio is the ratio of the reverse and forward rates. If $\rho < 1$, the net reaction must be in the direction of product formation. If ρ is zero, the reaction is as out of equilibrium as possible with no product present.

1.6 Modified Mass-Action Rate Laws

A typical reversible mass-action rate law will require both the forward and the reverse rate constants to be fully defined. Often however, only one rate constant may be known. In these

circumstances it is possible to express the reverse rate constant in terms of the equilibrium constant.

For example, given the simple unimolecular reaction, $A \rightleftharpoons B$, it is possible to derive the following:

$$\begin{aligned} v &= k_1 A - k_2 B \\ v &= k_1 A \left(1 - \frac{k_2 B}{k_1 A}\right) \\ \text{Since } K_{eq} &= \frac{k_1}{k_2} \\ v &= k_1 A \left(1 - \frac{\Gamma}{K_{eq}}\right) \end{aligned} \tag{1.9}$$

where Γ is the mass-action ratio. This can be generalized to an arbitrary mass-action reaction to give:

$$v = k_1 A^a B^b \ldots \left(1 - \frac{\Gamma}{K_{eq}}\right) = k_1 A^a B^b \ldots (1 - \rho)$$

where $A^a B^b \ldots$ represents the product of all reactant species, a and b are the **corresponding** stoichiometric amounts, and ρ is the disequilibrium ratio. For example, for the reaction:

$$2A + B \longrightarrow C + 2D$$

where k_1 is the forward rate constant, the modified reversible rate law is:

$$v = k_1 A^2 B \, (1 - \rho)$$

The modified formulation demonstrates how a rate expression can be divided up into functional parts to include both kinetic and thermodynamic components [**?**]. The kinetic component is represented by the term $k_1 A^a B^b \ldots$, while the thermodynamic component is represented by the expression $1 - \rho$.

We can also derive the modified rate law in the following way. Given the net rate of reaction $v = v_f - v_r$, we can write this expression as:

$$v = v_f \left(1 - \frac{v_r}{v_f}\right)$$

That is:

$$v = v_f (1 - \rho)$$

Further Reading

1. Sauro HM (2012) Enzyme Kinetics for Systems Biology. 2nd Edition, Ambrosius Publishing ISBN: 978-0982477335

1.7 Exercises

1. Define the terms:

a) Stoichiometric amount

b) Rate of change

c) Stoichiometric coefficient

d) Reaction Rate

2. Write out the mass-action rate law for the following reactions, assume each reaction is irreversible.

a) $A + B \rightarrow C$

b) $2A + B \rightarrow C$

3. Write out the mass-action rate law for the following reactions, assume each reaction is reversible.

a) $A \rightarrow B$

b) $A + B \rightarrow C$

c) $2A + B \rightarrow 3C$

4. Write out the rate of change for each species in the following reactions, assume the reaction rate is given by v:

a) $A \rightarrow B$

b) $2A + B \rightarrow C$

c) $2A + B \rightarrow A + C$

5. Write out the equilibrium constant for the following reactions:

a) $A \rightarrow B$

b) $A + B \rightarrow C$

c) $2A + B \rightarrow 3C + D$

6. Define the terms:

a) Mass-action ratio

b) Disequilibrium ratio

7. A reaction is found to have a disequilibrium ration of 0.99, what can you say about the reaction?

2

Enzyme Kinetics in a Nutshell

Enzymes

Enzymes are protein molecules that can accelerate a chemical reaction without changing the reaction equilibrium constant.

Enzyme Kinetics

Enzyme kinetics is a branch of science that deals with the many factors that can affect the rate of an enzyme-catalysed reaction. The most important factors include the concentration of enzyme, reactants, products, and the concentration of any modifiers such as specific activators, inhibitors, pH, ionic strength, and temperature. When the action of these factors is studied, we can deduce the kinetic mechanism of the reaction. That is, the order in which substrates and products bind and unbind, and the mechanism by which modifiers alter the reaction rate.

2.1 Michaelis-Menten Kinetics

The standard model for enzyme action describes the binding of free enzyme to the reactant forming an **enzyme-reactant complex**. This complex undergoes a transformation, releasing product and free enzyme. The free enzyme is then available for another round of binding to new reactant.

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \xrightarrow{k_2} E + P \tag{2.1}$$

where k_1, k_{-1} and k_2 are rate constants, S is substrate, P is product, E is the free enzyme, and ES the enzyme-substrate complex.

By assuming a steady state condition on the enzyme substrate complex, we can derive the Briggs-Haldane equation relation (sometimes mistakenly called the Michaelis-Menten equation):

$$v = \frac{Vm\ S}{K_m + S} \tag{2.2}$$

where V_m is the maximal velocity, and K_m the substrate concentration that yields half the maximum velocity.

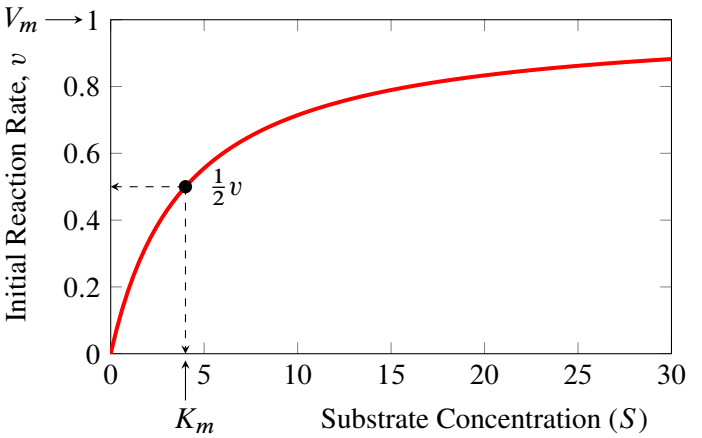


Figure 2.1 Relationship between the initial rate of reaction and substrate concentration for a simple Michaelis-Menten rate law. The reaction rate reaches a limiting value called the V_m. K_m is set to 4.0 and V_m to 1.0. The K_m value is the substrate concentration that gives half the maximal rate.

2.2 Reversible Rate laws

An alternative and more realistic model is the reversible form:

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \underset{k_{-2}}{\overset{k_2}{\rightleftharpoons}} E + P \tag{2.3}$$

The aggregate rate law for the reversible form of the mechanism can also be derived and is given by:

$$v = \frac{V_f\ S/K_S - V_r\ P/K_P}{1 + S/K_S + P/K_P} \tag{2.4}$$

Sometimes reactions appear irreversible, that is no discernable reverse rate is detected, and yet the forward reaction is influenced by the accumulation of product. This effect is

caused by the product competing with substrate for binding to the active site and is often called **product inhibition**. Given that product inhibition is a type of competitive inhibition we will briefly discuss it here. An important industrial example of this is the conversion of lactose to galactose by the enzyme $\beta-$galactosidase where galactose competes with lactose, slowing the forward rate [**?**].

To describe simple product inhibition with rate irreversibility, we can set the P/K_{eq} term in the reversible Michaelis-Menten rate law (2.4) to zero. This yields:

$$v = \frac{V_m S}{S + K_m\left(1 + \frac{P}{K_p}\right)} \tag{2.5}$$

2.3 Haldane Relationship

For the reversible enzyme kinetic law there is an important relationship:

$$K_{eq} = \frac{P_{eq}}{S_{eq}} = \frac{V_f\ K_P}{V_r\ K_S} \tag{2.6}$$

This equation shows that the four kinetic constants, V_f, V_r, K_P and K_S are not independent. Haldane relationships can be used to eliminate one of the kinetic constants by substituting the equilibrium constant in its place. This is useful because equilibrium constants tend to be known compared to kinetic constants which may be unknown. By incorporating the Haldane relationship, we can eliminate the reverse maximal velocity (V_r) from 2.4 to yield the equation:

$$v = \frac{V_f/K_S(S - P/K_{eq})}{1 + S/K_S + P/K_P} \tag{2.7}$$

Separating out the terms makes it easier to see that the above equation can be partitioned into a number of distinct parts:

$$v = V_f\ \cdot\ (1 - \Gamma/K_{eq})\ \cdot\ \frac{S/K_s}{1 + S/K_S + P/K_P} \tag{2.8}$$

where $\Gamma = P/S$. The first term, V_f, is the maximal velocity; the second term, $(1 - \Gamma/K_{eq})$, indicates the direction of the reaction according to thermodynamic considerations. The last term refers to the fractional saturation with respect to substrate. Thus we have a maximal velocity, a thermodynamic and a saturation term.

2.4 Competitive Inhibition

There are many molecules capable of slowing down or speeding up the rate of enzyme catalyzed reactions. Such molecules are called enzyme inhibitors and activators. One common type of inhibition, called **competitive inhibition**, occurs when the inhibitor is structurally similar to the substrate so that it competes for the active site by forming a dead-end complex.

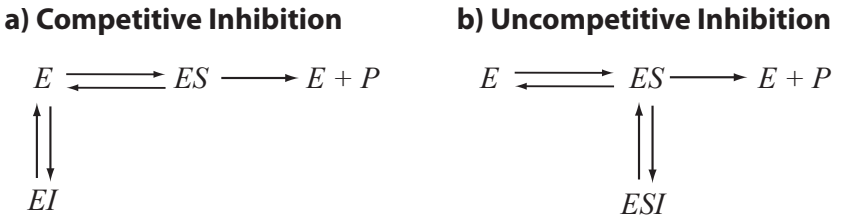


Figure 2.2 Competitive and Uncompetitive Inhibition. P is the concentration of product, E is the free enzyme, ES the enzyme-substrate complex, and ESI the enzyme-substrate-inhibitor complex.

The kinetic mechanism for a pure competitive inhibitor is shown in Figure 2.2(a), where I is the inhibitor and EI the enzyme inhibitor complex. If the substrate concentration is increased, it is possible for the substrate to eventually out compete the inhibitor. For this reason the inhibitor alters the enzyme's apparent K_m, but not the V_{m}.

$$\begin{aligned} v &= \frac{V_m\, S}{S + K_m\left(1 + \dfrac{I}{K_i}\right)} \\ &= \frac{V_m\, S/K_m}{1 + S/K_m + I/K_i} \end{aligned} \tag{2.9}$$

At $I = 0$, the competitive inhibition equation reduces to the normal irreversible Michaelis-Menten equation. Note that the term $K_m(1+I/K_i)$ in the first equation more clearly shows the impact of the inhibitor, I, on the K_m. The inhibitor has no effect on the V_m.

A reversible form of the competitive rate law can also be derived:

$$v = \frac{\dfrac{V_m}{K_s}\left(S - \dfrac{P}{K_{\mathrm{eq}}}\right)}{1 + \dfrac{S}{K_s} + \dfrac{P}{K_p} + \dfrac{I}{K_i}} \tag{2.10}$$

where V_m is the forward maximal velocity, and K_s and K_p are the substrate and product half saturation constants.

Product Inhibition

Sometimes reactions appear irreversible, where no discernable reverse rate is detected, and yet the forward reaction is influenced by the accumulation of product. This effect is caused by the product competing with substrate for binding to the active site and is often called **product inhibition**. Given that product inhibition is a type of competitive inhibition, we will briefly discuss it. An important industrial example of this is the conversion of lactose to galactose by the enzyme $\beta-$galactosidase where galactose competes with lactose, slowing the forward rate [**?**].

To describe simple product inhibition with rate irreversibility, we can set the P/K_{eq} term in the reversible Michaelis-Menten rate law (2.4) to zero. This yields:

$$v = \frac{V_m S}{S + K_m \left(1 + \frac{P}{K_p}\right)} \tag{2.11}$$

It is not surprising to discover that equation (2.11) has exactly the same form as the equation for competitive inhibition (2.9). As the product increases, it out competes the substrate and therefore slows down the reaction rate.

We can also derive the equation by using the following mechanism and the rapid-equilibrium assumption:

$$E + S \rightleftharpoons ES \longrightarrow EP \rightleftharpoons E + P \tag{2.12}$$

where the reaction rate v is assumed to be proportional to ES.

2.5 Cooperativity

Many proteins are known to be oligomeric, meaning they are composed of more than one identical protein subunit where each subunit has one or more binding sites. Often the individual subunits are identical.

If the binding of a ligand (a small molecule that binds to a larger macromolecule) to one site alters the affinity at other sites on the same oligomer, it is called **cooperativity**. If ligand binding increases the affinity of subsequent binding events, it is termed **positive cooperativity** whereas if the affinity decreases, it is termed **negative cooperativity**. One characteristic of positive cooperativity is that it results in a sigmoidal response instead of the usual hyperbolic response.

The simplest equation that displays sigmoid like behavior is the Hill equation:

$$v = \frac{Vm\ S^n}{K_d + S^n} \tag{2.13}$$

One striking feature of many oligomeric proteins is the way individual monomers are physically arranged. Often one will find at least one axis of symmetry. The individual protein

monomers are not arranged in a haphazard fashion. This level of symmetry may imply that the gradual change in the binding constants as ligands bind, as suggested by the Adair model, might be physically implausible. Instead, one might envision transitions to an alternative binding state that occurs within the entire oligomer complex. This model was originally suggested by Monod, Wyman and Changeux [**?**], abbreviated as the MWC model. The original authors laid out the following criteria for the MWC model:

1. The protein is an oligomer.
2. Oligomers can exist in two states: R (relaxed) and T (tense). In each state, symmetry is preserved and all subunits must be in the same state for a given R or T state.
3. The R state has a higher ligand affinity than the T state.
4. The T state predominates in the absence of ligand S.
5. The ligand binding microscopic association constants are all identical. This is in complete contrast to the Adair model.

Given these criteria, the MWC model assumes that an oligomeric enzyme may exist in two conformations, designated T (tensed, square) and R (relaxed, circle). The equilibrium between the two states has an equilibrium constant $L = T/R$, which is also called the **allosteric constant**. If the binding constants of ligand to the two states are different, the distribution of the R and T forms can be displaced towards either one form or the other. By this mechanism, the enzyme displays sigmoid behavior. A minimal example of this model is shown in Figure 2.3.

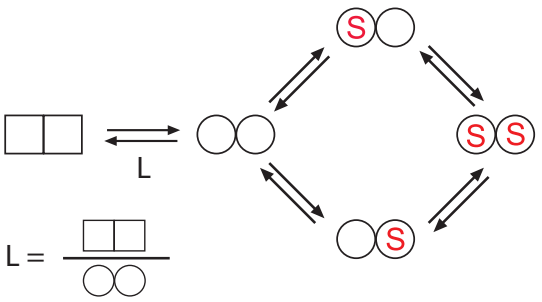


Figure 2.3 A minimal MWC model, also known as the exclusive model, showing alternative microscopic states in the circle (relaxed) form. L is called the allosteric constant. The square form is called the tense state.

