SigMat: A highly configurable MATLAB toolbox for modelling signalling networks based on matrix manipulations.

Martin Wong

September 14, 2016

Licence and Copyright Notice

Installation and Preamble

This manual outlines the steps required to use the SigMat package.

To use this package, either download the package, unzip the contents into any folder. Alternately you can create a fork of the repository (following this guide).

After doing this, add the root folder to the MATLAB search path. This can be done temporarily using the `addpath(genpath(pathname))` command (the command will add all subfolders within `pathname`). This will need to be done with each startup of the program. Alternately the package can be permenantly installed by finding the "Set Path" option in the MATLAB toolbar/menu and then adding `pathname` using the "add with subfolders" option. Only advanced users should considering doing this though.

There are a number of components to the documentation. The basic component involves simply using the program to simulate and perform parameter fitting on models.

For users interested in only the basic features, the out of the box SigMat model supports the following rate equations:

- Zeroth, first and second order mass action
- Hill functions
- dQSSA (prefactor method of enzyme kinetics)

and utilises Markov Chain Monte Carlo with simulated annealing to perform parameter analysis (though sigMat is set up so it is compatible wit other optimisation methods). This will be discussed in more detail in chapter 2.

The advanced component involves customising the various internal components of sigMat such as the reaction classification system, reaction definition system, matrix construction system and dynamic equation construction system. This will require a deeper understanding of the software architecture. The architecture and instructions for customising the package will be provided in chapter 3.

A quick start guide is included in chapter 1 to provide a hands on overview of the main basic components of the SigMat package.

1 Quick Start Guide

This section will describe a short workflow for creating a SigMat model and then running it. The following example is designed to be run through in succession. Please ensure it is all done in the one sitting. This tutorial should take about one hour to complete and will cover most of the basic features of the SigMat package. All commands beginning with `>>` need to be entered into the MATLAB terminal (you will see a similar symbol when you open MATLAB). Lines that looks like `##|`, where `##` are numbers, need to be entered into the editor and the file subsequently saved.

1.1 Installation

```
>> addpath(genpath('.\sigMat'))
```

`'.\sigMat'` is the path to the SigMat package. This may change depending on your installation. This must be surrounded by inverted commas (as it is a string).

1.2 Model Creation

```
>> mkModel('modelFile')
```

`'modelFile'` is the name of the SigMat model to be create. This must be surrounded by inverted commas (as it is a string, not a number). A window should now be opened showing the newly generated model file.

1.3 Simulation

1.3.1 Basics

If the template file has not been modified, the following can be run.

```
>> [t_1,Y_Dis_1] = findTC('modelFile',[0 100])
```

This runs the simulation between the times $t = [0, 1]$. `Y_Dis` is the complex dissociated concentrations for the time points given in `t`. Each column corresponds with the rows found in the modSpc components of the `'modelFile'` model. You can see this by noting that the output for `Y_Dis_1` is:

```
Y_Dis =

    1.0000    2.0000    3.0000    4.0000    5.0000
    1.0000    2.0000    3.0000    4.0000    5.0000
    1.0000    2.0000    3.0000    4.0000    5.0000
```

Which corresponds with the concentrations given in the model (the left numbers give the line number of the model file):

```
94| modSpc = 'mAKT'    ,'PM'   , 1;
95|          'AKT'     ,'Cyto', 2;
96|          'p473mAKT','PM'   , 3;
97|          'p473AKT' ,'Cyto', 4;
98|          'mTORC2'  ,'PM'   , 5;
```

Note that there is no change to any of the numbers between rows. This is because the system currently has no programmed reactions.

We can enable a reaction by changing:

```
162| rxn(end).k   = 0;
```

to

```
162|rxn(end).k   = 0.1;
```

Save the file (do this after every change that is described in the section). Next, enter and run the findTC command again. You will notice the `Y_Dis_1` output change, but all rows are still the same. This is because `findTC` finds the basal (zero input) response automatically. This can be disabled by adding the `'-b'` flag. So the command becomes:

```
>> [t_2,Y_Dis_2] = findTC('modelFile',[0 100],'-b')
```

The Y_Dis_2 output will now change between rows ending with:

```
0.0002    2.0000    3.9998    4.0000    5.0000
```

Each row corresponds to the response of each time point in the t column.

