ScrumPy - Metabolic Modeling with Python

M. G. Poolman

September 22, 2006

Contents

Chapter 1

Introduction

1.1 What is ScrumPy ?

1.2 Why Python ?

1.3 About this manual

1.4 Python for the non-programmer

Chapter 2

Using ScrumPy

2.1 The Python environment

When ScrumPy is started an initial window is opened as shown in figure 2.1. The initial blue text in the windows are diagnostics reporting version information about ScrumPy's internal components (this is still a 0.9.x release !). The black text is a welcome message from python, and the red `>>>` is a prompt indicating that you may type some command. This window is in fact a minimally modified Idle shell, Idle being part of the standard Python distribution. The prompt is waiting for you to type some Python. The menus are exactly the same as Idle's so check the Idle doc to find out what they do (File/PathBrowser and File/Save are quite useful).

The main difference between the ScrumPy window and Idle is that ScrumPy has some modules (`ScrumPy, IdlePath and idleinit`) not found in Idle. All the modelling functionality is to be found within the `ScrumPy` module, the other two are private and should be ignored. All modelling functionality is accessed via the ScrumPy module.

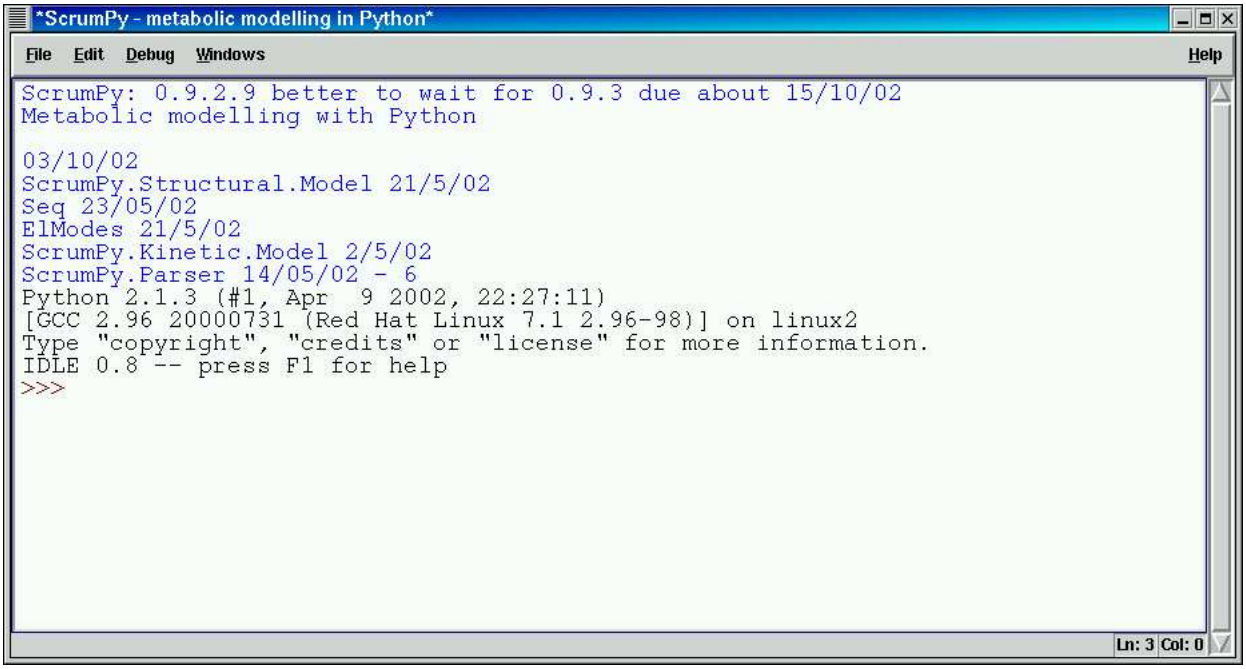


Figure 2.1: The initial ScrumPy window

Chapter 3

Defining and Generating Models

The activities of metabolic modelling, at least, those aspects that involve sitting at a computer, can be divided into three distinct areas: Model definition (i.e. specifying reactions, compounds etc.), Model interrogation (extracting useful data from the defined model), and data analysis. ScrumPy provides tools for all three of these, although the emphasis is on the first two.

Clearly we have to have some means of bringing a model into existence before it can be interrogated, and so this aspect will be addressed first. ScrumPy models are written in plain ascii files, whose syntax is described below. ScrumPy parses this file and produces a model object. Subsequent model interogation is performed by manipulating such objects.

3.1 Generating models

Models are generated with ScrumPy's `Model()` thus:

```
>>> MyModel = ScrumPy.Model()
```

On executing this the user is presented with a file request dialogue, from which to select a file containing a description of the model. An editor window (figure 3.1) containing the model description will appear. If there were errors in the input file an error message window describing the first detected error is displayed and the offending line highlighted in the editor window. Any further errors are reported in the ScrumPy window.

If you wish to create a new model, the file name to be associated with that model is passed:

```
>>> MyModel = ScrumPy.Model("MyNewModel.spy")
```

Assuming that `"MyNewModel.spy"` does not already exist, this will cause an empty editor window to open in which to enter your model description. If the file does exist, it will simply be loaded directly. ScrumPy model description files are identified by the "`.spy`" extension. This is currently compulsory.

As with the ScrumPy window, the editor window is a modified version of the idle editor window, having an extra - "ScrumPy" - menu. The other menus are identical to the idle editor windows. The compile menu items causes ScrumPy to save the file, parse the new content and, if error-free, update the Python variable associated with the model "(`MyModel`" in the example above). The actions of the other items are described later in section 4.1

There is no limit (other than those imposed by hardware) as to the number of mod-

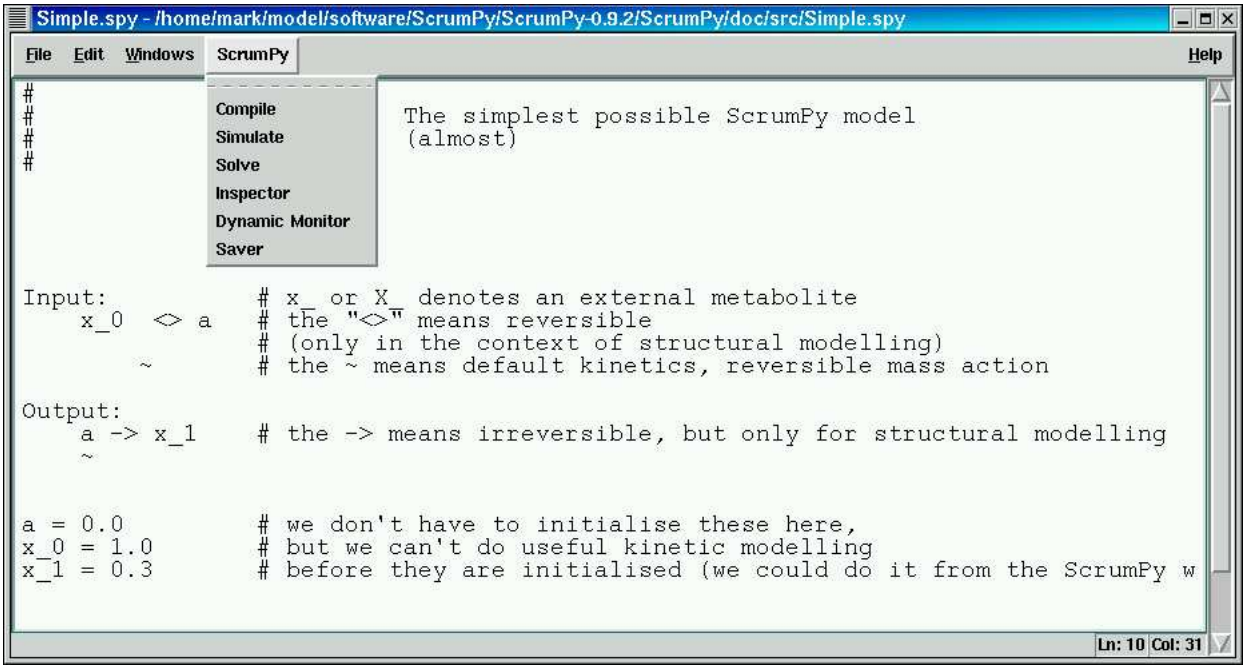


Figure 3.1: The ScrumPy model editor window, showing the contents of the ScrumPy menu.