In the **exclusive model** (Figure 2.3) the ligand can only bind to the relaxed form (circle). The mechanism that generates sigmoidicity in this model works as follows. When ligand binds to the relaxed form, it displaces the equilibrium from the tense form to the relaxed form. In doing so, additional ligand binding sites become available. Thus, one ligand

binding may generate four or more new binding sites. Eventually there are no more tense states remaining, at which point the system is saturated with ligand. The overall binding curve will therefore be sigmoidal and will show positive cooperativity. Given the nature of this model, it is not possible to generate negative cooperativity. By assuming equilibrium between the various states, it is possible to derive an aggregate equation for the dimer case of the exclusive MWC model:

$$v = V_m \frac{\frac{S}{k_R}\left(1+\frac{S}{k_R}\right)}{\left(1+\frac{S}{k_R}\right)^2 + L}$$

This also generalizes to n subunits as follows:

$$Y = \frac{\frac{S}{k_R}\left(1+\frac{S}{k_R}\right)^{n-1}}{\left(1+\frac{S}{k_R}\right)^n + L} \tag{2.14}$$

For more generalized reversible rate laws that exhibit sigmoid behavior, the reversible Hill equation is a good option. Invoking the rapid-equilibrium assumption, we can form a reversible rate law that shows cooperativity:

$$v = \frac{V_f \alpha \left(1-\rho\right)\left(\alpha+\pi\right)}{1+\left(\alpha+\pi\right)^2}$$

where $\rho = \Gamma / K_{eq}$ and α and π are the ratio of reactant and product to their respective equilibrium constant, α / K_S and π / K_P. For an enzyme with h (using the author's original notation) binding sites, the general form of the reversible Hill equation is given by:

$$v = \frac{V_f \alpha \ \left(1-\rho\right) \ \left(\alpha+\pi\right)^{h-1}}{1+\left(\alpha+\pi\right)^h} \tag{2.15}$$

2.6 Allostery

An allosteric effect is where the activity of an enzyme or other protein is affected by the binding of an effector molecule at a site on the protein's surface, other than the active site. The MWC model described previously can be easily modified to accommodate allosteric action.

The key to including allosteric effectors is to influence the equilibrium between the tense (T) and relaxed (R) states (See Figure 2.4). To influence the sigmoid curve, an allosteric

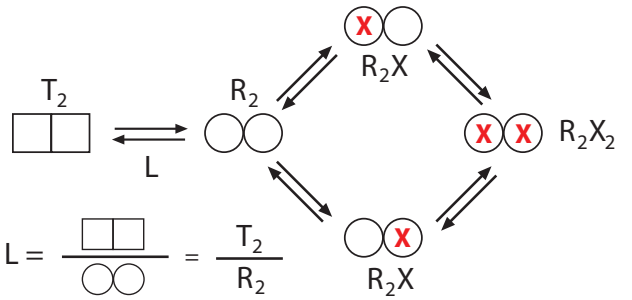


Figure 2.4 Exclusive MWC model based on a dimer showing alternative microscopic states in the form of T and R states. The model is exclusive because the ligand, X, only binds to the R form.

effector need only displace the equilibrium between the tense and relaxed forms. For example, to behave as an activator, an allosteric effector needs to preferentially bind to the R form and shift the equilibrium away from the less active T form. An allosteric inhibitor would do the opposite, that is bind preferentially to the T form so that the equilibrium shifts towards the less active T form. In both cases the V_m of the enzyme is unaffected.

The net result of this is to modify the normal MWC aggregate rate law to the following if the effector is an inhibitor:

$$v = V_m \frac{\alpha\,(1+\alpha)^{n-1}}{(1+\alpha)^n + L(1+\beta)^n} \tag{2.16}$$

where $\alpha = S/K_s$, $\beta = I/K_I$, and K_s and K_I are kinetic constants related to each ligand. A MWC model that is regulated by an inhibitor or an activator is described by the equation:

$$v = V_m \frac{\alpha\,(1+\alpha)^{n-1}}{(1+\alpha)^n + L\dfrac{(1+\beta)^n}{(1+\gamma)^n}}$$

There are also reversible forms of the allosteric MWC model but they are fairly complex. Instead, it is possible to modify the reversible Hill rate law to include allosteric ligands.

$$v = \frac{V_f \alpha \left(1 - \dfrac{\Gamma}{K_{eq}}\right)(\alpha+\pi)^{h-1}}{\dfrac{1+\mu^h}{1+\sigma\mu^h} + (\alpha+\pi)^h} \tag{2.17}$$

where:

$$\begin{aligned} \sigma &< 1 \quad \text{inhibitor} \\ \sigma &> 1 \quad \text{activator} \end{aligned}$$

Simple Hill Equations

When modeling gene regulatory networks, we often need simple activation and repression rate laws. It is common to use the following Hill like equations to model activation and repression, respectively. The third equation shows one example of how we can model dual repression and activation, where S_1 acts as the activator and S_2 the inhibitor. n_1 and n_2 are Hill like coefficients which may be used to alter the responsiveness of each factor.

$$\text{Activation: } v = \frac{V_m S^n}{K + S^n}$$

$$\text{Repression: } v = \frac{V_m}{K + S^n}$$

$$\text{Dual: } v = \frac{V_m S_1^{n_1}}{1 + K_1 S_1^{n_1} + K_2 S_2^{n_2} + K_3 S_1^{n_1} S_2^{n_2}}$$

2.7 Elasticities

Elasticities measure the response of a chemical reaction rate to changes in the immediate environment. For example, given a simple reaction such as:

$$S \rightarrow P$$

we can measure two elasticities, one with respect to S and the other with respect to P. Each elasticity gives us the response of the reaction rate when either S or P are changed, respectively. Mathematically, the elasticity is defined in terms of a scaled derivative:

$$\varepsilon_S^v = \frac{\partial v}{\partial S}\frac{S}{v} \simeq \frac{v\%}{S\%} \tag{2.18}$$

According to the definition, one can interpret an elasticity as a ratio of relative changes. Even though the elasticity is only defined for infinitesimal changes, we can approximate the elasticity in terms of small finite changes and conveniently interpret it as the ratio of percentage changes. For example, if we were to make a 2% change in S, and in turn observed a 0.5% change in the reaction velocity, then the value of the elasticity is given approximately by the ratio $0.5/2 = 0.25$. Full details of the elasticity and its properties can be found in the companion book Enzyme Kinetics for Systems Biology.

Unscaled Elasticity

We can also define the unscaled elasticity as:

$$\mathcal{E}_S^v = \frac{\partial v}{\partial S} \tag{2.19}$$

Deriving Elasticities

Elasticities can be computed numerically or derived analytically. For example, given the rate law:

$$v = kA$$

We can derive the elasticity with respect to A using the definition given by (2.18). This involves first differentiating the expression with respect to A and then scaling the derivative using A and v. In this case:

$$\frac{dv}{dA} = k$$

Scaling yields:

$$\frac{\partial v}{\partial A}\frac{A}{v} = kA/v = kA/(kA) = 1$$

The elasticity for a first order reaction is one. This means that a 1% change in A will yield a 1% change in v.

Further Reading

1. Sauro HM (2012) Enzyme Kinetics for Systems Biology. 2nd Edition, Ambrosius Publishing ISBN: 978-0982477335

2.8 Exercises

1. State one assumption used in deriving the Briggs-Haldane equation.

2. Show that the K_m is the concentration of substrate when the reaction rate is half $V_{\max}$.

3. What is the difference between a reaction that is reversible and one that shows product inhibition?

4. Write down the rate law for competitive inhibition. In competitive inhibition, the apparent K_m is changed but not the $V_{\max}$, justify this statement.

5. Define the term positive cooperativity

6. Define the elasticity coefficient

6. Derive the elasticity coefficients with respect to species A for the following rate laws:

a) $v = kA$

b) $v = kA^2$

c) $v = k_1A - k_2B$

d) $v = Af(k)$

3

Quick Introduction to Python

In this book we will be using Python and the assorted Python packages to model the dynamics of cellular networks. We will briefly covers some of the more important aspects of the Python programming language but will not attempt to teach Python to any great depth. For that the reader is directed to the many commercial and free texts that are available for those who want a more in depth description of how to program using Python. Of particular recommendation is the Python tutorial on the Python web site itself (https://docs.python.org/2/tutorial/), the New Mexico Tech Programming Tutorial (http://infohost.nmt.edu/tcc/help/pubs/lang/pytut27/pytut27.pdf), the codeacademy online course (http://www.codecademy.com/en/tracks/python), and A Byte of Python at http://www.swaroopch.com/notes/python/.

To get started let's define some terms that will be frequency used in the text. First and foremost, what is Python?

Python Python is an easy to learn general purpose interactive programming language. It has similar usability characteristics to Matlab or Basic. As such it is a good language to use for doing pathway simulations and is easily learned by new users. In recent years Python has also become more widely used as a general purpose scientific programming language and now supports many useful libraries and tools for scientists and engineers. All the scripts we provide in this book are written in Python.

SBML The Systems Biology Markup Language (SBML) is the de facto standard for exchanging models of biological pathways. SBML uses XML to represent pathway models and is used to communicate models between different software applications. Since the introduction of SBML there now exist model repositories such as Biomod-

els (https://www.ebi.ac.uk/biomodels-main/)(which contain large numbers of published models that can be downloaded and simulated. Details of SBML and its capabilities can be found at the main SBML web site, sbml.org. For the purpose of building and running models it is not necessary to know the details of how models are stored in SBML. It is sufficient to be able to load and save SBML files.

Antimony SBML has become a de facto standard for exchanging models of biological pathways. Any tool we use should therefore be able to support SBML. However SBML is a computer readable language and it is not easy for humans to read or write SBML. Instead more human readable formats have been developed. In this text book we will be using the Antimony pathway description language. Models can be described in Antimony then converted to SBML or vice versa.

libRoadRunner To support SBML from within Python we developed a C/C++ simulation library called libRoadRunner that can read and run models based on SBML. In order to use libRoadRunner within Python, we also provide a Python interface that makes it easy to carry out simulations with Python.

Spyder Integration of the various tools including Python is achieved by using spyder2 (https://code.google.com/p/spyderlib/). Spyder2 offers a MATLAB like experience in a friendly, cross-platform environment.

Matlplotlib Matplotlib is the standard plotting library for python.

Numpy Numpy is the standard library for for creating and manipulating data arrays.

3.1 Introduction to Python

One great advantage of the Python language is that is runs on many different kinds of computers, most notably Windows, Mac and Linux but also small and cheap computers such as the Raspberry Pi (http://www.raspberrypi.org/). All the simulations we describe in this book can be run on a $35 Raspberry Pi. A basic Python setup can be obtained from the python web site python.org. However to make it easier we can also obtained Python IDE (Integrated Development Environment). In the Python world there are many IDEs to choose from, ranging from very simple consoles to sophisticated development systems that includes documentation, debuggers and other visual aids. In this book we use the cross-platform IDE called spyder2 (https://code.google.com/p/spyderlib/).

The best way to learn Python is to download a copy and start using it. We have prepared installers that install all the relevant components you need, these can be found at tellurium.analogmachine.org. The Tellurium distribution includes some additional helper routines which can make life easier for new users. We have version for the Mac and Windows. We will use the Windows version here. To download the installer go to the web site tellurium.analogmachine.org, and click on the first link you see called

Download Windows version here. Run the installer and follow the instructions. Note that the Tellurium installer will *not* interfere with any existing Python installations you might already have.

Once Tellurium is installed go to the start menu, find `Tellurium WinPython` and select the application called `Spyder for Tellurium`. If successful you should see something like the screen shot in Figure 3.1 but without the plotting window. The screen-shot shows three important elements, on the left we see an editor, this is where models and Python code can be edited. On the lower right is the Python console where Python commands can be entered. At the top right we show a plotting window that illustrates some output from a simulation. For those familiar with IPython, the latest version of spyder2 supports the IPython console directly.

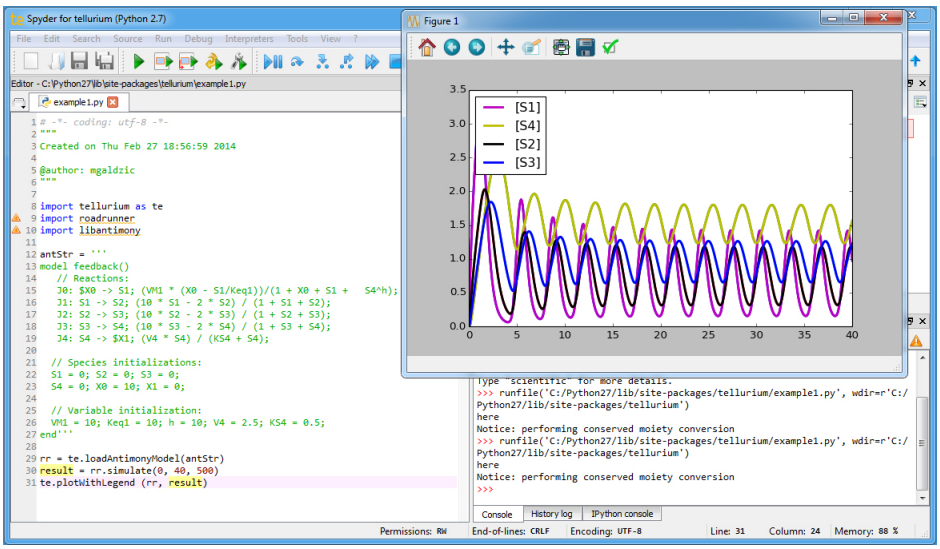


Figure 3.1 Screen-shot of Tellurium, showing editor on the left, Python console bottom right and plotting window top-right.

3.1.1 Running Commands from the Console

Once you have started the Tellurium IDE, let us focus on the Python console at the bottom right of the application. A screen-shot of the console is shown in Figure 3.2.

The >>> symbol marks the place where you can type commands. The following examples are based on Python 2.7. To add two numbers, say 2 + 5, we would type the following:

```
>>> print 2 + 5
7
>>>
```

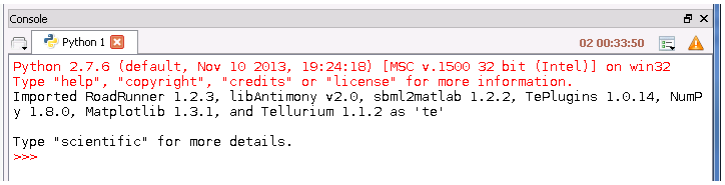


Figure 3.2 Screen-shot of Tellurium, focusing on the Python console.