1.3.2 Perturbations

To introduce and study perturbations on the system, the 'inp' flag can be used. For example if we want to study the same reaction as before, but consider the introduction of the enzyme "mTORC2" as a perturbation rather than an initial condition, we can first change

```
98| 'mTORC2','PM', 5};
```

to

```
98| 'mTORC2','PM', 0};
```

Then use the command:

```
>> [t_3,Y_Dis_3] = findTC('modelFile',[0 100],'inp','mTORC2',5)
```

Note the lack of the '-b' flag which implies basaling is enabled. The 'inp' flag indicates a perturbation is introduced. In this case 'mTORC2' is introduced with a concentration of 5. This reproduces Y_Dis_2, the basal disabled result of the unedited model.

1.3.3 Free parameters

Free parameters can be defined using NaN in place of parameter values in the model file. Parameters can either be concentrations, compartment sizes, reaction rates or Michaelis Constants. For example, if we replace

```
163| rxn(end).k = 0.1;
```

with

```
163| rxn(end).k = NaN;
```

Then using the command:

```
>> [t_4,Y_Dis_4] = findTC('modelFile',[0 100],'p',0.1,'inp','mTORC2',5)
```

will achieve the same result. However, running

```
>> [t_5,Y_Dis_5] = findTC('modelFile',[0 100],'p',1,'inp','mTORC2',5)
```

Will speed up the simulation as the rate constant is now increased from 0.1 to 1. Ending at

```
0.0000    2.0000    4.0000    4.0000    5.0000
```

Parameters can also be grouped. If we replace

```
94| 'mAKT','PM'  , 1;
95| 'AKT' ,'Cyto', 0;
```

with

```
94| 'mAKT','PM'  , [NaN 1];
95| 'AKT' ,'Cyto', [2   1];
```

and save the file, this now implies that `mAKT` is always initially half as abundant as AKT. We can run the simulation:

```
>> [t_6,Y_Dis_6] = findTC('modelFile',[0 100],'p',[1 0.1],'inp','mTORC2',5)
```

To achieve an output that's the same as result 4 (noting the first parameter is now the concentration of mAKT while the second is the rate constant). If we run:

```
>> [t_7,Y_Dis_7] = findTC('modelFile',[0 100],'p',[2 0.1],'inp','mTORC2',5)
```

From the result, the initial concentrations should be (note that the second column, the Akt column, is twice as large as the first mAKT column):

```
2.0000    4.0000    3.0000    4.0000         0
```

With the final time point being:

```
0.0005    4.0000    4.9995    4.0000    5.0000
```

1.4 Parameter Fitting

1.4.1 Basic Usage of MCMC

Enter the following command into MATLAB:

```
>> objfun = @(p) p^2;
```

This is a simply objective function (goodness of fit verses parameter value) based on a quadratic function. This can then be optimised using the following command:

```
>> [pts,logP] = MCMC(objfun,3,[-10 10])
```

In the case of the inputs (on the left), the `3` denotes the seed point the algorithm will start searching from. This has to have as many values as free parameters. The `[-10 10]` gives the search boundary. The first column is the lower bound and the second column is the upper bound. Each row is the boundary for a separate free parameter. In the case of the outputs (on the right), `pts` gives the list of found parameter sets and `logP` gives the corresponding objective value (goodness of fit value) for each point. The density of points given by the list of points is also a reflection of the goodness of fit within a particular parameter space. Plotting the density in a histogram using:

```
hist(pts)
```

Should produce a figure that is similar to Fig. 1.1.

1.4.2 Basic Usage of MCMC

An objective function that incorporates `findTC` and data will be more complex. An very simplified example will be provided here. First, go to `'modelFile'` and change:

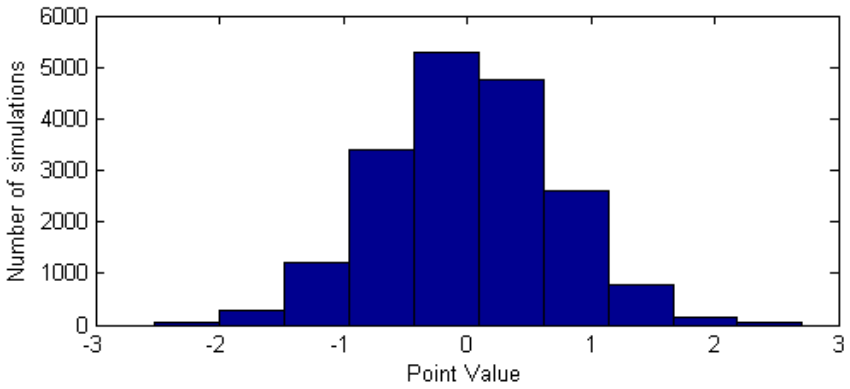

Figure 1.1: Fitting of a simple quadratic function.

```
170| rxn(end).k   = 0;
```