els that you can have open at one time. However, you should not open the same model twice, as there will be cross-talk between the two, as both of the model instances contain references to the same low-level data object. e.g.:

```
>>> m = ScrumPy.Model("LinFour.spy")
>>> m2 = ScrumPy.Model("LinFour.spy")
>>> m["B"]
0.0
>>> m2["B"]
0.0
>>> m["B"] = 1.2
>>> m["B"]
1.2
>>> m2["B"]
1.2
```

If you want to deal with multiple versions of the same (i.e. loaded from identical model description files) use the `GetState` and `setstate()` methods described in section 4.2.2.

3.2 Model definition

3.3 ScrumPy model description

ScrumPy model files are plain ASCII files which should have the extension ".spy". They can contain four types of statements: comments, reactions, initialisations and directives.

3.3.1 Comments

Comments are, as you might expect, entirely ignored by ScrumPy. They exist purely to provide additional information to yourself and (other) human readers. A comment starts with a "#" (hash) sign, and extends to the end of the line. Material before the # is read in the normal way. Further #s on their same line have no effect.

3.3.2 Reactions

Reactions are the essential components of a ScrumPy model description. Syntactically, they are comprised of three components: reaction name, stoichiometry, and rate equation. These components must all be present and appear in the order described, e.g.:

```
ATPase:                    # reaction name
ATP -> ADP + Pi          # stoichiometry
Vmax_ATPase * ATP/(      # rate equation
ATP + km_ATPase)
```

Identifiers

Any named quantity in a ScrumPy model file is termed an *identifier*, and must be written according to one of the following rules:

1. **C style** Identifiers are consistent with the C programming language, they must begin with an upper or lower case letter, followed any sequence of letters, digits or "_" (underscore). No other characters (including white space) are allowed.

2. Literal Identifiers can be any sequence of characters (exluding newline) begining and ending with " (quote mark). **Important !** Literal identifiers can only be used in conjunction with the `Structural()` directive, see section 3.3.4.3.3.4.

Reaction names

Reaction names are C style or literal identifiers ending with :" (colon). i.e. if a literal identifier is given the colon comes after the terminal quote.

Reaction stoichiometries

Reaction stoichiometries consist of substrate metabolites, a reaction symbol and product metabolites. Each reaction must have more than zero substrate and product terms and exactly one reaction symbol.

Table 3.1: Operators supported in ScrumPy rate equations

Operator	Meaning	Comment
**	power	x**y == x^y
	multiplication	
/	division	
+	addition	
-	subtraction/negation	unary negation acts as you would expect, e.g. x *-y is legal
()	parenthesis	
foo(bar)	function	bar is comma separated argument of possibly zero length. More about functions below

If a reaction has more than one substrate or product, individual metabolites must be separated by the "+" (plus) sign. Metabolite names follow the same rules as reactions, except that they cannot contain a colon anywhere. Metabolites are not predeclared: from the parser's point of view, if they appear in a stoichiometry, they must be a metabolite.

Any metabolite whose first two characters are either "`x_`" or "`X_`" will be treated as an external (or fixed) metabolite, all others will be treated as internal (free or floating).

If a metabolite enters a reaction with a stoichiometric coefficient other than one, this is indicated with a single, whitespace insensitive integer, e.g.:

```
PPi -> 2 Pi
```

At present stoichiometric constants are restricted to literal non-negative integers[1] .

The reaction symbol comes in two variants: "`<>`" and "`->`", denoting reversible and irreversible reactions respectively. This reversibility applies only to the structural modelling funbctions, whether or not the reaction is kinetically reversible depends on the rate equation.

Reaction stoichiometries are entirely whitespace insensitive and can contain none at all, although doing this does little to enhance the readability of the resulting model description.

**** ? Anything else about stoichiometries ? ****

Reaction rate equations

The form of reaction rate equations should familiar. The usual operators are supported (see table 3.1), and the "BODMAS" rules of precedence are followed.

Valid operands are either literal numbers (which will be promoted to double precision internally if need be), or model values which must either be a metabolite name, a parameter name, or "`Time`". The use of reaction names in rate equations is not permitted. Like metabolites, parameters are not declared in advance. They are brought into existence by being referred to in a rate equation, if they do not appear elsewhere in a stoichiometry statement they are treated as parameters. Once the model is loaded,

[1] At present the parser allows stoichiometric coefficients of zero. I'm unsure whether to treat this as a bug or a feature.

Functions in rate equations

ScrumPy rate equations can contain invocations of any function available via the standard C header files `math.h` and `stdlib.h`. These should be treated with some caution however, as ScrumPy passes these to the C compiler without any checking. There are three possible outcomes if you make a mistake with a function invocation. They are in increasing order of seriousness:

1. The C compiler will throw it out. In this case you will get a slightly verbose error message, and the resulting model will be unusable. (see section 3.3.6)
2. The Python interpreter will crash (taking Idle, and all its open editor windows etc. with it)
3. ScrumPy will generate (unpredictably) incorrect results.

If all this looks a bit alarming, don't use functions in rate equations. They are anyway rarely, if ever, needed in realistic rate laws.

external metabolites become parameters, and their identifier is unchanged (i.e. the "`x_`" remains).

ScrumPy also provides a default kinetic for reactions: If in instead of a rate equation as described above, the reaction stoichiometry is followed by a ~ (tilde) the reaction will be assigned mass action kinetics, with rate and equilibrium constants of unity. This is mainly useful for structural modelling when kinetics are ignored.

****** ?? is that enough about rate equations ?? *******

3.3.3 Initialisation

An initialisation is simply an identifier followed by an equals (=) sign followed by an expression conforming to the same syntax as a rate equation. Initialisations can appear anywhere between rate equations, the variables being initialised do not have to have been previously referred to and are not declared. There are however, some subtle differences between the syntax of rate equations and the rhs of an initialisation, due to the fact that initialisations are handled by Python rather than C.

Firstly, it is legal to initialise values that are not a part of the model itself. These can be useful to later initialise values that are a part of the model, e.g.:

```
TotalPhos = 10.0 # not part of the model, calculation intermediate
ATP = 8.1 # a metabolite in the model
ADP = TotalPhos - ATP
```

Secondly, type matters in initialisations. Errors will be generated if the final value of a model variable is non numeric. Also, in contrast to literal numbers in rate equations, integers are distinguished from real numbers, so:

```
Keq = 1/100
```

will result in `Keq` having a value of integer 0, which will then be promoted to real 0.0 in the final model. ScrumPy uses Python to calculate initialisations, and thus uses Pythons rules for type promotion: if a calculation involves an integer and a real number, the result will be real. However it makes more sense to make everything a real number unless there is a good reason for it to be an integer.

Thirdly, the Python "math" module is available, and functions within it must be accessed using the Python `Module.Function(args)` notation.

Initialising rate values

Rate values may be initialised as described above. However this is futile, as rate values are automatically initialised from the rate equations, before the model becomes available as a Python object.