Listing 3.1 Simple Arithmetic

Just like Matlab or Basic we can assign values to variables and use those variables in other calculations:

```
>>> a = 2
>>> b = 5
>>> c = a + b
>>> print c
7
>>>
```

Listing 3.2 Assigning values to variables

The types of values we can assign to variables include values such as integers, floating point numbers, Booleans (True or False), strings and complex numbers.

```
>>> a = 2
>>> b = 3.1415
>>> c = False
>>> d = "Hello Python"
>>> e = 3 + 6j
>>>
```

Listing 3.3 Different kinds of values: integer, floating point, Boolean, string and a complex number

Many functions in Python are accessible via modules. For example to compute the sin of a number we can't simply type `sin (3.1415)`. Instead we must first load the math module. We can then call the sin function:

```
>>> import math
>>> print sin (3.1415)
9.265358966049026e-05
>>>
```

Listing 3.4 Importing modules (libraries) into Python

In Tellurium we preload some libraries including the math library.

3.1.2 Repeating Calculations

One of the commonest operations we do in computer programming is iteration. We can illustrate this with a simple example that loops ten times, each time printing out the loop counter. This example will allow us to introduce the IDE editor. The editor is the panel on the left side of the IDE. In the editor we can type Python code, for example we can type:

```
a = 4.0
b = 8.0
c = a/b
print "The answer is:", c
```

Listing 3.5 Writing a simple program in the IDE editor

When we've finished typing this in the editor window, we can save our little program to a file (Select Menu: File/Save As...) and run the program by clicking on the green arrow in the tool bar of the IDE (Figure 3.3). If we run this program we will see:

```
The answer is: 0.5
>>>
```

Listing 3.6 Writing a simple program in the IDE editor

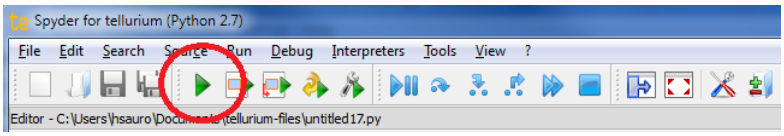


Figure 3.3 Screen-shot of Tellurium, focusing on the Toolbar with the run button circled.

The IDE allows a user to have as many program files open at once, each program file is given its own tab so that it is easy to move from program to the next. This is useful if we are working on multiple models or programs at the same time.

We will now use the editor to write the simple program that loops ten times, this is shown below:

```
for i in range (10):
    print i,
```

Listing 3.7 A simple loop in python

This will generate the sequence:

```
0 1 2 3 4 5 6 7 8 9
```

Listing 3.8 Result from simple loop program

There are a number of new concepts introduced in this small looping program. The first line contains the `for` keyword that can be translated into literal English as "for all elements in a list, do this". The list is generated by the `range()` function and in this case generates a list of 10 numbers starting at 0. `i` is the loop index and within the loop, `i` can be used in other calculations. In this case we just print the value of `i` to the console. Each time the program loops it extracts the next value from the list and assigns it to `i`.

Two things are important to note in the print line. The first and most important is that the line has been indented four spaces. This isn't just for aesthetic reasons but is actually functional. It tells Python what code should be executed *within* the loop. To elaborate we could add more lines to the loop, such as:

```
for i in range (10):
    a = i
    b = a*2
    print b,
print "Finished Loop"
```

Listing 3.9 A simple loop illustrating multiple statements

In this example there are three indented lines, this means that these three lines will be executed within the loop. The last line which prints a message, is not indented and therefore will not be executed within the loop. This means we only see the message appear once right at the end. The output for this little program is shown below.

```
0 2 4 6 8 10 12 14 16 18 Finished Loop
```

Another important point worth noting is the use of the , after the loop print statement. The comma is used to suppress a newline. This is why the output appears on one line only. If we had left out the comma each print statement would be on its own line.

A final word about `range()`, range takes up to three arguments. In the example we only gave one argument, 10. A single argument means create a list starting at zero, incrementing one for each item until 10 items have been created. Adding a second argument such as in `range (5, 10)` means start the list at 5 rather than zero. Finally, a third argument can be

used to specify the increment size. For example the command `range (1, 10, 2)` will yield the list:

```
[1, 3, 5, 7, 9]
```

The easiest way to try out the various options in range is to type them at the console to get immediate feedback.

The use of variables, printing results, importing libraries and looping are probably the minimum concepts one needs to start using Python. However there are a huge range of resources online to help learn Python. Of particular interest is the codecademy web site (http://www.codecademy.com/). This site offers an interactive means to learn Python (including other programming languages).

3.1.3 Making Decisions: Conditionals

As well are repeating calculations, another very common operation when writing program code is making decisions based on a previously computed value. A typical example is given below:

```
a = 3
if a > 2:
   print "a is greater than 2"
else:
   print "a is smaller than 2"
```

To make decisions in Python we use the key word `if` which is followed by the test we want to make. If the test is true the following indented code is executed. If the code has the optional `else` keyword, then if the test is false the else code is executed. The code fragment, `a > 2` is called a Boolean expression because it resolves to either `True` or `False`. There are a number of so-called Boolean operations that can be used in Boolean expressions. Table 3.1 illustrates some of common Boolean operators used in Python.

The operators `>`, `<`, `>=` and `<=` should be self-explanatory but a couple of the other operators are worth describing in more detail. The operator `==` may look odd but all it does is test if two things are the same, for example `1 == 2` will resolve to false whereas `5 == 5` will resolve to true. Likewise `!=` tests whether two things are **not** equal to each other, for example `"ATP" != "ATP"` will return false where as `"Glucose" == "Fructose"` will return true.

3.1.4 Creating Functions in Python

The other important concept to learn in Python is the notion of functions. We've already used the `sin` function in a previous section. Python comes with many hundreds of functions

Boolean Operator	Description
`>`	Greater Than
`<`	Less Than
`>=`	Greater than or equals
`<=`	Less than or equals
`==`	equals
`!=`	not equal to
`or`	true if either of two things are true
`and`	true if two things are simultaneously true
`not`	negation, `True` and `not False` are equivalent statements

Table 3.1 Selection of functions from the math package

like this. Table 3.2 lists a few of the functions that can be found in the math package.

Math Function	Description
`sin`	Compute Sine of an angle
`cos`	Compute Cosine of an angle
`tan`	Compute Tangent of an angle
`fabs`	Compute the absolute value of a number
`log10`	Compute logarithm to base 10 of a number
`sqrt`	Compute square root of a number

Table 3.2 Selection of function from the math package

We can also create our own functions. Often when writing a program, we may find that some operations are repeated often. In such cases it is convenient and good design practice to turn these sections of code into a function. Let's say we wrote a program that repeatedly needed to compute the area of a square, that is multiply the width and height. We could define a function called `area` that accepts the width and height as arguments and returns the area. The code below show how we would write this function:

```
def area (width, height):
   return width*height
```

Listing 3.10 Defining a function in Python

This is a fairly simple function but one could imagine much more complicated functions. For example consider writing a function to tell us whether a given number is prime or not. A prime number is a number that is only divisible by 1 or itself, therefore, 7 is a prime but 9 is not.

The code code below, Listing 3.3.5 defines a function `isPrime` that can be used to decide

whether a given number is a prime number or not. The function returns `True` is the argument to the function is a prime, otherwise it returns `False`. The first part of the function checks if the number coming into the function, `n` is zero or one, both of these numbers are by definition not prime numbers. Next we check if the number is even because all even number can not be prime since they are divisible by 2. Finally we start a loop checking to see if the number is divisible without a reminder by any number greater than 2. One final thing that this code introduces are comments. Comments are notes we add to the code to remind ourself of why we the code code the way we did. Comments are started with the hash symbol and the a text description. Note that comments are not executed.

One piece of code that has not been described is the `%` symbol. This computes the reminder of a number. For example `5 % 2` will yield 1 because the reminder after dividing 5 by 2 is one. A code fragment such as `5 % i == 0` is an easy way to test whether is number can be divided by `i` without a reminder.

```
def isPrime(n):
    # No negative numbers
    if n < 0:
       return False

    # zero and one are by definition not prime numbers
    if n == 0 or n == 1:
       return False

    # Anything divisible by 2 is not a prime
    if n % 2 == 0:
       return False

    for i in range(3, n):
        if n % i == 0:
            return False
    return True
```

Listing 3.11 Simple function to decide whether a number is prime or not

Comments are notes added to a program. Comments are started with a # symbol.

```
# This is a comment
```

3.2 Brief Introduction to Numpy

Numpy is an extremely useful package that supports arrays as well as other mathematical support in Python. To use Numpy first import the library as follows:

```
import numpy as np
```

3.2.1 Creating Arrays

There are various ways to create Numpy arrays, examples include:

```
# Create a simple 3 by 3 array of values
m = np.array([[1.5, 6.7, 3.4], [7.4, 5.6, 1.1], [0.7, 23.5, 3.4]])

# Create a populated vector, arguments are
# starting value, ending value, number of elements
>>> np.linspace (1, 10, 10)

# Create a 2 by 3 matrix with zero elements
>>> np.zeros((2, 3))
array([[ 0., 0., 0.], [ 0., 0., 0.]])
```

3.2.2 Indexing Elements

Unlike many scientific languages, Python indexes array from *zero*. This give the array: $m = np.array([[1.5, 6.7], [0.7, 23.5]])$, index 0, 0 is the element at the top/left row/column.

```
>>> m = np.array([[1.5, 6.7], [0.7, 23.5]])
>>> print m[0,0]
array([1.5])
```

3.2.3 Extracting Rows and Columns

Extracting a column from an array is very straight forward, use a single index that corresponds to the row of interest.

```
>>> m = np.array([[1.5, 6.7], [0.7, 23.5]])
>>> print m[0]
array([1.5, 6.7])
```

To extract a column we can use the slicing syntax. This is extended from the normal python slicing syntax. A numpy slice takes the form: array [from, to]. That is it pulls out a section of the array starting and from and ending at to. The from and to entries have the following syntax, start:end where start and end are numbers corresponding to array indices. Note that the end value is end-1 indexth. Some examples will make this clear

```
>>> m = np.array([[1.5, 6.7], [0.7, 23.5]])
>>> print m[0:2,0:2]  # Means print out the entire matrix
array([[1.5, 6.7], [0.7, 23.5]])
```

With this syntax it is straight forward to extract subarrays from a larger array. However when make the slicing more useful is that most the arguments are optional. For example missing out the start and end value in the from argument and leaving just the colon, means all rows. Likewise with the to argument, a single colon here means all columns. Knowing this, the entire array can be expressed using `m[:,:]`. What is more useful is when we omit the arguments selectively. For example, `m[:,1]` means extract column index 1. Or `m[2,:]` means extract the 3rd row.

3.2.4 Stacking Arrays

A very useful technique is being able to stack arrays either one above the other or concatenating them side by side. In simulation this can be useful when multiple simulations are carried out and a single array constructed that contains all the simulations. To stack one array on top of the other use the numpy method `vstack`:

```
>>> m1 = np.array([[1, 6], [2, 5]])
>>> m2 = np.array([[3, 9], [4, 7]])
>>> m3 = np.vstack ((m1, m2))
>>> print m3
array([[1, 6],
       [2, 5],
       [3, 9],
       [4, 7]])
```

To stack arrays horizontally, use the `hstack` method:

```
>>> m1 = np.array([[1, 6], [2, 5]])
>>> m2 = np.array([[3, 9], [4, 7]])
>>> m3 = np.hstack ((m1, m2))
>>> print m3
array([[1, 6, 3, 9],
       [2, 5, 4, 7]])
```

For both `vstack` and `hstack` to work the dimensions of the arrays to be stacked must be compatible. For example two arrays that are to be stacked horizontally must have the same number of rows.

The final thing to node is that syntax for the stack methods use a tuple to indicate what arrays to stack. A tuple is a simple list of values, very much list a python list but more efficient. The advantage of using a tuple is that we can use the stacking methods to stack

as many arrays as we like in one statement, for example:

```
>>> m1 = np.array([[1, 6], [2, 5]])
>>> m2 = np.array([[3, 9], [4, 7]])
>>> m3 = np.array([[5, 3], [6, 2]])
>>> m4 = np.hstack ((m1, m2, m3))
>>> print m4
array([[1, 6, 3, 9, 5, 3],
       [2, 5, 4, 7, 6, 2]])
```

3.3 Plotting Graphs

One of the most important abilities to learn is to plot data. The library most commonly used in python to plot data is matplotlib combined with the array handling library Numpy. Let's start by importing matplotlib and numpy into python:

```
>>> import numpy as np
>>> import pylab as pl
```

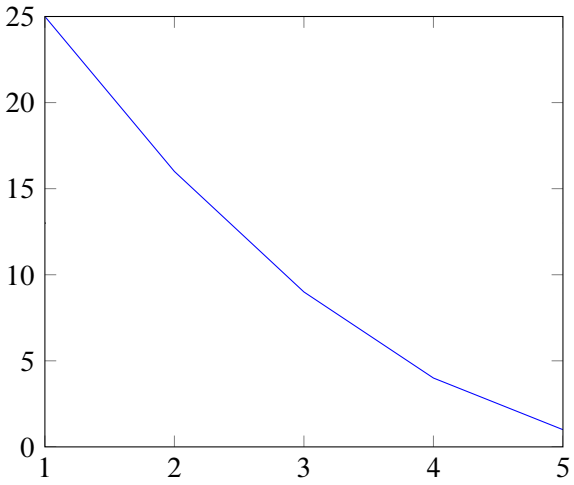
3.3.1 Basic Plotting

Let's start by examining how to do line plots. The primary plotting command is `plot`. This command can take many different arguments but the most common will by x and y data. For example:

```
# Make two arrays for x and y data
x = [1,2,3,4,5]
y = [25,16,9,4,1]

# plot the data
pl.plot (x,y)

# show the plot
pl.show()
```



3.3.2 Changing Markers and Line Styles

To mark the individual points in a plot we include additional arguments in the plot command that indicates the type of marker we'd like to use. In the following example we use `marker='o'` as the third argument (use round markers) and `linestyle=''` as the fourth (don't plot a line).