to

```
170| rxn(end).k   = 1;
```

Click on File → New → Script, or click "new script" on the upper left of the screen. Copy all of the below into the window that opens.

```
function resid = obj(p)

% Input Data
tDat  = [0 0.5 1 1.5 2 2.5 3];
yExp  = [0 2.44 3.08 3.25 3.30 3.31 3.31]';
sdExp = [0.15 0.3 0.6 0.75 0.96 1.2 1.05]';

% Simulated Data
[t,Y_Dis] = findTC('modelFile',tDat,'p',p,'inp','mTORC2',5);
ySim = Y_Dis(:,3);

% Normalise between experimental and simulated data
ySim = ySim.*median(yExp)/median(ySim);

% Calculate objective value
resid = sum( ( (ySim-yExp)./sdExp ).^2 )/2;
```

Now run the following commands to perform fitting of normalised data.

```
>> opts = MCMCOptimset('ptNo',2000);
>> [pts1,logP1] = MCMC(@obj,[],[1 5;1e-1 1e2]opts)
```

Go away and make a coffee. This run will take approximately 10 minutes to finish. After that is done, go to `obj.m` and change line 13 from:

```
ySim = ySim.*median(yExp)/median(ySim);
```

to

```
%ySim = ySim.*median(yExp)/median(ySim);
```

Save this file as `obj.m`. This will now fit absolute data. Run:

```
>> [pts2,logP2] = MCMC(@obj,[],[1 5;1e-1 1e2],opts)
```

This run will approximately take another 20 minutes. To analyse the result, run:

```
>> subplot(1,2,1);
>> hist3c([pts1(:,1) log10(pts1(:,2))],[25 25]);
>> subplot(1,2,2);
>> hist3c([pts2(:,1) log10(pts2(:,2))],[25 25]);
```

The resulting figure should look something like the figures on the right of Fig. 1.2. The left figures are the theoretical plots of the objective function landscape within the parameter space where the colour shows the goodness of fit against the parameter values (red is a perfect fit while blue is a poor fit):

You can see here that the MCMC result has picked a small subsection of the parameter space and reproduced the objective landscape. It can be seen from the theoretical plot though that there is a great deal of inherent variability in the parameter values. This variability is replicated by the algorithm. On the other hand, the algorithm is not focused on seeking out the true value. The true parameter value is actually:

```
[mAKT] = 2
k  = 2.3
```

There is a chance that the MCMC output does not overlap with this region.

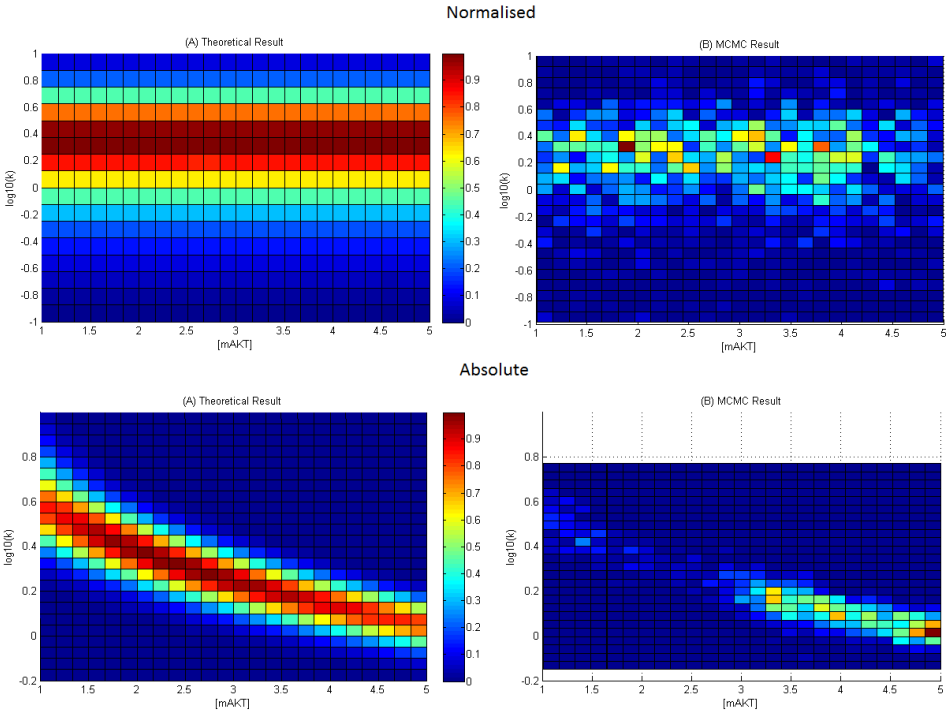

Figure 1.2: Toy model parameter fit. The top figures show the normalised result. The bottom figure shows the unnormalised result. The two left figures show the theoretical result, while the right two shows an example of the MCMC result.