Default initialisation

If parameter or metabolite values are not initialised, default values of `NaN` (not a number) are assigned to parameters, and defaults of zero to metabolites. Under these circumstances the user will be warned as to which particular quantities are not initialised, but the model remains usable. Failing to initialise unitialised parameters before invoking simulation ore steady-state functions will cause `NaN` to propagate into all concentration and rate values.

The reasoning behind this is that it is possible to think of real situations in which a metabolite does have an initial value of zero, but for most kinetic parameters this is a meaningless value, some non-zero value must be specified. Defaulting to `NaN` for parameters mean that the error will be spotted if the model is subsequently used incorrectly, but users are not forced to initialise parameters if they don't want to. There are at least two circumstances in which this is useful: in a purely structural investigation of a model, or if the user intends to obtain parameter values from some external source before starting kinetic modelling.

3.3.4 Directives

Directives do not form part of the model description *per se*, but exist to direct ScrumPy to treat the model description in some particular fashion. Directives have the same form as a python procedure: the direcive name, follwed by a possibly empty parenthesized paramenter list. They have no return value. There a currently only two directives (but more to follow):

```
Structural()
```

Instructs ScrumPy to ignore all following kinetic information and to treat the model as purely structural. If you are not interested in the kinetics this has the advantage of being much faster, and allows you to omit all initialisations without being nagged about it. For best results, `Structural()` should be the first statement in a ScrumPy file (but this is not compulsory).

`External(mets)`

Where `mets` is a comma delimited list of metabolites, directs ScrumPy to treat those metabolites as external. There can be any number of `External()` directives and their effect is additive with the identification of externals with the `x_` specifier. Non-existant metabolites (i.e. those that do not appear in the stoichiometry of any reaction) will cause a warning to be generated when the model is compiled, but will otherwise be ignored.

`AutoExtern()`

Automatically make unbalancable metabolites (i.e. those missing a consuming or a producing reaction) external. The list thus generated will be added to metabolites defined as external by the `Exrternal()` directive and the `x_` specifier.

`ElType(type)`

Specifies the data type used for elements in the stoichiometry matrices. Currently this can be `int`, `float`, or `ArbRat`. If this directive is not used ScrumPy defaults to ArbRat. This is primarily intended for users who wish to work directly with the stoichiometry matrix. In the current version certain methods, notably `m.ElModes()`, `m.EnzSubsets()`, `m.Consmoieties()` and `m.sm.NullSpace()` will behave unpredictably if the default type is not used. There is no point in using it for small models, but use of `int or float` can considerably speed up performance of large models.

`Include(Models)`

It is sometimes useful to be able construct large models from a number of other, otherwise independent models. This functionality is supported by the `Include()` directive. Multiple includes are supported, either by passing a comma delemited list of models or by using seperate directives, i.e.:

```
Structural()
Include(Model1.spy,Model2.spy)
```

is exactly equivalent to :

```
Structural()
Include(Model1.spy)
Include(Model2.spy)
```

Nested `Include()`s are supported, so in the above example `Model1.spy` could contain `Include(Model3.spy)` and so on. Reaction names within the overall model must be unique: if a reaction name is duplicated, a warning will be generated, and the duplicate reaction ignored.

When a model using this directive is loaded, editor windows for all models will open. The whole model will be recompiled by the ScrumPy/Compile menu item in any of the windows.

The implementation of `Include()` is still fairly basic and a number of caveats and restrictions apply:

1. The mechanism only works for structural models, the top level model (i.e. the one not included by anything else) must include a `Structural()` directive. It is not mandatory for `Include()`d models to have the `Structural()` directive, but there will be a performance penalty if they do not.

2. Care must be taken to ensure that the `External()` directive is used consistently within all models. If a metabolite is declared external in one file but not another it's internal/external status is not predictable. Probably the best approach is to only use `External()` in the top level model, at least for metabolites that are common to more than one file.

3. Model files must either be specified as an absolute path, or reside within ScrumPy's current working directory.

4. Cyclic dependencies are not currently detected, there presence **WILL** cause severe and terminal problems.

5. Changing the include structure of a model on the fly may cause gui problems. If you experience this, the solution is to save (not compile) all model files, close the editor windows, and load the model from fresh.

3.3.5 Compiling models

Once one is satisfied with the model it must be compiled. This can be accomplished form the GUI or the command line. The editor window menu item "`ScrumPy/Compile`" will first save the file to disk and then process it, handling any errors as described below. If the "`Structural()` directive is not being used, two addtional files are created "MyModel.spy.c" and "MyModel.spy.so" containing the C language and binary representations of the models' kinetics. These can be deleted at any time, although this may incur a slight speed penalty, the next time the model is compiled.

Doing `MyModel.Reload()` at the idle prompt has the same effect as compiling from the menu, although this is one occaision that the GUI is probably more convenient.

3.3.6 Load errors

If an error is detected in an input file, an error window will pop up, informing you of the line number at which the *first* error was detected, some indication of the nature of the error, and the highlighting the offending text in the editor window. This message, along with any further errors will also be printed on your Python terminal. ScrumPy cannot guarentee to determine the exact position of the error, but it does guarentee that an error is present in the model description at or before the reported line. In practice the error will be at most in the reaction previous to the one in which the error is reported.

Errors that are reported without a line number, represent either a problem with the installation, or possibly a bug in ScrumPy - see sections 7.1.2 and 6.1 for further details.

Models containing errors are unusable: there are only two things that can be safely done under any cicumstances: a model can always be complied (or reloaded) as described above, and the usability of a model can be dermined at the command line with `MyModel.IsUsable()`, which returns boolean false (0) if there were errors in the input file. In practice this is only needed by hard-core hackers, not ordinary humans.

Chapter 4

Controlling and Interrogating Models

Having loaded our model, we are in a position to start doing things to it. Some actions can be performed from the ScrumPy menu in the editor window, these are useful for initial debugging of a model and some initial exploration of it's behaviour, but far more comprehensive and flexible facilities are available using the Python interface in the ScrumPy window.

4.1 The GUI tools

4.1.1 The ScrumPy menu

All GUI tools are accessed from from the ScrumPy menu in the editor window. This is a tear-off menu, so once the model is defined, it is convenient to tear the menu off, and minimise the editor window.

The first item on the menu, "Compile", is used to inform ScrumPy to take notice of the changes made in the editor window. If there are no errors in the model description, assuming you have none of the GUI tools described below open, the model will be updated internally. ScrumPy will not congratulate you on successfully writing an error-free input file: no news is good news. If there are errors present then the behaviour will be as described in section 3.3.6.

The other items on the menu all require that the model has been succesfully compiled. Selecting them if this is not the case will bring about a mild admonishment.

4.1.2 The Simulate item

Clicking the "Simulate" item in the ScrumPy menu creates a Simulation tool as shown in figure 4.1. This is as simple to use as it looks: set the desired step size and number of steps and click on the "OK" button, and the model will be simulated accordingly. The Simulation tool can be removed by closing the window, or by clicking the cancel button. This simply destroys the window, and has no effect on the state of the model. It is possible to have as many simulation tools active at once as you wish, but there is probably not a lot of benefit to be gained from doing so (prizes for the best suggestion as to why this might be useful !).