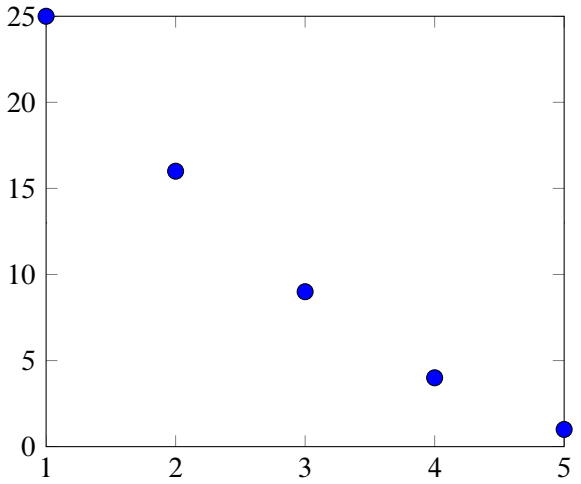
```
# Make two numpy arrays for x and y data
x = np.array ([1,2,3,4,5])
y = np.array ([25,16,9,4,1])

# plot the data as scatter plot
pl.plot (x,y, marker='o', linestyle='')

# show the plot
pl.show()
```

'o'	Round		
's'	Square		
'p'	Pentagon	''	No lines
'*'	Star	'-'	Continuous line
'h'	Hexagon	'-.'	Dash-dot line
'+'	Plus	':'	dotted line
'x'	x marker		
'D'	Diamond		

Table 3.3 Matplotlib Markers and Line Styles



There is a wide variety of markers to chose from, some are listed in Table 3.3. L:Ikewise there are a variety of line styles shown in Table **??**

3.3.3 Changing Colors

The color of markers and lines can be changed using the color argument. One way to do this is to call plot twice once to add the markers and a second time to add the lines. In matplotlib, each call to `plot` will add an additional plot to the existing graph. The show command releases the graph so that subsequent call to plot will add the plots to a new graph.

```
# Make two numpy arrays for x and y data
x = np.array ([1,2,3,4,5])
y = np.array ([25,16,9,4,1])

# plot the data as a line and scatter plot
pl.plot (x,y, marker='o', linestyle='')
```

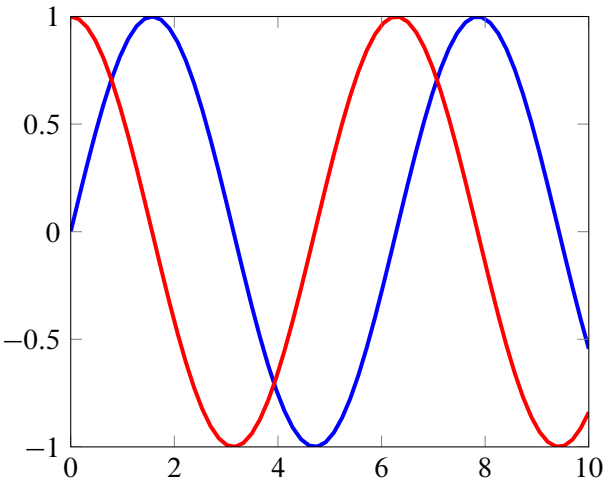


Figure 3.4 Plotting Sine and Cosine Curves

```
pl.plot (x,y, color='g')

# show the plot
pl.show()
```

3.3.4 Plotting Functions

What if we wanted to plot the sine and cosine curves? TO do this we would first compute arrays that contain sine and cosine curves, the plot each one.

```
# Make two arrays for x and y data
x = np.linspace (0, 10, 64)
# Use the numpy sin/cos functions which can use vectors as arguments
s = np.sin (x)
c = np.cos (x)

# plot the data as a line and scatter plot
pl.plot (x, s, color='b', linewidth=2.5)
pl.plot (x, c, color='r', linewidth=2.5)

# show the plot
pl.show()
```

3.3.5 Arithmetic using Numpy

One of the more useful features of the numpy library is that is allows one to use the normal arithmetic operations, such as addition and multiplication to be applied to arrays of data. For example suppose we had a vector `1,2,3,4` and we wanted to added 4 top every element. One way would be to set up a loop and ad 4 to each element. A much more convenient way is to use the numpy vector arithmetic. To add 4 to every element of the array we wouldsimply add 4 to the array, as follows:

```
>>> x = np.array([1,2,3,4])
>>> y = x + 4
>>> print x
array([5, 6, 7, 8])
```

Note that it is very important that the array store in `x` is a numpy array. If we did `x = [1,2,3,4]`, `x` would be a list and we couldn't do the arithmetic. Given this ability we can use this to easily create arrays of data created from arbitrary functions. For example:

```
def myfunc (x):
    return np.sin(x*1.2) + np.cos(x*2.3)

>>> x = np.linspace (0, 40, 200)
>>> y = myfunc (x)
>>> pl.plot (x,y)
>>> pl.show()
```

3.4 Exercises

1. What is Python?
2. What does the acronym SBML stand for?
3. What is SBML?
4. What is antimony?
5. What is libRoadRunner?
6. What is numpy?
7. What is matplotlib?
8. Plot the irreversible Michaelis-Menten equation as a function of substrate concentration.

   Use a $K_m = 0.5$ and $V_{\max} = 5$.

9. Plot a range of Michaelis-Menten curves using K_m values 0.5, 1.5, 2.5, 3.5, 4.5
10. Show that at the K_m value, the reaction rate is half the $V_{\max}$.
11. Plot the irreversible Hill equation as a function of substrate concentration.

    Hint: Be aware that the power operator, `^` does not work with vectors, You must instead use the `np.power (x,n)` function.

    Use a $K_m = 0.5$, $V_{\max} = 5$ and $n = 4$.
12. Plot a range of Hill equation curves using n values 0, 1, 2, 4, 8, 16
13. Explore how changes in K_m, $V_{\max}$ and n affect the shape of the reaction response.
14. Derive the elasticity for the Brigg-Haldane enzyme rate law and plot the elasticity as a function of substrate level.
15. Plot the elasticity for the Hill equation.
16. What differences do you observe between the elasticity response of the Briggs-Haldane and Hill equation?

4

Networks in a Nutshell

4.1 Mass Conservation

In the first chapter we saw how the rate of changes of a species was a function of the reaction rate and the stoichiometric coefficient, that is:

$$\frac{dA}{dt} = c_a v \tag{4.1}$$

What would happen if a species, A, was involved in two reactions, one reaction producing A and another reaction consuming A:

$$\xrightarrow{v_1} A \xrightarrow{v_2}$$

where v_1 and v_2 are the production and consumption rates respectively. How would we write out the rate of change of A? We can invoke mass conservation to help us. In particular, any change in A must be due to difference between A that is produced and A that is consumed, that is: Rate of production of A:

$$\frac{dA_p}{dt} = v_1$$

Rate of consumption of A:

$$\frac{dA_c}{dt} = -v_1$$

Summing the two gives us the overall rate of change of A

$$\frac{dA}{dt} = \frac{dA_p}{dt} + \frac{dA_c}{dt} = v_1 - v_2$$

In general for a species S_i that has many production rates and many consumption rates the overall rate of change is given by:

$$\frac{dS_i}{dt} = \sum_j c_{ij} v_j \tag{4.2}$$

where c_{ij} is the stoichiometric coefficient for species i with respect to reaction, j.

4.2 Exercise

Write out the net rates of change (dS_i/dt) for the following reaction systems. The reaction are given on the left and the reaction rate on the right:

a)

$$S_1 + S_2 \rightarrow S_3 \quad v_1$$

b)

$$\begin{aligned} S_1 &\rightarrow S_2 \quad v_1 \\ S_2 &\rightarrow S_3 \quad v_2 \\ S_3 &\rightarrow S_4 \quad v_3 \end{aligned}$$

c)

$$\begin{aligned} \rightarrow S_1 &\quad v_1 \\ S_1 \rightarrow &\quad v_2 \\ S_1 \rightarrow &\quad v_3 \end{aligned}$$

d)

$$\begin{aligned} S_1 &\rightarrow S_2 \quad v_1 \\ S_2 &\rightarrow S_1 \quad v_2 \end{aligned}$$

e)

$$\begin{aligned} S_1 &\rightarrow 2S_2 \quad v_1 \\ S_2 &\rightarrow S_3 \quad v_2 \\ S_3 &\rightarrow \quad v_3 \\ \rightarrow S_3 &\quad v_4 \end{aligned}$$

f) Make up your own reaction network and write out the net rate of change equations.

Detailed Model	Modeling individual addition of nucleotides and amino acids
Moderate Detail	Model production of mRNA and protein as simple pools
Simple Model	Model production of protein, merge transcription and translation in to one reaction
Promoter Activity	Model either using a Hill equation or explicitly as a binding/unbinding event.

Table 4.1 Different ways to model gene expression

4.3 Modeling Gene Networks

Metabolic and signaling networks can be modelled using a sequence of reactions as shown in the previous section. For example assume a protein A binds protein B to form complex C. Complex C degrades as a result of binding to another protein D. Sequestration of D by protein E to form complex F. This scenario could be modeled using the following set of reactions:

$$A + B \rightarrow C \quad v_1 \qquad \text{Protein binding step}$$

$$C + D \rightarrow D \quad v_2 \qquad \text{Protein degradation catalyzed by } D$$

$$D + E \rightarrow F \quad v_2 \qquad \text{Protein sequestration of } D$$

Given a suitable description of the biological process it shouldn't be difficult to come up with a suitable set of reactions. Gene networks are treated slightly differently however. The modeling of gene expression can be carried out at different levels. The most detailed model might consist of individual reactions that add nucleotides to a growing mRNA strand or the addition of individual amino acids to a grown peptide. We will not consider such detailed models here. Alternatively we can model gene expression using pools for mRNA and protein. Finally we an dispense with the mRNA completely and model gene expression as a single process that produces protein. Additional modifications can be made to the promoter mechanism. The simplest approach is to represent the rate of gene expression using a simple Hill equation that is a function of the transcription factor. Alternatively it is possible to explicitly model the transcription factor binding to the promoter site (Figure 4.1).

A model for simplest case would be:

$$v = \frac{Vm\ T}{K + T}$$

where T is the concentration of transcription factor, V_m the maximal rate of gene expression, K a Michaelis like constant, and v is rate at which protein is made. A more compli-

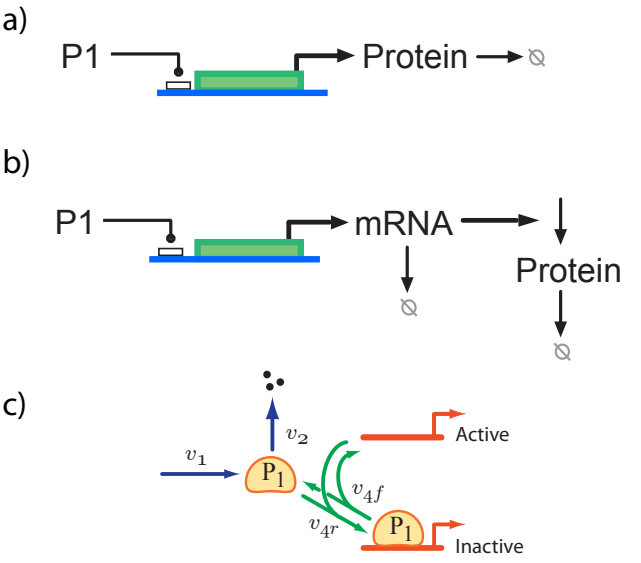


Figure 4.1 Three different ways to model gene expression. a) Simple expression of protein; b) Modeling mRNA production. Note that mRNA acts as a catalyst for protein synthesis. c) Modeling the promoter explicitly.

cated model can be built by explicitly modelling the mRNA pool. This is shown below:

$$\begin{aligned} &\rightarrow \text{mRNA} v_1 \\ \text{mRNA} &\rightarrow \text{degradation products} \\ \text{Protein} &\rightarrow \text{degradation products} \end{aligned}$$

What is missing form this description are the rate laws. The rate of mRNA production can be model using a Hill equation where the rate of mRNA production is a function of transcription factor. The degradation of mRNA can be modelled as a simple first-order mass-action kinetic law, $v = k\text{mRNA}$. The protein synthesis step may require some explanation. In Figure 4.1 we depict protein synthesis as being catalyzed by mRNA. This is reasonably since mRNA is not degraded by the ribosome. There are various possibilities for the protein synthesis rate law. The simplest is a first-order mass-action rate law that is a function of mRNA, $v = k\text{mRNA}$. However we might feel that it is possible to saturate the ribosome with mRNA in which case we might use a Briggs-Haldane rate law: $v = V_m\text{mRNA}/(K + \text{mRNA})$. Finally we must include a protein degradation step which again can be modeled as a first-order mass-action rate law, $v = k\text{Protein}$. The full set of

differential equations for the model shown in Figure 4.1 will be:

$$\frac{d\,\text{mRNA}}{dt} = \text{Hill Equation} - \text{Degradation rate}$$

$$\frac{d\,\text{Protein}}{dt} = \text{Protein Synthesis} - \text{Degradation rate}$$

4.4 Exercises

1. Consider the simple network:

$$\rightarrow S \rightarrow$$

Assume that the first reaction has a constant reaction rate of v_o and the second reaction a first-order rate equal to kS. Answer the following questions:

a) What kinds of mechanisms could you image would allow the first reaction to have a constant rate?

b) Write out the differential equation for S, i.e dS/dt.

c) At steady state, the rate of change of S is zero. Determine the steady state level of S.

2. Write out the differential equation for the model shown in Figure 4.1c. There should be three differential equations, one for P_1, P_1Inactive and one for Active state.

5

Entering Models

Int he last chapter we briefly saw how to write down the differential equations given a network. As fun as that might be, it is error prone. Instead we use automated tools to derive the equations for use. This is the purpose of Antimony.

5.1 Describing Reaction Networks using Antimony

Antimony is a language for describing reaction networks. Such networks can be chemical reaction networks, metabolic networks, protein signaling networks or gene regulatory networks.

The code shown in the panel below illustrates a very simple model using the Antimony syntax followed by two lines of Python that uses libRoadRunner to run a simulation of the model. In this section we will briefly describe the Antimony syntax. A more detailed description of Antimony can be found at http://antimony.sourceforge.net/index.html.

```
import tellurium as te

r = te.loada ('''
  S1 -> S2; k1*S1;
  S1 = 10; k1 = 0.1
''')

r.simulate (0, 50, 100)
r.plot()
```

Listing 5.1 Model expressed in Antimony and simulated using libRoadRunner

The main purpose of Antimony is to make it straight forward to specify complex reaction networks using a familiar chemical reaction notation.