2 Basic Features

The basic components of the sigMat algorithm consists of a number of features that provides the basics for completing a study of siganlling pathway models. The basic workflow for this type of project are general:

1. Model construction and defintion in an unambigious way
2. Simulation of the model to quantitatively determine its kinetics
3. Optimisation of the parameters to tune the model to observed data based on a custom defined relationship.

These correspond with the three fundamental sigMat features, which are:

1. The sigMat model format: lists the details of a model in a human readable format.
2. the findTC (find time course) simulator: takes a sigMat model, and in conjunction with various options, returns the quantitative kinetic behaviour
3. the MCMC optimiser: takes a custom defined objective function and through a stochastic method, saves as many well fitting points as defined in the program options

The general workflow to use this package is shown in Fig. 2.1.

Each of the package related components will be discussed in turn. Components that need to be made by the user are discussed in the sections where relevant (largely in the optimisation components of the workflow).

2.1 SigMat Model Format

The SigMat model format is designed to, in some ways, mirror the components of the SBML format. However, it has been simplified and streamlined for improved userfriendliness.

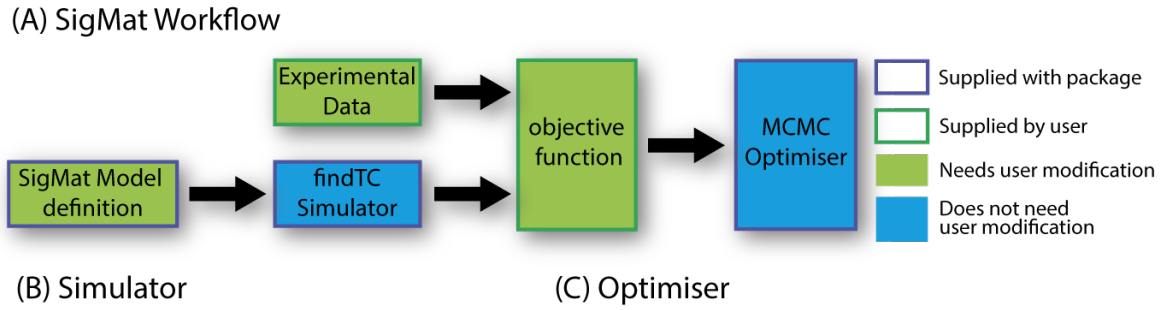

Figure 2.1: Illustration of the basic workflow in sigMat. Blue outline indicates a package defined syntax, blue fill indicates code included in the package. Green outline indicates custom syntax, green fill indicates custom code required.

Models define five different sets of information:

1. Model options. The only one relevant currently is whether species quantities in the model are absolute amounts or concentrations

2. Compartments. Defines a list of compartments which species may be localised to in this model.

3. Species. List of species that exist in this model and the compartment name in which they reside.

4. Reactions. A list of chemical reaction equations and their relevant parameters that make up the interactions in the system.

5. Default boundaries for the possible sigMat parameter classes.

sigMat models are stored as `.m` scripts. These can be created using the command:

```
>> mkModel(filename)
```

This generates a barebones sigMat model file with the same name which can be manually populated. It also contains some basic explanations. An example model file (more concise than the file generated by `mkModel`) for illustrative purposes (which is used as a quality control test case for the package) is shown below:

```
%% Model options
spcMode = 'a';
rxnRules = @odeKinetic;

%% Compartment definition
% spcComp = {'Compartment name', relative size};
```

```
modComp = {'Cytosol', 1;
           'Plasma_Membrane', 0.1};

%% Model species definition
% modSpc ='State name', 'Compartment'  , conc/param;
modSpc = {'A'         ,'Cytosol'  , 1;
          'B'         ,'Plasma_Membrane' , 0};

%% Reactions
rxn(end+1).desc = 'A -> B';
    rxn(end).sub = 'A';
    rxn(end).prod= 'B';
    rxn(end).k   = 0.1;

%% Features of default parameters
% Bnd.* = [lb ub]
% * is the parameter type
Bnd.k0  = [1e01 1e04];
Bnd.k1  = [5e-5 5e-1];
Bnd.k2   = [5e-5 5e-1];
Bnd.Km   = [1e-2 1e02];
Bnd.Conc = [1e-1 1e1];
Bnd.n   = [1 4];
Bnd.Comp = [0 1];
Bnd.r    = [0 1];
```

Some of the numbers in the model are **parameters**. Keep this in mind as you read through this documentation. The concept of **parameters** will not be explained yet. Just know that these are all the same class of numbers from a model definition perspective and there is a syntax system associated with them that gives them special meaning.