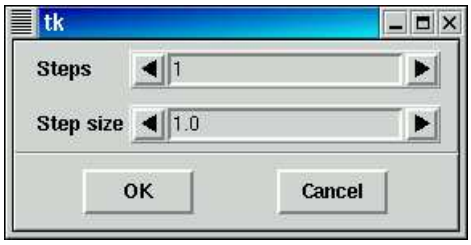


Figure 4.1: The ScrumPy Simulate tool

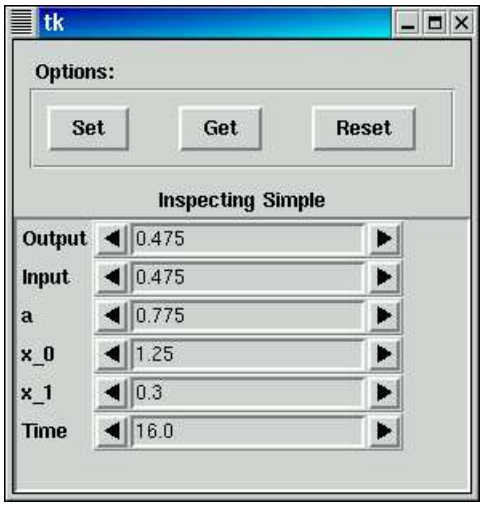


Figure 4.2: The inspector tool

4.1.3 The Solve item

Clicking the "Solve" item causes the model to (attempt to) find its steady-state. Success or failure will be indicated in a pop-up dialogue box. In our experience the steady-state algorithm (a combination of Newtons method and simulation) is fairly robust. The most likely reason for failure is either that the model does not have a static steady-state (i.e. has periodic or more complex behaviour) or there is a mistake in the model. Typical causes include stoichiometrically unbalanced reactions, or reactions with missing substrate or product sensitivity. This item simply invokes the "`FindSS()`" method as described in section 4.2.3. If the solver fails for some reason, the model will be returned to the state that it was in before a solution was attempted.

Note that when you hit the 'OK" in the simulate tool, or the Solve item, although the model will be updated internally, it is neccessary to use one or more of the tools described below in order to see anything happen.

The Inspector

The Inspector tool, as shown in figure 4.2, shows a list of all named values in the model. Values may be altered by using the arrow buttons, or typed directly in to the

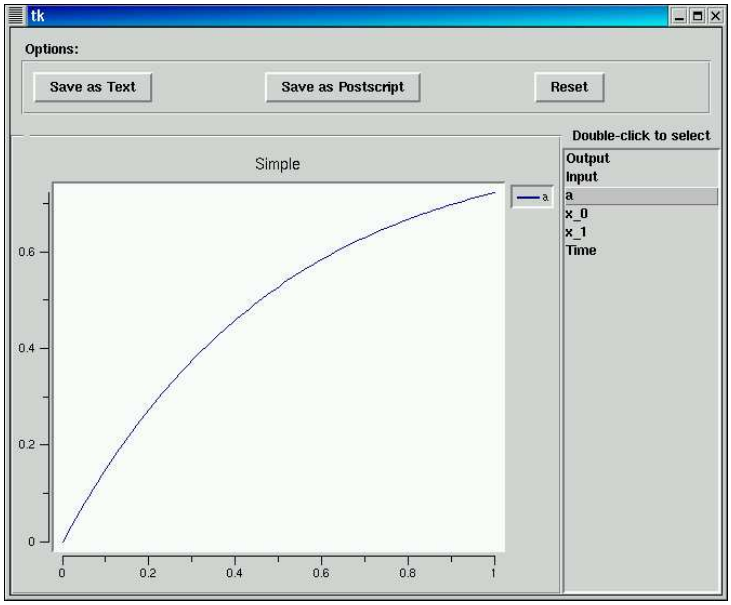


Figure 4.3: The Dynamic Monitor tool

entry field. The model is updated internally by clicking on the "Set" button. The "Get" button updates the inspector to reflect the values currently in the model. However this is not needed if working from the GUI tools, as these cause the inspector to update automatically. The "Reset" button regenerates the entire list, i.e. names as well as values. As with the 'Get" button this is not needed if working only with the GUI tools.

4.1.4 The Dynamic Monitor

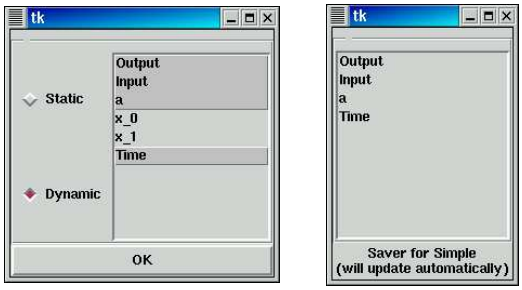
The Dynamic Monitor tool, figure 4.3 is used to plot, in real time, the results of simulations. A value from the model is selected from the list on the right hand side, and the graph updated automatically when the model is simulated. A Dynamic Monitor can only plot one named value at a time, but it is possible to repeatedly select different values to plot, and to have more than one operating simultaeneously. Dynamic Monitors have quite a high processing overhead, an perfromance suffers noticably if more than two or three are open at the same time.

Date held in a Dynamic Monitor can be saved in two ways: as text or postscript. Saving as text causes all data (including that which has scrolled off the left-hand side of the graph) held within the monitor to be saved as a plain ascii, tab delimited file.

Saving as postscript causes the visible graph to be saved as an encapsulated postscript file. The reset button destroys currently saved data (the data which is saved as text is the data held since the last reset, changing the value plotted causes a reset).

The Saver Tool

The Saver tool, figure 4.4(A) allows arbitrary data associated with the model to be saved for posterity. Multiple values are selected from the list on the right hand side, and a save mode selected from the buttons on the left hand side. If "dynamic" mode is



A **B**

Figure 4.4: The Saver tool in **A** its intial, and **B** active state

selected the saver will record data at every simulation time-point, if "static" is selected then the Saver updates at whenever a steady-state is calculated.

Once the selections are made the Saver is made active by clicking the "OK" button. This causes the Saver to prompt for a file name to write to and then redraw itself showing only the selected value names, as seen in figure 4.4(B). In this state the the Saver simply records values until either the model is recompiled or the window closed. Once this happens the user is prompted to save or discard the recorded data. Data is recorded as simple tab delimited ascii.

4.2 Using Python

Fun and exciting though the GUI tools are, as with all GUIs they are inherently inflexible and limited. Such limitations are overcome by using the Python language, as decribed below. If your experince of computing has thus far been confined to point'n'click interfaces, do not be scared. Python is such an easy language to learn that the only real danger will be trying to run before you can walk: knowing (bits of) a language does not make you a programmer. Try it, you'll love it, its a way of life.

A ScrumPy model is (an instance of) a Python object, and actions on models are performed either *via* a models *items* and *attributes*.

4.2.1 Model items

Items represent those quantities that are specific to the model in question, and are the names of reactions, metabolites, and parameters. They are referred to with an index notation, in general:

```
>>> x = MyModel["ItemName"]
```

causes the current value associated with `ItemName` to be assigned to the local Python variable `x`. Items can be assigned to as well as from; in the following:

```
>>> Glycolysis["ATP"] = 0.2 >>> NH4Assim["VmaxUptake"] *= 2
```

the first line sets the value of the item called "`ATP`" in a model called "`Glycolysis`" to 0.2, and the second line doubles the value of an item called `"VmaxUptake"` in the model called `"NH4Assim"`.

If an item name is not present in the model, attempting to read it will return `NaN` (== "not a number"). The fact that an attempt to read a value returns `NaN` does not

necessarily mean that the item does not exist: uninitialised parameters are set to `NaN`, and this value is likely to propagate into variables (i.e. concentration and rate values) if a model is used with uninitialised parameters. Attempting to assign to a non existent name will have no effect (other than a warning on stderr). If you need to determine whether or not a particular item name exists in a model (possible in a batch environment) the `Exists()` method is provided[1] :

```
>>> if m.Exists("xxx"):
print "m has xxx"
else:
print "couldn't find xxx in m"


couldn't find xxx in m
>>>
```

In addition to the Parameter, Concentration and Rate values already described, all models contain an item called "`Time`", representing the models time. This can be read or written to in exactly the manner as other items. Model items can be assigned to any numeric value, attempting to set an item to a non-numeric value will raise a python exception, leaving the model unchanged.