A chemical reaction can be an enzyme catalyzed reaction, a binding reaction, a phosphorylation, a gene expressing a protein or any chemical process that results in the conversion of one of more species (reactants) to a set of one or more other species (products). In Antimony, reactions are described using the notation:

```
A + ...  -> P + ...
```

where the reactants are on the left side and products on the right side. The left and right are separated by the `->` symbol. For example:

```
A -> B
```

describes the conversion of reactant `A` into product `B`. In this case one molecule of `A` is converted to one molecule of `B`. The following example shows non-unity stoichiometry:

```
2 A -> 3 B
```

which means that two molecules of `A` react to form three molecules of `B`. Bimolecular and other combinations can be specified using the `+` symbol, that is:

```
2 A + B -> C + 3 D
```

tells us that two molecules of `A` combine with one molecule of `B` to form one molecule of `C` and three molecules of `D`.

To specify species that do not change in time (boundary species), add a dollar character in front of the name, for example:

```
$A + B -> C
```

means that during a simulation `A` is fixed.

Reactions can be named using the syntax `J1:`, for example:

```
J1:  A + B -> C
```

means the reaction has a name, `J1`. Named reaction are useful if you want to refer to the flux of the reaction; kinetic rate laws come immediately after the reaction specification. If only the stoichiometry matrix is required, it is not necessary to enter a full kinetic law, a simple `... -> S1; v;` is sufficient. Here is an example of a reaction that is governed by a Michaelis-Menten rate law:

```
A -> B; Vm*A/(Km + A);
```

Note the semicolons. Here is a more complex example involving multiple reactions:

```
MainFeed:    $X0 -> S1;  Vm*X0/(Km + X0);
TopBranch:    S1 -> $X1; Vm1*S1/(Km1 + S1);
```

```
    BottomBranch: S1 -> $X2; Vm2*S1/(Km2 + S1);
```

There is no need to pre-declare the species names shown in the reactions or the parameters in the kinetic rate laws. Strictly speaking, declaring the names of the floating species is optional, however this feature is for more advanced users who wish to define the order of rows that will appear in the stoichiometry matrix. For normal use there is no need to pre-declare the species names. To pre-declare parameters and variables see the example below:

```
    const Xo, X1, X2; // Boundary species
    var S1;          // Floating species

    MainFeed:    $X0 -> S1;  Vm*X0/(Km + X0);
    TopBranch:    S1 -> $X1; Vm1*S1/(Km1 + S1);
    BottomBranch: S1 -> $X2; Vm2*S1/(Km2 + S1);
```

We will describe this in more detail in the next section but we can load an Antimony model into libRoadRunner using the short-cut command `loada`. For example:

```
import tellurium as te
rr = te.loada ('''
    const Xo, X1, X2; // Boundary species
    var S1;          // Floating species

    MainFeed:    $X0 -> S1;  Vm*X0/(Km + X0);
    TopBranch:    S1 -> $X1; Vm1*S1/(Km1 + S1);
    BottomBranch: S1 -> $X2; Vm2*S1/(Km2 + S1);
''')
```

To reference model properties and methods, the property or method must be proceeded with the roadrunner variable. e.g. `rr.S1 = 2.3;`

When loaded into libRoadRunner the model will be converted into a set of differential equations. For example, consider the following model:

```
    $Xo -> S1;  v1;
     S1 -> S2;  v2;
     S2 -> $X1; v3;
```

This model will be converted into:

$$\frac{dS_1}{dt} = v_1 - v_2$$

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

Note that there are no differential equations for X_o and X_1. This is because they are fixed and do not change in time. If the reactions have non-unity stoichiometry, this is taken into account when the differential equations are derived.

5.1.1 Initializing of Model Values

To initialize the concentrations and parameters in a model we can add assignments after the network is declared, for example:

```
MainFeed:    $X0 -> S1;  Vm*X0/(Km + X0);
TopBranch:    S1 -> $X1; Vm1*S1/(Km1 + S1);
BottomBranch: S1 -> $X2; Vm2*S1/(Km2 + S1);

X0 = 3.4;  X1 = 0.0;
S1 = 0.1;
Vm = 12; p.Km = 0.1;
Vm1 = 14; p.Km1 = 0.4;
Vm2 = 16; p.Km2 = 3.4;
```

5.1.2 Setting up Compartments

For multi-compartment models, or models where the compartment size changes over time, one can define compartments in Antimony by using the 'compartment' keyword, and designate species as being in particular compartments with the 'in' keyword. For example

```
# Examples of different compartments

compartment cytoplasm = 1.5, mitochondria = 2.6
const S1 in mitochondria
var S2 in cytoplasm
var S3 in cytoplasm
const S4 in cytoplasm

S1 -> S2; k1*S1
S2 -> S3; k2*S2
S3 -> S4; k3*S3
```

Example 5.1

Describe the following network using Antimony. The rate laws are given by $v_1, v_2, \ldots$.

```
   -> A; v1;
A -> B; v2;
```

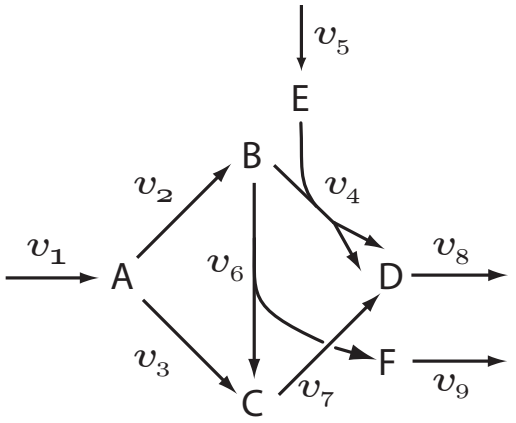


Figure 5.1 Reaction Network

```
    A -> C; v3;
B + E -> 2 D; v4;
      -> E; v5;
    B -> C + F; v6;
    C -> D; v7;
    D -> ; v8;
    F -> ; v9
```

5.2 Including Additional Equations with the Model

Sometime additional equations need to be included with the model. For example, computing the pH during a simulation or changing an input in a defined way. The example below shows how the pH can be computed as te model is simulated:

```
# Computing the pH in the model

pH := -log10 (H)
# These two reaction consume and product hydrogen ions
$Xo -> H; k1*Xo
H -> $X1; k2*H

H = 0.001
```

Note the use of the assignment symbol `:=`. The use of this symbol is to distinguish this expression from a simple assignment which only happens *once* at the start of a simulation. The use of `:=` means that the equation gets evaluated continuously. This means that the pH can be inspected at any time during a simulation. Another application for model equations

is when we wish to input a signal into our model. For example if we wanted to inject a sine wave in to a model we might use the antimony script:

Example 5.2

Assume the following network $X_o \leftarrow S_1 \leftarrow$ where X_o is a boundary metabolite. Write an Antimony model where X_o starts at a value of `startXo = 2`, then increases at a linear rate, k. Assume that the consumption of S_1 is governed by $k_2 S_1$ and product by $k_1 X_o$.

```
startXo = 2
Xo := k*time + startXo;

$Xo -> S1; k1*Xo;
S1 -> ;    k2*S1;

k = 0.75; k1 = 2;  k2 = 0.2; S1 = 0;
```

Notice how the amplitude is assigned using the initialization symbol = since we only need to do this once. The sine function equation uses the := symbol because the sine wave must be injected into the model continuously.

5.3 Loading and Running Models in Python

libRoadRunner is a high performance simulator that can simulate models described using SBML. In order to use Antimony with libRoadRunner it is necessary to first convert an Antimony description into SBML and then load the SBML into libRoadRunner. Telluirum provides a handy routine called `loadAntimonyModel` to help with this task (The short-cut name is loada). To load an Antimony model we first assign an Antimony description to a string variable, for example:

```
model = '''
   S1 -> S2; k1*S1;

   S1 = 10; k1 = 0.1;
'''
```

We now use the `loadAntimonyModel (model)` or `loada` to load the model into libRoadRunner.

```
>>> rr = te.loadAntimonyModel (model)
```

Listing 5.2 Loading an Antimony model

In this book we generally use the short-cut command as follows:

```
rr = te.loada ('''
    S1 -> S2; k1*S1;

    S1 = 10; k1 = 0.1;
 ''')
>>>
```

Listing 5.3 Loading an Antimony model using the short-cut command

Note that `loadAntimonyModel` and `loada` are part of the Tellurium Python package supplied with the Tellurium installer. If the Tellurium packages hasn't been loaded, use the following command to load the Tellurium package:

```
>>> import tellurium as te
```

Listing 5.4 Importing the Tellurium Package

5.3.1 Loading Antimony Models

As we've seen before, loading an Antimony model is most easily achieved by using the `loada` method from the tellurium library.

```
import tellurium as te
r = te.loada ('''
   S1 -> S2; k1*S1;
   S1 = 10; k1 = 0.1;
''')
```

5.4 Models as Differential Equations

Sometimes a model is provided in the form of a set of differential equations. This may be because the model cannot be easily expressed as a reaction scheme. Sometimes it might be desirable to create a hybrid model where part of the model is described as a reaction scheme and another part as one or more differential equations. Antimony allows one to describe differential equations directly using a special syntax. A differential equation such as:

$$\frac{dx}{dt} = f(x)$$

is described in Antimony using:

```
import tellurium as te
```

```
r = te.loada ('''
   x' = f(x)
''')
```

dx/dt is represented by the syntax `x'`. The example below shows how we can use Antimony to expression the famous Lorenz chaotic model. This model has three differential equations for the variables, x, y, and z. We've also included a simulation command that will solve the differential equations (see a later chapter).

```
import tellurium as te

# Example showing how to describe a model using ODES
# Example implements the Lorenz attractor.

r = te.loada ('''
     x' = sigma*(y - x);
     y' = x*(rho - z) - y;
     z' = x*y - beta*z;

     x = 0.96259;  y = 2.07272;  z = 18.65888;

     sigma = 10;  rho = 28; beta = 2.67;
''')

result = r.simulate (0, 20, 1000, ['time', 'x', 'y', 'z'])
r.plot (result)
```

Other models such as predator/prey or mechanical models describing motion can also be expressed this way.

5.5 Exercises

1. Obtain the paper by Edelstein BB, Biochemical Model with Multiple Steady States and Hysteresis, J. theor Bio, 29, 57-62 (1970). Enter the model described in the paper (equations, 1, 2, 3) in Antimony format.

2. Obtain the paper by Heinrich, Rapoport and Rapoport, Metabolic regulation and mathematical models., Prog Biophys Mol Biol. 1977;32(1):1-82. Enter the model described as scheme 3 (page 18) into Antimony foramt.

3. Pick a paper of your own choice and enter the model in Antimony format.

6 Sharing Models

6.1 Sharing Models

Imagine the following scenarios:

1. You find a paper that includes a very interesting model of a signaling pathway. You'd like to get this model running. The paper describes the model in words and has a short appendix where it describes some of the equations they used. You spend the next 4 days trying to get the model working in Matlab, but you just can't get it to work. You try to contact the authors but she says all the files were misplaced when the student left the lab. All that is available is what is in the paper. So you spend another week trying to get the model to work. The frustrating bit is it sort of works but doesn't quite give the same results as descried in the paper. You decide to spend one more week, but it still doesn't work so you give up.

2. You find a paper that includes a very interesting model of a signaling pathway. You'd like to get this model running. The paper describes a model and as part of a supplement provides a file you can download. This file is for a simulation application that is described in another paper. You search out the paper and find a description of the application. The paper also tells you where you can get the executable. Unfortunately it only runs on Linux and you've no idea how to use Linux because the authors expect you to compile the application yourself. You contact the authors but again those who originally knew how to get the application running not longer work in the group. Instead you find a friendly programmer who tries to get the application running. After a week of hacking away your friendly programmer finally has the application running.

You manage to run the model after getting a crash course in Linux but there are some numerical errors. The friendly programmer goes back and has another go and after two days manages to fix the problem. You have a working model. The problem now is, because you don't know much about hardcore programming there is no way to explain to someone else how to run the model on their computer.

3. You find a paper that includes a very interesting model of a signaling pathway. You'd like to get this model running. The paper describes a model and as part of a supplement provides a file you can download. This file is for a simulation application that runs on Windows that is described in another paper. You search out the paper and find a description of the application and a web site where where you can get the executable. The web site still exists, but it looks a little old so you're a bit suspicious. You download the application and find out that there are missing dependent files from the application which means it won't run on the latest version of Windows. Unfortunately the source code isn't available so you can't even try to recompile the application to run the model so you give up.

4. You find a paper that includes a very interesting model of a signaling pathway. You'd like to get this model running. The paper describes a model and as part of a supplement provides a file you can download. This file turns out to be a COMBINE archive which contains everything you need to reproduce the graphs shown in the paper. You find an application that supports the COMBINE archive (such as Tellurium), you load it and it reproduces the paper's results flawlessly.

There are many scenarios like the ones described above where often it is impossible to get a published model working. There are many reasons for this that include:

1. Missing data
2. Incorrect data (units wrong, values wrong)
3. Undefined terms/graph axes
4. Mismatch between text and model
5. Wrong model supplied with paper
6. Only one model supplied but multiple simulations described
7. Simulation environment no longer available
8. Model no longer available (url points to a non-existent page)
9. Model only supplied as a binary

How can we avoid these problems? One way is to have everyone publish their models in an agreed community standard that is both independent of the computer operating system and the application you might use to run the model. Two such standards currently exist, SBML and CellML. We will concentrate on SBML here.

SBML is based on XML and closely follows the way existing modeling packages represent models. For example, SBML represents biochemical networks as a list of chemical transformations. It employs specific and different elements to represent spatial compartments, molecular species, and parameters. In addition, SBML also has provision for rules which can be used to represent constraints, derived values, and general math. SBML (sbml.org), like any standard, has evolved with time. Major revisions of the standard are captured in levels, while minor modifications and clarifications are captured in versions. An example of a major change within the standard would be the use of MathML in level two of SBML, whereas level one encoded infix (common algebra) strings to denote reaction rates and rules. The most recent level of SBML is level three where new functionality can be supported through extension packages.

Along with the standardization of model representation, there has been an obvious desire to create model repositories where models published in journals can be stored and retrieved. There are currently five repositories with varying degrees of quality and usability. The most promising is the UK based BioModels Database, which at the current time (July 2013) holds over nine hundred and sixty three curated and working models that can be downloaded in standard SBML and other formats. BioModels also has the great benefit of providing programmatic access to its database via web services, which allows any software program to access the database seamlessly across the internet. Models stored in the BioModels Database are curated, meaning that models will reproduce the author's original intention. In addition, the models are liberally annotated so model components can be referenced from other database sources.

6.1.1 Loading SBML Models

RoadRunner can also reads models using the SBML format. If you have a SBML model stored on your hard drive, it is possible to load that model either by giving the document contents or the path to the SBML file. Let's assume you have a model called mymodel.xml in `C:\MyModels`. To load this model in Windows we would use the command:

```
import roadrunner
rr = roadrunner.RoadRunner("C:/MyModels/mymodel.xml")
```

On the Mac or Linux one might use:

```
import roadrunner
rr = roadrunner.RoadRunner("/home/MyModels/mymodel.xml")
```

6.1.2 Loading Models from Biomodels

Since the introduction of SBML, a number of institutions have built repositories of published models. Possibly one of the most well known is biomodels. Tellurium, via libRoadRunner, has direct support for accessing models at the biomodels repository. The example below shows roadrunner being loaded with biomodels `BIOMD0000000010`.

```
str = "http://www.ebi.ac.uk/biomodels-main/download?mid=BIOMD0000000010"
rr = roadrunner.RoadRunner(str)
```

6.2 Generating SBML and Matlab Files

Tellurium can import and export standard SBML [1] as well as export Matlab scripts for the current model.

6.2.1 Exporting SBML

To load a model in SBML, load it directly into libRoadRunner. For example:

```
>>> rr = roadrunner.RoadRunner ('mymodel.xml')
>>> result = rr.simulate (0, 10, 100)
```

There are two ways to retrieve the SBML, one can either retrieve the original SBML loaded using `rr.getSBML()` or retrieve the *current* SBML using `rr.getCurrentSBML()`. Retrieving the current SBML can be useful if the model has been changed. To save the SBML to a file we can use the Tellurium helper function `saveToFile ()`, for example:

```
>>> te.saveToFile ('mySBMLModel.xml', rr.getCurrentSBML())
```

6.2.2 Exporting Matlab

To convert an SBML file into Matlab, use the `getMatlab` method:

```
import tellurium as te

rr = te.loada ('''
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2;
    S1 = 10; k1 = 0.1; k2 = 0.2;
''')
```

```
# Save the SBML
te.saveToFile ('model.xml', rr.getSBML())

# Save the Matlab
te.saveToFile ('model.m', rr.getMatlab())
```

6.3 Test models

Tellurium comes with a set of test models than can be easily accessed using the testing API. To get a list of current test models, enter the following:

```
import tellurium as te

print te.listTestModels()
```

This will return a list of string names where the names are names of models. To load a particular test model enter the following:

```
import tellurium as te

sbmlStr = te.getTestModel ('feedback.xml')
```

This will return the SBML of the model as a string. To load the test model directly into libRoadRunner enter the following:

```
import tellurium as te

r = te.loadTestModel ('feedback.xml')
m = r.simulate (0, 10, 100)
r.plot()
```

6.4 Exercises

1. Enter the following model into Tellurium using Antimony:

```
    $S1 -> S2; k1*S1;
    S2 -> S3;  k2*S2;
    S3 -> $S4  k3*S3;
    S1 = 10; k1 = 0.1; k2 = 0.2; k3 = 0.3;
```

Save the SBML for this model to your hard drive. Look at the file in an editor and try to

identify the various parts of the model.

2. Obtain a copy of another simulator, possibilities include Coyote (http://wp.me/P41o0Y-cV), COPASI (http://www.copasi.org/ or SBMLsimulator. (http://www.ra.cs.uni-tuebingen.de/software/SBMLsimulator/).

3. Using the same model, covert the model into a Matlab function. Look at the Matlab file in an editor and try to identify the various parts of the model.

7 Running a Simulation

7.1 Time Course Simulation

Once a model has been loaded into libRoadRunner, performing a simulation is very straight forward. To simulate a model we use the libRoadRunner `simulate` method. This method has many options but for everyday use four options will suffice. The following panel illustrates a number examples of how to use simulate.

```
>>> result = r.simulate ()
>>> result = r.simulate (0, 10)
>>> result = r.simulate (0, 10, 100)
>>> result = r.simulate (0, 10, 100, ['time', 'S1'])
```

Listing 7.1 Calling the simulate method

Argument	Description
1st	Start Time
2nd	End Time
3rd	Number of Points
4th	Selection List

Let us focus on the forth version of the simulate method that takes four arguments. This call will run a time course simulation starting at time zero, ending at time 10 units, and generating 100 points. The results of the run are deposited in the array `result`. At the end

of the run, the `result` array will contain columns corresponding to the time column and all the species concentrations as specified by the forth argument. The forth argument can be used to change the columns that are returned from the simulate method. For example:

```
>>> result = r.simulate (0, 10, 1000, ['S1'])
```

will return an array 1,000 rows deep and one column wide that corresponds to the level of species `S1`.

Note that the special variable `Time` is available and represents the independent time variable in the model.

To visualize the output in the form of a graph, we call the libRoadRunner plot method. In the following example we return one species level, `S1` and three fluxes. Finally we plot the results.

```
   result = r.simulate (0, 10, 1000, ['Time', 'S1', 'J1', 'J2', 'J3']);
   r.plot()
```

or if we are not interested in keeping the data itself we can use the libRoadRunner plot:

```
   r.simulate (0, 10, 1000, ['Time', 'S1', 'J1', 'J2', 'J3']);
   r.plot()
```

It is possible to set the output column selections separately using the command:

```
r.selections = ['time', 'S1']
```

This can save some typing each time a simulation needs to be carried out. By default the selection is set to time as the first column followed by all molecular species concentrations. As such it is more common to simply enter the command:

```
>>> result = r.simulate (0, 10, 50)
```

In fact even the start time and end time and number of points are optional and if missing, simulate will revert to its defaults.

```
>>> result = r.simulate()
```

7.1.1 Plotting Simulation Results

Tellurium comes with Matplotlib, a common plotting package used by many Python users. To simplify its use we provide two simple plotting calls:

```
te.plotArray (array)
te.plotWithLegend (r, array)
r.plot()
```

The first takes the resulting array generated by a call to `simulate` and uses the first column as the x axis and all subsequent columns as y axis data. The second call takes the roadrunner variable as well as an array and does the same kind of plot but this time adds a legend to the plot. We will use the second plotting command in the next section where we merge together multiple simulations. The third plot command is associated with libRoadRunner and will plot the last set of data generated from a simulation. The advantage of the first two calls is that they can take additional matplotlib settings.

7.1.2 Setting and Getting Values

Often during a modeling experiment one will need to change parameter values, initial condition or inspect values after a simulation has completed. For model parameters and species concentration one can access to these values using the syntax `r.X`. If a model has a kinetic constant `k1`, then its value can be changed or inspected using the following syntax, assuming `r` is the roadrunner object:

```
r.k1 = 1.2
print r.k1
```

If a model contains the species ATP, then its value can be changed or inspected using the following syntax, assuming `r` is the roadrunner object:

```
r.ATP = 1.5
print r.ATP
```

Changing initial conditions is slightly different. The initial conditions represent the values of the species that are the starting values for a simulation. The easiest way to set the starting values is to use the method `reset`. This copies the current set of initial conditions to the model. We can also change the initial conditions if we wish. There are a number of methods to get and set the initial conditions of a loaded model. The values stored in the initial conditions are applied to the model whenever it is reset.

To set the initial conditions we can use the following syntax:

```
# Set the initial concentration of S1
r.model['init([S1])'] = 3.4
print r.model['init([S1])'
```

The 'init([S1])' string is used to indicate the particular state variable we wish to set the initial condition for. We can set both the amount or concentration of a species. For example

to set the amount we simply leave out the square bracket:

```
# Set the amount of S1
r.model['init(S1)'] = 3.4
print r.model['init(S1)'
```

We can also retrieve the init strings using the two calls:

```
print r.model.getFloatingSpeciesInitAmountIds()
['init(S1)', 'init(S2)']
r.model.getFloatingSpeciesInitConcentrationIds()
['init([S1])', 'init([S2])']
```

It is also possible to set the entire vector of initial conditions using the method calls:

```
r.model.setFloatingSpeciesInitAmounts ([2.6, 7,8])
r.model.setFloatingSpeciesInitConcentrations ([1.2, 34.5])
```

Retrieving Reaction Rates or Fluxes The fluxes through the individual reactions can be obtained by either referencing the name of the reaction (e.g. `J1`), or via the short-cut command `rv`. The advantage to looking at the reaction rate vector is that the individual reaction fluxes can be accessed by indexing the vector (see example below). **Note that indexing is from zero.**

```
>>> print rr.J1, rr.J2, rr.J3
3.4, ...
>>> for i in range (0, 2):
...    print rr.rv()[i]
3.4
etc
->
```

7.1.3 Selecting Output from a Simulation

RoadRunner supports a range of options for selecting what data a simulation should return. The simulate method, by default returns an structured array, which are arrays that also contain column names. These can be plotted directly using the built in `rr.plot()` function, or by adding the `plot=True` keyword argument to simulate().

The output selection defaults to time and the set of floating species. It is possible to change the simulation result values by changing the selection list. For example assume that a model has three species, `S1`, `S2`, and `S3` but we only want simulate() to return time in the first column and `S2` in the second column. To specify this we would type:

```
result = r.simulate (0, 10, 100, ['time', 'S2'])
```

In another example let say we wanted to plot a phase plot where S1 is plotted against S2. To do this we type the following:

```
result = r.simulate(0, 10, 100, ['S1', 'S2'])
```

7.1.4 Resetting Simulations

There are three important reset methods:

- `reset`: Copies the current initial conditions to the model ready for a simulation
- `resetAll`: Copies the current initial conditions and the set of parameters that were originally loaded.
- `resetToOrigin`: resets the model back to what is was when the model was first loaded.

For example:

```
import tellurium as te

rr = te.loada ('''
  J1: S1 -> S2; k1*S1;
  J2: S1 = 10; k1 = 0.1
''')

m1 = rr.simulate (0, 50, 100)
rr.reset()
rr.k1 = 0.5
m2 = rr.simulate (0, 50, 100)
```

Listing 7.2 Named Reactions

7.1.5 Access to Fluxes or Rates of Reaction

Reactions in an Antimony model can be named. For example, the model shown below defines two reactions where the first reaction is given the name `J1` and the second reaction `J2`.

```
import tellurium as te
```

```
rr = te.loada ('''
  J1: S1 -> S2; k1*S1;
  J2: S1 = 10; k1 = 0.1
''')

rr.simulate (0, 50, 100)
rr.plot()
```

Listing 7.3 Named Reactions

The reaction rate through a given reaction can be accessed by referencing the name of the reaction. For example to print out the reaction rate for reaction `J1`, we can use the Python code:

```
>>> print rr.J1
```

The reaction names can also be used in selection lists. Thus if we wish to plot the change in reaction rates as a function of time we can use the syntax:

```
>>> result = rr.simulate(0, 10, 100, ['time', 'J1', 'J2])
```

Alternatively the vector of reactions rates can be obtained using the method:

```
>>> r.getReactionRates()
```

which returns an array of values, corresponding to the reaction rates returned by the method `r.getReactionIds()`.

7.1.6 Access to Rates of Change

During a time course simulation, species concentrations will change. The rate at which a species, `x`, changes is represented by the expression `"x'"`. Given the model shown below:

```
import tellurium as te

rr = te.loada ('''
  J1: S1 -> S2; k1*S1;
  J2: S1 = 10; k1 = 0.1
''')

rr.simulate (0, 50, 100)
rr.plot()
```

Listing 7.4 Named Reactions

the rate of change of species S1 and S2 can be obtained using the syntax:

```
>>> print r.model["S1'"], r.model["S2'"]
```

Alternatively the vector of rates of change can be obtained using the method:

```
>>> r.getRatesOfChange()
```

which returns an array of values, corresponding to the species names returned by the method `r.model.getFloatingSpeciesIds()`.

Rates of change can also be set in the `simulate` selection list:

```
import tellurium as te

rr = te.loada ('''
  J1: S1 -> S2; k1*S1;
  J2: S1 = 10; k1 = 0.1
''')

# Note the use of the double quote that allows us to use S1'
m = rr.simulate (0, 50, 100, ['time', 'S1', "S1'"])
```

Listing 7.5 Named Reactions

7.1.7 Applying Perturbations to a Simulation

Often in a simulation we may wish to perturb a species or parameter at some point during the simulation and observe what happens. One way to do this in Tellurium is to carry out two separate simulations where a perturbation is made between two flanking simulations. For example, let's say we wish to perturb the species concentration for a simple two step pathway and watch the perturbation decay. First, we simulate the model for 10 time units; this gives us a transient and then a steady state.

```
import numpy # Required for vstack
import tellurium as te

rr = te.loada ('''
     $Xo -> S1;  k1*Xo;
      S1 -> $X1; k2*S1;

     Xo = 10; k1 = 0.3; k2 = 0.15;
''')

m1 = rr.simulate (0, 40, 50)
```

We then make a perturbation in S1 as follows:

```
rr.S1 = rr.S1 * 1.6
```

which increases S1 by 60%. We next carry out a second simulation:

```
m2 = rr.simulate (40, 80, 50)
```

Note that we set the time start of the second simulation to the end time of the first simulation. Once we have the two simulations we can combine the matrices from both simulations using the Python command vstack

```
% Merge the two result arrays together
m = numpy.vstack ((m1, m2))
```

Finally, we plot the results, screen-shot shown in Figure 7.1.

```
te.plotArray (m)
```

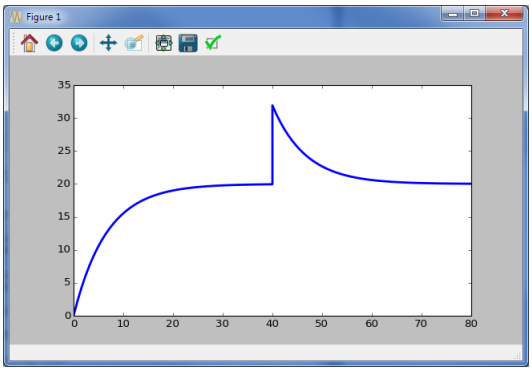


Figure 7.1 Screen-shot from Matplotlib showing effect of perturbation on S1.