4.2.2 Model attributes

The attributes of a model consist of those things that any ScrumPy model is guaranteed to have. They can be further subdivided into data attributes and method attributes.

Data attributes are mainly "meta-parameters" of the model. They do not form a part of the model itself, but influence the behaviour of the low-level algorithms that act upon it. Examples include things like the number of iterations of Newtons algorithm to use before giving up, tolerances for steady-state determination and simulation, and so on. The casual user can ignore them entirely. Other, higher level and more useful, data attributes include the stoichiometry matrix described in section 4.3.3.

Method attributes provide the bulk of the modelling functionality of ScrumPy. Apart from assigning to/from individual model item values described above, all modelling activities in ScrumPy revolve around invoking method attributes. In contrast to the index syntax for item access, attribute access is accomplished with 'dot" syntax, i.e.:

```
>>> model.Method()
```

Here, "`Method`" identifies the attribute, and the parentheses indicate that it is to be invoked.

Reading or writing multiple values in a model

Sometimes it is convenient to address more than one value simultaneously. This is accomplished via the "`GetVals`" and "`SetVals`" methods:

```
>>> vals = Calvin.GetVals(["Rubisco", "RuBP_ch"])
>>> Calvin.SetVals(["Rubisco", "RuBP_ch"], [1,2])
```

In the first line we retrieve a list of values from the Calvin model, and in the second, we assign the values values to those in a literal list.

[1] Yes folks, in Idle you can do if:/else interactively !

Reading and writing vectors

A ScrumPy model contains 3 vectors (or arrays) which store the models concentration, rate, and parameters values. In order to specify which of the three we are interested in, ScrumPy provides three constants: `Conc, Vel` and `Param`, to refer to concentrations, velocities (== rates) and parameters respectively. Given this, vectors can be retrieved with the `GetVec()` method:

```
>>> ConVec = m.GetVec(ScrumPy.Conc)
```

In the above example `ConVec` is a Python list containing a copy of the concentration vector. These values will be unaffected by any subsequent action on the model, and can thus be used to hold "snap-shots" of the model to be restored at a later time.

In a similar fashion the user can set the values of a vector with the `SetVec()` method:

```
>>> m.SetVec(ScrumPy.Param, ParaVec)
```

Here the (the values of the) models parameter vector is set to the values in `ParaVec`. As with `GetVec`, this is a copy operation: changes to the model will not be reflected in the vector element values, and *vice-versa*. The vector passed in to `SetVec()` must be of the same length as the vector in the model. This can be readily verified with the `SizeVec` method, which returns the length of the specified vector, e.g:

```
>>> VecDic =
ScrumPy.Conc:"Concentration ",
ScrumPy.Vel:"Rate ",
ScrumPy.Param:"Parameter "
>>> for v in VecDic.keys():
print "Size of ", VecDic[v], "vector = ", m.SizeVec(v)


Size of Rate vector = 4
Size of Concentration vector = 3
Size of Parameter vector = 2
>>>
```

Here we first create a Python dictionary to map the ScrumPy vector identifiers to something that can be read by humans, and then scan through the dictionary to print a handy table of vector lengths. If you attempt to pass a list of incorrect length to `SetVec()`, a python exception (`RuntimeError: vector size missmatch` since you ask) will be raised. This looks ugly, but is quite safe: both the model and the input vector remain untouched.

There are other variations on the theme of getting and setting multiple values, these are `GetRescue(), SetRescue(), GetState(), SetState(), and GetStateSaver()`. They will all be documented fully real soon.

Simulating models

Time course simulation is achieved by the "`Simulate()`" method:

```
AnyModel.Simulate(TimeStep=None, NSteps=1, RetData=[], MonFuncs=[])
```

The meaning of these arguments are:

TimeStep The size of the time step. If this is not specified (or is None)the previous

Elements in vectors

The `GetVec()` and `SetVec()` were originally intended for situations in which the ordering within the vector is unimportant, however there may be cases in which the identities of individual elements in a vector is desired. To this end the `GetVecNames()` method is provided:

```
>>> print m.GetVecNames(ScrumPy.Conc)
['D', 'B', 'C']
```

The return from `GetVecNames()` is a list of strings corresponding to model variables, in the same order in which the values of those variables occur in the list returned from `GetVec()`, so

```
>>> MetNames = m.GetVecNames(ScrumPy.Conc)
>>> MetVals = m.GetVec(ScrumPy.Conc)
>>> MetVals[MetNames.index("B")]
```

Is equivalent to simply:

```
>>> m["B"]
```

value of time step is used. If no previous time step has been specified, a default value of 1 is used. Errors will result if TimeStep ≤ 0.

NSteps The number of steps (of size TimeStep) to be evaluated. Must be ≥ 1, defaults to 1, if not supplied.

RetData A list of names whose values are to be recorded at each step.These will be returned as a DataSet (described later). This set will always include time, whether or not it is specified.

MonFuncs A list of functions, taking the model as there sole argument, that will be invoked after each time step.

Any dynamic monitors that have been opened from the GUI will be updated at each time step. In addition to the returned DataSet, if the integrator ran into problems, a list of bad points and associated error messages will be returned. In practice this happens rarely, and it is generaly safe to ignore the possiblity, at least if working interactively. If you don't explicitly check that to see if the return was a DataSet or a (DataSet, []) tuple, and the latter was the case, obvious errors will occur. None the less this is not a particularly elegant error handling mechanism, and it will be replaced in the nest version.

For example:

```
>>> SimRes = Calvin.Simulate(0.01, 100, ["RuBP_ch", "Rubisco", "ATP_ch"])
>>> SimRes.SaveFile()
```

Simulates the Calvin model for 100 steps in time increments of 0.01. Values of Rubisco are plotted at each time step, and values of Rubisco, ATP, and RuBP are saved in the DataSet, "SimRes", whose contents are then saved in a file.

4.2.3 Steady-state determination

Steady-state determination is achieved by the "FindSS()" method, which uses,by default, a combination of of Newton's method and simulation. Variations on this algorithm, and attributes that affect it are described in the advanced section. For most models the default values seem adequate.

4.2.4 Numerical errors

It is important to realise that both the "`Simulate()`" and "`FindSS()`" methods request that the model *attempts* to simulate or determine a steady-state, and that this attempt might fail. It is therefore prudent, at least in a model whose behaviour is not well known, to check for such problems after executing either of these methods. There are three simple ways of doing this:

Model.IsOK() Returns 1 or 0 (Boolean True or False) according to whether or not the last method with the potential to fail, succeeded.

Model.ErrorMsg() Returns a string describing the most recent error.

Model.ReportLastError() Is the intuitive GUI version of `ErrorMsg()`, i.e. it prints the same thing in a window.

It is also important to realise that this reporting only refers to the numerical errors arising in Simulate(), FindSS(), or other functions that invoke these internally. The model must have loaded successfully before numerical errors can be checked for.

4.2.5 Metabolic Control Analysis

The coefficients of MCA are defined as the scaled first derivative of some quantity against another. Flux control coefficients are the scaled first derivative of an enzyme activity against a steady-state flux, elasticities are the scaled first derivative of a instantaneous reaction flux against a metabolite concentration, and so on. Likewise, group coefficients are defined as the sum of responses of some defined value to a group of other values.