7.1.8 Using Antimony to Implement Events

The Antimony languages allows user to include events that should happen at specific times. For example, in the previous model S1 was increased 60% at time 40. In Antimony we would specify this using the at syntax:

```
import tellurium as te

rr = te.loada ('''
     $Xo -> S1;  k1*Xo;
```

```
      S1 -> $X1; k2*S1;

     Xo = 10; k1 = 0.3; k2 = 0.15;

     at time > 40: S1 = S1*1.6
''')

m = rr.simulate (0, 80, 200)
rr.plot()
```

7.2 Perturbations and Events

```
import tellurium as te
import pylab as plt

r = te.loada("""

    v := alpha*sin (time) + k1*Xo + 2;

    $Xo -> S1; v;
   J1: S1 ->; k2*S1;

     k1 = 0.1; k2 = 0.2; Xo = 1;
     alpha = 1;

     at (time > 40): alpha = 0;
""")

m = r.simulate(0, 100, 200, ['time', 'S1', 'J1'])
plt.plot (m[:,0], m[:,1], label='k1=' + str (r.k1))
plt.plot (m[:,0], m[:,2], label='k2=' + str (r.k2))
plt.legend(loc='center')
```

7.3 Other Model Properties of Interest

There are a number of predefined objects associated with a reaction network model which might also be of interest. For example, the stoichiometry matrix, `sm`, the rate vector `rv`, the species levels vector `sv` and `dv` the rates of change vector.

```
print rr.sm()  # Print the stoichiometry matrix
print rr.rv()  # Print vector of reaction rates
print rr.sv()  # Print the vector of species concentrations
```

```
print rr.dv()  $ Print the vector of rates of change
```

The names for the parameters and variables in a model can be obtained using the short-cuts:

```
print rr.fs()  # List of floating species names
print rr.bv()  # List of boundary species names
print rr.ps()  # List of parameter names
print rr.rs()  # List of reaction names
print rr.vs()  $ List of compartment names
```

7.4 More on Plotting

In a previous chapter we described some of the basic plotting facilities that come with the Python package matplotlib. In particular the plot command is used to plot data. One small issue with the plot command than can make its use a little tedious is that is requires separate x and y arrays whereas the `simulate` command returns a single array with the first column the independent data and subsequent columns the dependent data. It is possible to use Python's slicing syntax to extract the x and y columns as in:

```
plot (result[:,0],result[:,1:])
```

which splits the columns into the first column and all remaining columns. To help new users avoid such issues we provide two methods of interest. The first is a plot commend directly on the roadrunner variable. For example:

```
r.plot()
```

where `r` is a roadrunner variable. This will plot the last data simulated, using the first column as the independent variable and subsequent columns as the dependent columns. The nice feature about this command is that it also adds a legend to the plot. For more flexibility Tellurium also offers the `plotArray` command. An example of how to use it is given below:

```
import tellurium as te

m = rr.simulate (0, 80, 200)
te.plotArray (m)
```

`plotArray` takes a single argument which is an array of data. The advantage of plotArray is that is can be used in conjunction with the `show=False` which allowing multiple graphs to be plotted on the same plot, for example:

```
import tellurium as te

r.k1 = 1.5
```

```
m = rr.simulate (0, 80, 200)
te.plotArray (m, show=False)

r.reset()
r.k1 = 2.0
m = rr.simulate (0, 80, 200)
te.plotArray (m, show=True)
```

7.4.1 Controlling Axes etc

There will often be situation where more control over a plot is required. For example a common need is to be able to control the axes limits. To change the characteristics of a plot first import ptlab:

```
import pylab
```

With pylab one can now set things like axes limits. For example to set the limits on the y and x axes and axes titles use:

```
>>> pylab.xlim ((0,5))
>>> pylab.ylim ((0,10))

>>> pylab.xlabel ('Time')
>>> pylab.ylabel ('Concentration')
>>> pylab.title ('My First Plot ($y = x^2$)')
```

Here is a complete example:

```
import tellurium as te
import pylab

# Example showing how to embellish a graph, change title, axes labels.
# Example also uses an event to pulse S1

r = te.loada ('''
   $Xo -> S1; k1*Xo;
   S1 -> $X1; k2*S1;

   k1 = 0.2; k2 = 0.4; Xo = 1; S1 = 0.5;
   at (time > 20): S1 = S1 + 0.35
''')

# Simulate the first part up to 20 time units
m = r.simulate (0, 50, 100, ["time", "S1"]);
```

```
plt.ylim ((0,1))
plt.xlabel ('Time')
plt.ylabel ('Concentration')
plt.title ('My First Plot ($y = x^2$)')
r.plot (m)
```

7.5 Exercises

1. Build a model of a closed system:

$$So \xrightarrow{v_1} S1 \xrightarrow{v_2} S2 \xrightarrow{v_3} S3$$

a) What is a closed system?

b) If the system is closed the concentrations of S_o and S_3 should be fixed. Why is this?

Assume that the rate laws for the three reactions are given by:

```
So -> S1;   k1*So - k2*S1;
S1 -> S2;   k3*S1 - k3*S2;
S2 -> S3;   k5*S2 - k6*S3;
```

and the following parameter values:

```
So = 4;    S3 = 0;
k1 = 1.2;  k2 = 0.45;
k3 = 0.56; k4 = 0.2;
k5 = 0.89; k6 = 0;
```

c) Carry out a simulation of this system and plot the time course for the concentrations of S_o, S_1, S_2 and S_3 using t = 0 to t = 50. Once the system settles down what is the net flux through the pathway? Hint: You may need to name a reaction to get hold of the flux.

2. Turn the previous system into an open system by fixing S_o and S_3. Rerun the same simulation. What is the net flux through the pathway?

What is the difference in the net flux between the close and open system?

3. Figure 7.2 shows a two gene circuit with a feedforward loop. Assume the following rate

laws for the four reactions:

$$v_1 = k_1 X_o$$
$$v_2 = k_2 x_1$$
$$v_3 = k_3 X_o$$
$$v_4 = k_4 x_1 x_2$$

Assume that all rate constants are equal to one and that $X_o = 1$. Assume X_o is a fixed species.

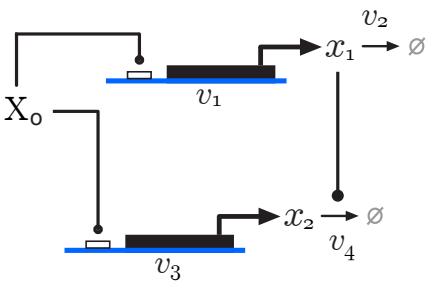


Figure 7.2 Two gene circuit with feedfoward loop.

Based on this model answer the following questions:

a) Express this model as an Antimony script

b) Write out the differential equations for x_1 and x_2.

c) Run a simulation of the system from 0 to 10 time units.

d) Change the value of X_o to 2 (double it) and rerun the simulation for another 10 time units from where you left off in the last simulation. Combine both simulations and plot the result, that is time on the x-axis, and X_o and x_2 on the y-axis.

e) What do you see from the last simulation?

f) Show algebraically that the steady state level of x_2 is independent of X_o (Hint: Set the differential equations to zero)

4. Consider the following model:

$$\rightarrow S \quad v_o$$
$$2S \rightarrow \quad k\ S^2$$

a) Write out the differential equation for S.

b) Enter the model into Tellurium and set the following values:

$k1 = 0.5,\ vo = 2,\ S = 0$

Run a simulation from 0 to 10. What do you observe?

c) Set the value of $S = 0.000001$. Rerun the simulation, what do you observe?

d) Derive the analytical solution for the steady state and attempt to explain your observations.

5. Let us now consider an enhancement to the previous model

$$\begin{aligned} \rightarrow S \quad & k_o + k_1 S^2/(k_2 + S^2) \\ S \rightarrow \quad & k\ S \end{aligned}$$

This time we have made the consumption step a simple first order reaction. The first step is slightly more complicated. It has a basal rate of k_o and a nonlinear positive feedback rate given by a Hill function with a Hill coefficient of 2. Setup up the model by using the following parameters values:

$$vo = \frac{6}{11}, k1 = \frac{60}{11}, k2 = 11, k3 = 1$$

Run the following experiments

a) Set $S = 0.6$ and run a simulation from time zero to 60. Record the value of S.

b) Set $S = 5.6$ and run a simulation from time zero to 60. Record the value of S.

c) What do you observe?

d) Run many simulations for a range of starting values of

$S = [0.9, 1.1, 1.3, 1, 5, 1.7, 2.1, 2.3, 2.6]$.

Graph each simulation on the same plot so you can observe the effect.

Hint: To plot multiple simulations on the same graph you can use `te.plotarray (m)` together with the argument `show=False`.

```
import tellurium as te

# Example of how to hold a plot in Tellurium

m = r.simulate (0, 80, 200)
te.plotArray (m, show=False)

# Change something, simulate and replot
m = r.simulate (0, 60, 100)
te.plotArray (m, show=True)
```

What do you observe?

6. Repeat the last exercise but this time you can only use the matplotlib method, `plot`. Use the internet to look up how to use the plot method and a python technique call slicing in order to extract the appropriate columns from the simulation result array.

8

Steady State Analysis

8.1 Steady State

To evaluate the steady-state first make sure the model values have been previously initialized, then enter the following statement at the console.

```
>>> rr.steadyState()
```

This statement will attempt to compute the steady state and return a value indicating how effective the computation was. It returns the norm of the rate of change vector (i.e. $\sqrt{\sum dydt}$). The closer this is to zero, the better the approximation to the steady state. Anything *less* than 10^{-4} usually indicates that a steady state has been found.

Once a steady state has been evaluated, the values of the species will be at their steady state values, thus `S1` will equal the steady state concentration of `S1`.

Exercise

Find the steady state for the following system using the steadyState command:

```
import tellurium as te

r = te.loada ('''
      $Xo -> S1; k1*Xo;
      S1 -> $X1; k2*S1;
```

```
      Xo = 5.6; X1 = 0;
      k1 = 0.34; k2 = 0.67;
''')
```

The command, `r.dv()` will return a vector of current rates of change. `dv` is a shorthand for `r.getRatesOfChange()`. Likewise `r.rv()` will return a vector of current reaction rates – shorthand for `r.getReactionRates()`.

Use the commands `r.dv()` and `r.rv()` to show that the system is at steady state.

8.1.1 Stability Analysis

The stability of a steady state can be determined by computing the eigenvalues of the Jacobian computed at the steady state. The jacobian matrix can be returned using the command: `rr.getFullJacobian()`). The following example illustrates this method.

```
import tellurium as te

r = te.loada ('''
      $Xo -> S1; k1*Xo;
      S1 -> S2; k2*S1 - k3*S2;
      S2 -> $X1; k4*S2;

      Xo = 5.6; X1 = 0;
      k1 = 0.34; k2 = 0.67;
      k3 = 0.12; k4 = 0.87l
''')

r.steadyState();
print r.getFullJacobian()
>>>
          S1,    S2
S1 [[ -0.67,  0.12],
S2  [  0.67, -0.99]]
```

To determine the eigenvalues of the Jacobian we use the tellurium method `getEigenvalues` as follows:

```
import tellurium as te

r = te.loada ('''
      $Xo -> S1; k1*Xo;
      S1 -> S2; k2*S1 - k3*S2;
      S2 -> $X1; k4*S2;
```

```
      Xo = 5.6; X1 = 0;
      k1 = 0.34; k2 = 0.67;
      k3 = 0.12; k4 = 0.871
''')

r.steadyState();
print te.getEigenvalues (r.getFullJacobian())
array([-0.50442359, -1.15557641])
```

An alternative is to use the libRoadrunner method `r.getFullEigenValues()`

```
print r.getFullEigenValues ()
array([-0.50442359, -1.15557641])
```

In this case both eigenvalues are negative indicating that the steady state is stable.

8.2 Metabolic Control Analysis

Metabolic control analysis (MCA) is a mathematical framework for describing metabolic, signaling and genetic pathways. MCA quantifies how variables, such as fluxes and species concentrations, depend on network parameters. In particular it is able to describe how network dependent properties, called control coefficients, depend on local properties called elasticities. MCA was originally developed to describe the control in metabolic pathways but was subsequently extended to describe signaling and genetic networks.

Metabolic control analysis is the study of how sensitive the system is to perturbations in parameters and how those perturbations propagate through the network. Two kinds of sensitivity are defined, system and local. The local sensitivities are described by the elasticities. These are defined as follows:

$$\varepsilon_S^v = \frac{\partial v}{\partial S}\frac{S}{v} = \frac{\partial \ln v}{\partial \ln S}$$

Given a reaction rate v_i, the elasticity describes how a given effector of the reaction step affects the reaction rate. Because the definition is in terms of partial derivatives, any effector that is perturbed assumes that all other potential effectors are unchanged.

The system sensitivities are described by the control and response coefficients. These come in two forms, flux and concentration. The flux control coefficients measures how sensitive a given flux is to a perturbation in the local rate of a reaction step. Often the local rate is perturbed by changing the enzyme concentration at the step. In this situation the flux control coefficient with respect to enzyme E_i is defined as follow:

$$C_{E_i}^J = \frac{dJ}{dE_i}\frac{E_1}{J} = \frac{d\ln J}{d\ln E_i}$$

Likewise the concentration control coefficient is defined by:

$$C_{E_i}^{S} = \frac{dS}{dE_i}\frac{E_1}{S} = \frac{d\ln S}{d\ln E_i}$$

where S is a given species. The response coefficients measure the sensitivity of a flux or species concentration to a perturbation in some external effector. These are defined by:

$$R_X^J = \frac{dJ}{dX}\frac{X}{J} = \frac{d\ln J}{d\ln X}$$

$$R_X^S = \frac{dJ}{dX}\frac{X}{S} = \frac{d\ln S}{d\ln X}$$

where X is the external effector.

8.2.1 Control Coefficients

To compute control coefficients use the statement:

`x = getCC (Dependent Measure, Independent parameter)`

The dependent measure is an expression usually containing flux and metabolite references, for example, `S1, J1`. The independent parameter must be a simple parameter such as a Vmax, Km, ki, boundary metabolite (X0), or a conservation total such as `cm_xxxx`. Examples include:

```
rr.getCC ('J1', 'Vmax1')
rr.getCC ('J1', 'Vm1') + rr.getCC ('J1', 'Vm2')
rr.getCC ('J1', 'X0')
rr.getCC ('J1', 'cm_xxxx')
```

8.2.2 Elasticity Coefficients

To compute elasticity coefficients use the statement:

`x = getEE (Reaction Name, Parameter Name)`

For example:

```
rr.getEE ('J1', 'X0')
rr.getEE ('J1', 'S1')
```

Since `getCC` and `getEE` are built-in functions, they can be used alone or as part of larger expressions. Thus, it is easy to show that the response coefficient is the product of a control coefficient and the adjacent elasticity by using:

```
R = rr.getCC ('J1', 'X0')
print R - rr.getCC ('J1', 'Vm') * rr.getEE ('J1', 'X0')
```

8.3 Exercises

1. Consider the following model which we've see in a previous chapter:

$$\begin{aligned} \rightarrow S \quad & k_o + k_1 S^2/(k_2 + S^2) \\ S \rightarrow \quad & k\ S \end{aligned}$$

The first step has a basal rate of k_o and a nonlinear positive feedback rate given by a Hill function with a Hill coefficient of 2. The consumption step, $k\ S$, is a simple first order reaction. Setup the following parameters values:

$vo = 6/11, k1 = 60/11, k2 = 11, k3 = 1$

Run the following experiments

a) Set $S = 0.6$ and find the steady state using the steady state function.

b) Set $S = 5.6$ and find the steady state using the steady state function. Record the value of S.

c) What do you observe?

d) Call the steady state function many times for a range of starting values of

$S = [0.9, 1.1, 1.3, 1, 5, 1.7, 2.1, 2.3, 2.6]$.

e) How many steady states did you find?