ScrumPy provides a single method, "`ScaledSensits()`" that with the appropriate arguments, can determine all the coefficients of MCA, and some not as yet defined:

```
coeffs = Model.ScaledSensits(Params, Vars, Pert=1e-6,
Time="Inf", Points=20, Func=None)
```

these parameters have the following meaning:

Params A list of names of one or more quantities that are to be varied. They can, but do not need to be, parameters of the model.

Vars A list of names of one or more quantities whose response is to be measured.

Pert The relative size of the perturbation to be made to Params to determine the sensitivity. This defaults to a value of 10^{-6} and can be safely left alone in most cases.

Time The time over which the sensitivity is to be determined. If this is equal to zero, then the instantaneous sensitivity (i.e. elasticity if Params is a concentration, and Vars reaction rates) is determined. If time is finite then the sensitivity of Vars

More about numerical errors

In a batch environment strings describing the errors are not a great deal of use, especially if you intend to take some kind of corrective action. To this end the method `model.LastErr()` returns an integer describing the most recent problem, as described below.

`m.LastErr()`	`m.ErrorMsg()`	Meaning
0	"No Error"	No error
1	"Negative concentration"	-ve concentration(s) encountered during simulation or steady state determination.
2	"Requested tolerance not achieved"	Unable to determine steady state to tolerance requested in the model (default is 10^{-6}).
3	"Solver can't get steady state with these start conditions"	More serious than above, bad choice of initial metabolite values, problem with model definition, or possibly (eek !) a bug.
4	"Simulation can't proceed from this position"	Problems in the time course simulator. Most likely reason is unitialised parameters or a very large or stiff model.
5	"Out of memory"	If you see this I'll buy you a beer.
6	"Couldn't open file"	Not currently used, if you see this it's a bug
7	"No more data"	See 6
8	"Model not defined"	No attempt has been made to load the model
< 0 or > 8	!! CRASH !!	You have found a bug

to Params after that length of time is determined. In this case Points simulation steps are taken to reach Time. If Time is equal to "Inf", the default, steady-state sensitivity is determined.

Func If supplied must be a function that takes a model as its input and returns double as it's output. The output from ScaledSensits will then be the scaled sensitivity of Func(Model) to Params.

`ScaledSensits()` returns a list of scaled sensitivities of `Vars` to `Params`, for example:

```
>>> Calvin.ScaledSensits(["Rbco_vm"], ["RuBP_ch", "PGA_ch", "Rubisco"])
[-1.29305545895, 0.0259370244547, 0.00128151446349]
>>>
```

returns the concentration control coefficient of Rubisco activity on its' substrate and product, and the flux control coefficient on its' own reaction rate.

4.2.6 Evolution Strategy for Model Fitting/Optimisation

Model fitting and optimisation is attained with (instances of) the `ScrumPy.FitPop` class, instantiated thus:

```
pop = ScrumPy.FitPop(m, "data", nParents, nOffspring, GNames, len(GNames),
Lambda, MaxRel)
```

where arguments have the following meanings:

m is a valid ScrumPy model

data is the name of data file, whose format is described below

nParents is the number of individuals who go on to reproduce in the next generation

nOffspring ¿nParents, the total number of Offspring produced in the next generation

GNames a list of names of model items that will be mutated to fit the data

Lambda (not lambda !!) the size of relative mutation at each reproduction MaxRel
The relative limit values beyond gene values are not allowed to mutate

The data file is white-space delimited ascii, the first row is a list of model item names, the first item must be time (i.e. at present we can only do time course fitting.) Time intervals do not need to stat at zero and do not need to be equally spaced.

In general the items in the data file will be model variables, but this is not mandatory. The names in GNames will be parameter names. If the model contains conservation cycles it is reccomended that you remove the conservation parameters, called "`CONS_nSUM`" where '`n`' is a non-negative integer, from the list. If only some parameters are unkown, also remove the names of known values from GNames.

As a rule of thumb the number of parents should be made equal to the number of genes, and the number of offspring five times greater. A good default value for Lambda is about 0.05.

Having set up the population we need to let it evolve:

```
for n in range(40):
print pop[0][0]
pop.DoGenerations(10)
```

Here we do $40 \times 10 = 400$ generations, printing a helpful message every 10. Populations have list-like characteristics, essentially they are a list of organisms sorted (best first) in order of fitness. Organisms are a 2-tuple of fitness and genome, so our message is simply the value of the most fit individual in the population. Choosing optimal values of Lambda and MaxRel is a bit of a black art, and some interactive juggling may be required to obtain the best fit.

4.3 Structural analysis of models

4.3.1 Moiety conservations

Conservation relationships are determined with the `ConsMoieties` method which returns a ScrumPy dynamic matrix (see section 5.3):

```
>>> conmo = m.ConsMoieties()
```

```
>>> print conmo
r/c RuBP ADP BPGA GAP DHAP FBP F6P G6P PGA G1P X5P SBP S7P R5P Ru5P
E4P ATP Pi
ATP [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
Pi [2, -1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1]
```

4.3.2 Enzyme subsets

Enzyme subsets are obtained thus:
`>>> ess = m.EnzSubsets()`
Where `ess` is an instance of "`EnzSubset`", a class with dictionary/simple data-base functionality. Each susbset held in this database has assigned to it, amongst other things, a 'State' which can take one of four values:

Dead At steady-state the reactions in this subset carry no flux.

Irrevesible The subset can only carry flux in one direction.

Reversible As you'd expect, flux can be carried in either direction.

Empty Only used in during calculation. It's a bug if you see this.

Each subset held within an "`EnzSubset`" instance can be passed to the standard python "`len()`" and `str()`. The former gives you the number of reactions in that subset, and the latter a string representing the list of reaction names with their weighting factor, and the state of that subset. Each subset can be accesed in a list-like fashion, each itm in the list being a tuple of the reaction name and a rational weighting of the enzyme within the the subset.

The "`EnzSubset`" class can be interrogated in a number of ways:

With the Python built-in `len()` and `str()` functions:

`len()` Returns the number of subsets.

`str()` Concatenates the `str()` function for the individual subsets, seperating them with a '`\n`'.

With the EnzSubset class methods:

`keys()` Returns a list of keys of individual subsets.

`DeadSSs()` Returns a list of length 0, or 1 containing the key of Dead the subset.

`IrevSSs()` **and** `RevSSs()` Return possibly empty lists of keys of irreversible and reversible subsets respectively.

ToMatrix(ColOrder) Returns the enzyme subsets in as a matrix, each subset is represented by a row, each reaction by a column, elements are the contribution made by an enzyme to its' corresponding subset. The "`ColOrder` paramenter is optional. If given, it must be a list of enzyme names in the system, each name present exactly once. This then specifies the column order of the matrix, otherwise the column order is arbitrary.

Example

```
>>> ss = m.EnzSubsets()   # get the subsets
>>> len(ss)               # how many ?
21
>>> print len(ss.DeadSSs()), len(ss.IrrevSSs()), len(ss.RevSSs())
    #how many dead, ireversible and reversible ?
0 11 10
>>> revs = ss.RevSSs()      # get the keys of reversible subsets
>>> for r in revs:          # print a nicely formatted list
    if len(ss[r]) >1:       # of reversible subsets with more
        print r, " :\t",    # than one reaction
        for e in ss[r]:
            print e[0], "\t",  # print the name only
        print


SubSet9 :    PFP     UGPase    NDPK
SubSet1 :    Pexp    AT
```

4.3.3 The stoichiometry matrix

The stoichiometry matrix of a model is held in a data attribute, `sm`, and although it can be accessed directly, it is much safer to work with a copy of it:
`>>> sm = m.sm.Copy()`

This returns an instance of a stoichiometry matrix as defined in the module `ScrumPy.Structural.StoMat`. This is a sub-class of the dynamic matrix class described below.

The condensed stoichiometry matrix

The condensed stoichiometry matrix of a model (i.e. the stoichiometry matrix in terms of enzyme subsets rather than individual reactions) is determined thus:
`>>> csm,ess = m.CondensedSM()`
which generates the tuple `(csm,ess)`, where `csm` is the condensed stoichiometry matrix as an instance of `ScrumPy.Structural.StoMat.StoMat`, and `ess` is an instance of EnzSubset as described above.

4.3.4 Elementary modes

The elementary modes of a model are obtained with the `ElModes` method:
`modes = m.ElModes()`
The data oject returned is an instance of the `ModesDB` class, implementing simple database functionality for the set of calculated elementary modes. To a large extent, elementary modes analysis becomes an exercise in interrogating this data-base.

Number of elementary modes

The number of elementary modes in a model can be determined by the standard Python `len` function:

```
>>> mo = m.ElModes()
>>> len(mo)
42
>>>
```

Viewing elementary modes

The `Modes()` method of the `ModesDB` class returns a handy Python string representation of the elementary modes in terms of reactions and their coefficients. The string contains new-line characters, so use the Python print statement for optimal viewing satisfaction:

```
>>> print mo.Modes()
-1 PGM -1 PGI -1 FBPase -1 Ald1 1 StPase 2 TPT_DHAP 1 TPI
.
.
.
1 StPase 1 LReac 1 StSynth
>>>
```

Viewing elementary mode stoichiometries

The overall stoichiometries, in terms of external metabolites, of elementary modes can be viewed in the same way as elementary modes, using the `Stos()` method[2] :

```
>>> print mo.Stos()
-2 x_Pi_cyt -1 x_Starch_ch     2 x_DHAP_cyt
.
.
.
-12 x_Proton_ch -12 x_NADPH_ch -6 x_CO2          1 x_Starch_ch 12 x_NADP_ch

>>>
```

Negative coefficients represent consumption of a metabolite and positive, production. It is possible that empty lines will appear in the output from the `Stos()` output. Such lines correspond to elementary modes that niether consume nor produce external metabolites: futile cycles.

More information about futile cycles can be obtained with the `Futile()` method. This returns a new `ModesDB` object containing only the futile cycles from from the original:

```
>>> fut = mo.Futile()
>>> len(fut)
```

[2]This method only works with models loaded from ScrumPy files, not SCAMP files.

Table 4.1: Filtering methods of the ModesDB class

Method	Behaviour
PosFlux(n)	modes for which reaction n carries a positive flux
NoFlux(n)	modes for which n carries no flux
NegFlux(n)	modes for which n carries a negative flux
Consumes(m)	modes which consume metabolite m
Unused(m)	modes which have no net effect on m
Produces(m)	modes which produce m

```
1
>>> print fut.Modes()
1 StPase 1 LReac 1 StSynth

>>>
```

Filtering elementary modes

It is often the case that calculating the elementary modes of a system yields an embarrassment of riches, and some degree of filtering is required to classify modes according to criteria determined by the problem under consideration. In additiopn to the special case of futile cycles, there are six such methods summarised in table 4.1. All of these return new instances of `ModesDB` and such queries can be chained together. All of the methods in table 4.1 can take an optional, named, argument, `neg`, the default value of which is zero. If set to any non-zero value the search criteria is negated.

```
>>> assims = mo.Consumes("x_CO2") # modes that consume external CO2
>>> stor = mo.Produces("x_Starch_ch") # modes producing starch
>>> dark = mo.PosFlux("LReac", 1) # modes that do not have positive
flux in LReac
>>> surprise = mo.PosFlux("Rubisco",1).Consumes("x_CO2")
>>> len(surprise)
0
>>>
```

The fourth line in this example chains together two queries, the first extracting those elemntary modes that do not have a positive Rubisco flux, the second extracting from that set those elementary modes that consume external CO_2. These are assigned to `surprise` as Rubisco is the only reaction capable of consuming CO_2 in the model in question. Happily we see that the length of `surprise` is zero.

Chapter 5

Utility modules and classes

5.1 DataSets

The DataSets module provides a single class, DataSet, whose purpose in life is to provide convenient data handling. From the modellers' perspective it has two particularly useful attributes: it can read and write data in a sensible ASCII format, and it can interface with the gnuplot plotting program. Taking an earlier example:

```
>>> SimRes = Calvin.Simulate(0.01, 100, ["RuBP_ch", "Rubisco", "ATP_ch"],
"Rubisco")
>>> SimRes.SaveFile()
```

We can plot the data we have retrieved:

```
>>> Plotter = SimRes.ScatterPlot("Time", "Rubisco") # plot the data
and create a plotter
>>> Plotter("set term post") # put the plotter in postscript mode
>>> Plotter("set out 'PlotRes.ps'") # direct future plotter output
to the file 'PlotRes.ps'
>>> Plotter.replot() # send output to the file
```

DatSets can do a whole lot of other things which will be described here in the fullness of time.

5.2 Plotter

5.3 Dynamic matrices

The structural components of ScrumPy all rely heavily on the dynamic matrix class, and some knowledge of this class will be needed if you wish to do go beyond the functions outlined in 4.3

Dynamic matrices are defined in the DynMat module and can be loaded from Python thus:

```
>>> from ScrumPy.Util import DynMatrix
```

. The module can be used independently from the rest of ScrumPy, although it does depend on some of the other modules in the ScrumP/Util directory.

subsubsectionCreating dynamic matrices New dynamic matrices are created via the matrix constructor in DynMatrix:

```
>>> mtx = DynMatrix.matrix(nrows, ncols, Conv)
```

Where nrows and ncols are the numbers of rows and columns respectively, and Conv if a function that converts a single argument into the type of the callers choosing. The matrix will subsequently contain elements of this type. The nrows and ncols arguments default to 0 (zero) and Conv defaults to an arbitrary rational type1 , which has the advantage of being immune from round-off or overflow error.

Dynamic matrices can also be created by copying existing matrices, for example, to obtain the stoichiometry matrix from model, m:

```
>>> sm = m.sm.Copy()
```

gives you a local copy of the stoichiometry matrix. Note that simply assigning sm:

```
>>> sm = m.sm
```

just creates a local reference to the model's stoichiometry matrix and changing the local version will change the actual stoichiometry matrix in the model. This is almost certainly a bad idea.

Copy() takes an optional argument of Conv as with the constructor this defines the element type of the new matrix, and defaults to arbitrary rational, so:

```
>>> sm = m.sm.Copy(float)
```

gives you a floating point representation of sm.

Row and column names

The rows and columns of a dynamic matrix (DynMat hereafter) have names associated with them. By default these are simply "row0"..."rowN" and "col0"..."colN". Names exist as lists of strings (in rnames and cnames) and can be accessed there, but such direct access is strongly disrecomended. The sensible approach is to supply names when creating new rows and columns to a matrix using the NewRow/Col() methods:

```
>>> colnames = ["Mary", "had", "a", "little", "lamb"]
>>> rownames = ["She", "thought", "it", "rather", "silly"]
>>> m = DynMatrix.matrix()
>>> for name in colnames:
m.NewCol(name=name)


>>> for name in rownames:
m.NewRow(name=name)


>>> m

[Mary had a little lamb] She[0, 0, 0, 0, 0]
thought[0, 0, 0, 0, 0]
it[0, 0, 0, 0, 0]
```

```
rather[0, 0, 0, 0, 0]
silly[0, 0, 0, 0, 0]
```

These methods also allow you to specify an optional list of initial values. However this is slightly risky as there is no checking to ensure that these are consistent with the existing matrix. In general elementwise access is to recomended.