2. For each steady state you found in question 1 determine its stability properties

9 Running Parameter Scans

9.1 Introduction

There are two ways to carry out parameter scans in Telluirum/Python. The first is to write your own piece of code to do the scanning. The second is to use the supplied package ParameterScan that provides a different way to run simulations and plot graphs. In this chapter we will only look at writing your own scanning code.

First thing to do is look at some code that will run a series of time course simulation, plot each solution and add a legend.

```
# Parameter Scan
# This code will run five simulations, each simulation
# having a different values for a rate constant

import tellurium as te
import numpy as np
import pylab
from matplotlib.pyplot import cm

r = te.loada ('''
    J1: $X0 -> S1; k1*X0;
    J2: S1 -> $X1; k2*S1;

    X0 = 1.0; S1 = 0.0; X1 = 0.0;
    k1 = 0.4; k2 = 2.3;
''')
```

```
for r.k1 in np.linspace (0.4, 5, 5):
    r.reset()
    m = r.simulate (0, 4, 100, ["Time", "S1"])
    te.plotArray (m, resetColorCycle=False, label='k1 = ' + str (r.k1), show=False)

pylab.legend()
```

The script first setups up the model and initializes the parameters and initial conditions. We use a for loop to create a range of values for the rate constant k_1. Note how we use `r.k1` as the variable to assign the new values to. Next within the for loop, we do three things. First we reset the simulation. What this does is reset all the species concentrations back to their original values (Note is doesn't change the parameter values). The second statement within the for loop, carries out the actual simulation and returns the results in `m`. The third and final statement in the for loop is to plot the result. We use `plotarray` to to do this. This method takes its first argument to be the result array followed by optional matplotlib settings. The first is to `resetColorCycle=False`, this ensures that we get a different colored line on each turn of the loop. The other thing is to set `show=False` to that each plot is overlayed on the other. We assign a label to the simulation, a label that indicate the value of k_1 we used in the simulation. These labels will be used to create the legend. Once out of the for loop we to display the legend. Figure **??** shows the result.

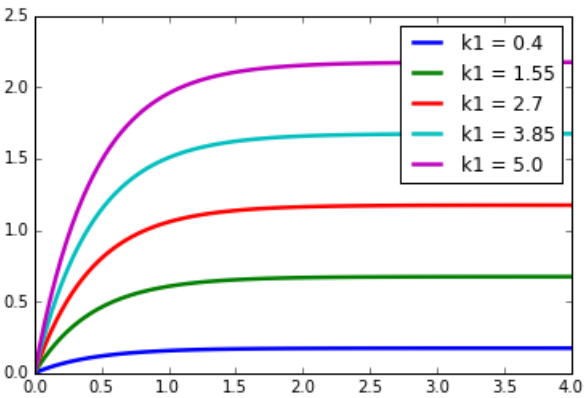


Figure 9.1 Plotting time course simulations as a function of k_1. Includes a legend to indicate what line is what.

In the following example we repeat the last example but this time we only use standard matplotlib commands.

```
# Parameter Scan
# This code will run five simulations, each simulation
# having a different values for a rate constant
```

```
import tellurium as te
import numpy as np
import pylab

r = te.loada ('''
    J1: $X0 -> S1; k1*X0;
    J2: S1 -> $X1; k2*S1;

    X0 = 1.0; S1 = 0.0; X1 = 0.0;
    k1 = 0.4; k2 = 2.3;
''')

for r.k1 in np.linspace (0.4, 5, 5):
    r.reset()
    m = r.simulate (0, 4, 100, ["Time", "S1"])
    pylab.plot (m[:,0], m[:,1], label='k1 = ' + str (r.k1))

pylab.legend()
```

The example illustrates the use of array slicing to extract the appropriate columns of data from the result array.

10 Model Fitting

10.1 Introduction

In constructing computational models of biochemical systems, we make choices about what reaction steps, regulatory interactions and molecular species to include. Given these choices, how good is the model? Does the model adequately describe existing knowledge about the system? Can the model make useful predictions? Some of the model parameters might be estimated experimentally but many will be unknown. How can we estimate these parameters and how well can they be estimated? Such questions fall under the umbrella of model fitting.

Fitting a model means adjusting the parameters of the model until the behavior of the model matches some known experimental data. Details of model fitting are given in the companion book: Systems Biology: Introduction to Pathway Modeling [**?**] and only a very brief explanation will be given here.

10.2 Fitting Models

To understand how the fitting process works, consider a simple model:

$$S_1 \xrightarrow{k_1} S_2$$

We start an experiment with an initial amount of S_1 and observe the change in S_1 as it reacts to form S_2. Figure 10.1 shows both a solid curve representing a simulation of the model,

and four experimental data points for the concentration of S_1. The first data point at time zero represents the initial concentration of S_1 which we assume is error free. Measurements are collected at time points 0.5, 1, 2, and 3.5. The e_i terms represent the difference between the experimental data point and the simulation curve. Fitting is the process where we attempt to adjust the parameters of the model (k_1 in this case), such that the difference, e_i, between the simulation curve and the data points is **minimized**.

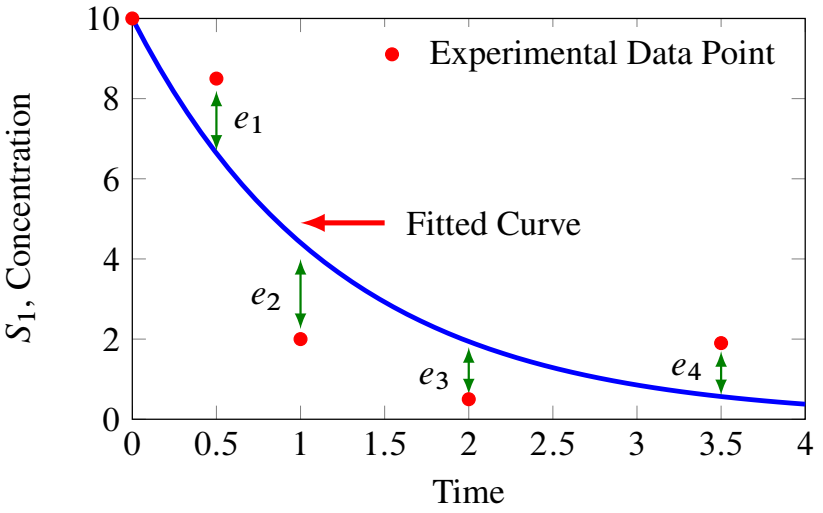


Figure 10.1 Model curve and experimental data plotted on the same graph. The solid line is the simulated model, the points represent experimental data. The experimental data has errors, e_i, such that they do not exactly match the model curve. Model fitting attempts to minimize the e_i terms by adjusting the model parameter values.

Let us indicate the experimental data points using the symbols, x_i and y_i, where x_i is the independent variable time and y_i the dependent variable. Assume there are N data points. We will indicate the model using the expression $f(x_i; p_1 \dots p_m)$, where p_i is the ith parameter in the model. That is, for a given set of parameters and time point x_i, the function f will return the corresponding model y_i^m value. If the model is a set of differential equations, we would run a simulation in order to obtain the value of y_i^m at x_i. The fitting procedure will attempt to minimize the difference between the model f and the data points, that is minimize:

$$y_i - f(x_i; p_1 \dots p_m)$$

Because the difference between a data point and the model may be positive or negative depending on the error in the data point (See e_2 for example in Figure 10.1), we take the square of the difference to make the term positive:

$$(y_i - f(x_i; p_1 \dots p_m))^2$$

This difference only corresponds to one data point, and we should be considering all data points when trying to fit the model. Therefore we sum up all the differences and attempt to minimize the total sum, that is:

$$\sum_{i=1}^{N}(y_i - f(x_i; p_1 \dots p_m))^2$$

We can take this one step further and reason that the most uncertain data points should contribute less to the sum compared to those which have been measured more precisely. We therefore weight each difference by the standard deviation, σ, that corresponds to that data point. This assumes that we have some measure of uncertainty, if we don't we set the weight to one:

$$\chi^2 \equiv \sum_{i=1}^{N}\left(\frac{y_i - f(x_i; p_1 \dots p_m}{\sigma_i}\right)^2 \tag{10.1}$$

The above equation can also be expressed in the following equivalent form to emphasize the weighing in terms of the variance, σ^2:

$$\chi^2 \equiv \sum_{i=1}^{N}\frac{1}{\sigma_i^2}(y_i - f(x_i; p_1 \dots p_m))^2$$

This equation is called the **weighted chi-square sum of squares**[1] and can vary between zero and infinity. If the model is a set of differential equations, the f function is a list of data points from a simulation run. For example, using the previous model let us assume the parameter k_1 is set to -0.95. Table **??** shows an example of computing the chi-square given some data points and results from a model run.

An important variant on the chi-square is the **reduced chi-square** (10.2) which is used when looking at the quality of the fit and estimating the confidence in the fitted parameter.

$$\chi^2_{\text{reduced}} \equiv \frac{1}{N-P}\sum_{i=1}^{N}\frac{1}{\sigma_i^2}(y_i - f(x_i; p_1 \dots p_m))^2 \tag{10.2}$$

N is the number of data points and P the number of parameters to be fitted in the model. The difference $N - P$ is called the **degrees of freedom**. This measure gives insight into whether a model is over-fitted in the sense that there is much more data that parameters to be estimated.

[1] The notation χ^2 is possibly misleading. The χ^2 is not the square of a quantity χ and is why the term chi-square is used rather than chi-squared. The ^{2} is simply to remind us of the square on the right-hand side of the definition.

10.3 Optimization Algorithms

A brute force method for fitting a model is to run a simulation of the model many times with random parameter values until we find a set of parameters that gives us simulation data that matches the experimental time series. One problem with this approach is that we will spend a great deal of time coming up with random parameter values in the hopes that at least one set will match the experimental data. This however is unlikely, and the brute force method is rarely used in practice. Instead, special search algorithms have been devised, called **optimization algorithms**, to search for the best set of parameters in a systematic way.

Optimization is an iterative process. It involves making an initial guess for the parameters, p_i, computing the χ^2 value, and using a rule that adjusts the parameter values such that the χ^2 is reduced in the next iteration. This procedure is repeated many times until the χ^2 can no longer be reduced, at which point the iteration stops. If the fit was successful, the model should be able to reproduce the experimental data given the final set of parameters.

There are many optimization algorithms to choose from and the python scipy library has a large variety of possible optimization methods to choose from. Popular methods include:

1. Nelder and Mead
2. differential_evolution
3. Levenberg-Marquardt

These methods are described in more detail in [**?**]. Possibly the most important aspect of fitting a model is the computation of the chi-square sum of squares. When computing the chi-square for differential equation models, we must compute the time course trajectory and determine the chi-square from the difference between the simulated time course and the time course determined experimentally. For example, consider the simple three reaction model shown below.

```
import tellurium as te

r = te.loada("""
   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;

   S1 = 1; S2 = 0; S3 = 0;
   k1 = 0.15; k2 = 0.45;
""")
```

Let us assume we have experimental data for the change over time in `S2` given some defined initial conditions. We can run the model using libRoadRunner and compare the experimental time course to the simulated time course. The function below gives such a function:

```
def computeChiSquare(p):
    r.reset()
    for i in range(0, fit.nParameters):
        r.model[fit.toFit[i]] = p[i]
    m = r.simulate (fit.timeStart, fit.timeEnd, fit.numberOfPoints)

    a1 = y_data - m[:,2]
    return numpy.sum (a1*a1)
```

Let's look at this function in detail. The first line resets the model back to it initial state. We must do this each time because optimization is iterative and a previous call to the function will have left the state of the simulation at the end point of the simulation. The next two lines set up the model currently to the most recently computed values for the parameters we wish to fit. Line four carries out the simulation and the last two lines compute the sums of squares.

11 Stoichiometric Analysis

11.1 Stoichiometric Analysis

libRoadRunner supports a number of method to obtained structural information about the reaction network. The most common information to retrieve is the stoichiometry matrix.

11.1.1 Stoichiometry Matrix

```
import tellurium as te

rr = te.loada ('''
   var ES, S1, S2, E;

  J1: E + S1 -> ES; v;
  J2: ES -> E + S2; v;
  J3: S2 -> S1; v;
''')

print rr.getFullStoichiometryMatrix()

# Output
      J1, J2, J3
ES [[  1, -1,  0],
S1  [ -1,  0,  1],
S2  [  0,  1, -1],
```

```
E   [ -1,  1,  0]]
```

11.1.2 Conservation Matrix

To obtain the conservation matrix for a model use the model method, `getConservationMatrix`. Note that in the Antimony text we use the `var` word to predeclare the species so that we can set up the rows of the stoichiometry matrix in a certain order if we wish. This allows us to obtain conservation matrices with only positive terms.

```
import tellurium as te

rr = te.loada ('''
   var ES, S1, S2, E;

  J1: E + S1 -> ES; v;
  J2: ES -> E + S2; v;
  J3: S2 -> S1; v;
''')

print rr.getConservationMatrix()
print rr.fs()

# Output
[[ 1.  1.  1.  0.]
 [ 1.  0.  0.  1.]]
['ES', 'S1', 'S2', 'E']
```

The result given above indicates that the conservation relations, `ES + S1 + E` and `E + ES` exist in the model. As a result, Tellurium would generate two internal parameters of the form `cm` corresponding to the two relations.

12

Projects

References

[1] Hucka, M., A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J. H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le NovÃÆŠÂÂÍre, L. M. Loew, D. Lucio, P. Mendes, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. 2003. Bioinformatics **19**:524–531.

History

1. VERSION: 1.00 (Silver)

   Date: 2014-18-3

   Author(s): Herbert M. Sauro

   Title: Tutorial on Tellurium

   Modification(s): First edition, first printing