5.3.1 accessing elements and rows

By integer index

The traditional array notation applies:

```
>>> el = m[i][j]
>>> m[i][j] = value
```

retrieving and assigning the value of element (i,j) respectively, where i and j are integers. This method of assigning values is not recomended, as the type of value will not be converted to that of the other elements. A better way of achieving the same thing is having both the row and column indices in the same bracket:

```
>>> el = m[i,j]
>>> m[i,j] = value
```

This ensures that value is converted to the correct type (or else an exception is raised).

Using a single index returns a reference to a row:

```
>>> r = m[idx]
```

Modifying it modifies the matrix itself.

Indexing by name

Row and column names may be used as indices in the same general manner as integers, names and integer indices may be mixed:

```
>>> m = DynMatrix.matrix()
>>> for c in ["John", "James", "Jerry"]:
m.NewCol(name=c)


>>> for r in ["Jenny", "Jean", "Joan"]:
m.NewRow(name=r)


>>> m["Jenny", "John"] = 1
>>> m["Jean", 1] = 2
>>> m[2, "Jerry"] = 3
>>> m

Jenny[1, 0, 0]
```

Table 5.1: Row and column access to dynamic matrices.

Action	Row	Col	Note
Get	GetRow(idx)	SetCol(idx)	idx is string or integer
Set	SetRow(idx, seq)	SetCol(idx,seq)	sequence is of consistent length

```
Jean[0, 2, 0]
Joan[0, 0, 3]
>>>
```

the only exception is that the form:

```
>>> m["hello"]["sailor"]
>>> # or
>>> m[integer][string]
```

won't work, because the first index returns a reference to a row, which is just an ordinary Python list.

5.3.2 Row and column access

The methods for acessing rows and columns within a DynMat are intuitively straightforward an summarised in table 5.1 . The only pitfall is that when setting a row or column it is the user's (i.e. your) responsibility to make sure that the sequence is of the correct length, or subsequent behaviour will be unpredictable.

Chapter 6

Contribute

6.1 Bug Reporting

6.1.1 Bug definition

Bugs are when the features decribed in this doccument do not work as described, or produce an pbvious numerical error. The reason for this caveat is that Python has no enforcable encapsulation, and a lot of the ScrumPy modules contain functions which directly access C/C++ structures and will cause any amount of havoc.

Likewise various data members of modules and classes should be treated as private or at least read only, changing them will certainly cause ptoblems.

If you find a bug, please make sure it's not listed below, and send the following:

1. The ScrumPy model description
2. The shortest Python script that (re)produces it.

6.1.2 Known bugs

Deprecation warning on Import If you have the stats package installed, import ScrumPy will generate an ugly red deprecation warning. This is harmless, and will go when depenendence on the stats package is removed (planned RSN).

The model editor starting the model editor, i.e. during m = ScrumPy.Model(), or m.Edit() if the window has been closed, generates an ugly red traceback. It is safe to ignore and will vannish in the fulness of time.

Time dependent functions Time dependent functions are not being calculated correctly, and at present should not be used.

Chapter 7

Installation

7.1 Prerequisites

7.1.1 OS

This version has been tested and developed under x86 Linux (Red Hat 7.1) and Solaris. As long as the tools and packages described below are in place, ScrumPy should install on any Unix-like OS. If you are interested in porting ScrumPy to other platforms, please contact the author.

7.1.2 Python

At present ScrumPy is only known to work with Python2.1. We haven't tried it with 2.0 or 2.2 and welcome any field reports.

ScrumPy can use a number of (too many ?) other modules, listed in table 7.1. The Pmw and BLT packages are used for the simulation monitor. If you do not have the monitor window remains empty, but simulations proceed as described. The other optional modules are used by as-yet undoccumented functions - everything described here will work without them.

7.1.3 Non Python tools

In order to install and run ScrumPy, a number of other packages are needed. They are all readily available. They are all to be found as part of the Red Hat (7.1) distro, but you don't have them, follow the URLs below.

gcc and Gnu make

These are part of any Linux distro. See http://gcc.gnu.org for versions for other OSs. Non Gnu versions of make might work, but it's not guarenteed.

Swig

Swig is used to generate the interface betweeen the Python and C/C++ layers of ScrumPy. It is to be found in Red Hat 7.1, if you don't have it the URL is http://www.swig.org. ScrumPy requires the older, stable 1.1p5 version, the 1.3 versions do not work.

Table 7.1: ScrumPy prerequisites. Most of these should come with you Linux distro, so check your CDs before following these URLs. Also note that these URLs are for the modules homepages. If you are running an RPM based Linux have a try http://www.rpmfind.org .

Module	Provides	Optional	Download
Numeric	Numerical functions	No	http://sourceforge.net/ projects/numpy
Ply	Pasrer functions	No (but distributed with ScrumPy)	http://systems.cs.uchicago.edu/ply/
Gnuplot	Interface to Gnuplot plotter	Yes	http://monsoon.harvard.edu/ mhagger/Gnuplot/Gnuplot.html
Gnuplot	The Gnuplot program	Yes	http://www.gnuplot.org
stats	statistical functions for DataSets	Yes	http://www.nmr.mgh.harvard.edu/ Neural_Systems_Group/gary/python/ statstest.zip
Scientific	Non-linear least squares in DataSets	Yes	http://starship.python.net/ crew/hinsen/scientific.html
Pmw	Monitor plotter	Yes	http://pmw.sourceforge.net/
Blt	Graph function for Pmw	yes	ftp://ftp.tcltk.com/pub/blt/ BLT2.4.tar.gz

f2c

Some of the low-level numerical code started life as fortran, and hence requires the f2c libraries. f2c is to be found in Red Hat 7.1 or at ftp://netlib.bell-labs.com/netlib/f2c/ or http://www.netlib.org/f2c/readme .

7.2 Installing

7.2.1 Instant gratification

Unpack the .tar.gz file, cd into the directory thus produduced, and as root, type
`# python2.1 install.py`

This will build the entire distribution in a sub-directory (hereafter refered to as ScrumPyHome), `ScrumPy` of Python's `site-packages` directory, typically, `/usr/lib/python2.1/site-packages`

7.2.2 Customised

If for some reason you do not wish to install into the default location, for instance because you do not have root access on your machine, the `setup.py` script can take a single argument: the target path to which you wish ScrumPy to be installed. This target path must not be within the installation directory. Also, if you do this you will need to inform Python where to find ScrumPy, either by modifying the `PYTHONPATH` environment variable, or by editing Python's `sys.path`.

7.2.3 Invocation

Once installed, ScrumPy is started by by running the trivial `ScrumPy` script, found in `ScrumPyHome/bin`. This script assumes the default location for ScrumPyHome, and will need editing if a customised installation was performed.

When run, this will start Idle (the Python integrated development environment) with ScrumPy pre-loaded, and some other minor modifications. In principle it is possible to simply import ScrumPy as a module into an existing Idle session, or script file. However, ScrunPy's GUI components may behave unpredictably in these circumstances, and such modes of use are not suported as of release 0.9.2.9. If you *really* need to use ScrumPy in standalone scripts (perhaps to run batch jobs on a remote machine), e-mail me for advice.

7.3 Copyright and Licence

7.3.1 Exceptions

The files in the directories Kinetic/clib/scampi/minpack and Kinetic/clib/scampi/miscfort were taken from the GNU Octave source code, available from http://www.octave.org, the file Kinetic/clib/scampi/hybrdl/hybrdl.c is a C translation via f2c of a fortran file of the same name, also from Gnu octave, and the files in Kinetic/clib/scampi/lsoda, unless otherwise indicated, are from Alan Hindmarsh's LSODA and are included with his kind permission.