We will discuss each of these components in turn.

2.1.1 Model Options

Various options "switches" may need to be declared at the top of the file. Currently only two such "switch" exists. First is the `spcMode` switch. This switch is a string and when:

- `spcMode = 'a';` - Species quantities in the model file are absolute amounts (to be converted into concentration later).

- `spcMode = 'c';` - Species quantities in the model file are concentrations.

The second is the `rxnRules` switch. It points to the reaction rule set that will be used to compile the model. odeKinetic is what comes with the package. If a custom rule set is made, this switch can be changed to point to that rule set. See the advanced section for how rule sets can be made.

2.1.2 Defining Compartments

Model compartment is defined with the `modComp` cell array. It is an $n \times 2$ array. Each row defines a new compartment. Cells in the first column contains strings that defines the name of the compartment. Cells in the second column contain **parameters**. An example is:

```
%% Compartment definition
% modComp = {'Compartment name', relative size};

modComp = {'Cytosol', 1;
           'Plasma_Membrane', 0.1};
```

Note: compartment names MUST be unique.

2.1.3 Defining Species

Model species is defined with the `modSpc` cell array. It is an $n \times 3$ array. Each row defines a new species. Cells in the first column contains strings that defines the name of the species. Cells in the second column contain the compartment name of the species associated with that row. *The name of the compartment here must match one that has been defined in* `modComp`. Cells in the third column contain **parameters**. An example is:

```
%% Model species definition
% modSpc ={'State name', 'Compartment'  , conc/param};

modSpc = {'A'          ,'Cytosol'         , 1;
          'B'          ,'Plasma_Membrane' , 0};
```

2.1.4 Defining Reactions

Model species is defined with the `rxn` structure array. Each array structure indicates a different reaction. Each structure contains the following fields:

- label: A name for the reaction. Has no impact on the function of the package.
- sub: cell array with each cell containing strings of previously defined species names of species that are substrates in this reaction. There can only be a maximum of two substrates (as three substrate reactions are considered rare and is better modelled as multiple two substrate reactions).
- prod: cell array with each cell containing strings of previously defined species names of species that are products of this reaction. There is no limit to the number of this.
- enz: string of previously defined species names of species that is *the* enzyme of this reaction. I.e. there can only be one enzyme.
- k: rate constant. Is a **parameter**.
- Km: equilibrium/dissociation constant. Is a **parameter**.
- n: hill coefficient. Is a **parameter**.
- r: geometry coefficient (describing interface size relative to compartment size of cross compartment reactions). Is a **parameter**.

Unlike the other sets of information, where all components must be defined, not all fields in the structure needs to be given a value. In fact, the presence and absence of information in the fields is how the package determines what the relevant rate equation is. Table 2.1 shows a table outlining the structure configuration and the inferred rate equation.

So the following example,

```
rxn(end+1).desc = 'A + B -> C';
    rxn(end).sub = {'A','B'}
    rxn(end).prod= 'C';
    rxn(end).k   = 0.1;
```

infers a second order reaction. The following example,

```
rxn(end+1).desc = 'A -> B | E';
```

```
    rxn(end).sub = {'A'};
    rxn(end).prod= {'B'};
    rxn(end).enz = 'E';
    rxn(end).k   = 0.1;
```

infers a mass action simplification of enzyme kinetics. And the final example,

```
rxn(end+1).desc = 'A -> phi | E';
    rxn(end).sub = 'A';
    rxn(end).enz = 'E';
    rxn(end).k   = 0.1;
    rxn(end).Km  = 10;
    rxn(end).n   = 1.3;
```

infers a hill function based degradation reaction.

Note: it is not possible to have a two substrate enzymatic reaction, because the substrate enzyme interaction will already make it a two species reaction.

2.1.5 Defining Parameters

With the syntax introduced up to now, it is possible to fully define a model. However, more often than not, a system is not completely known, which is why they are studied in the first place. The most common source of uncertainty is in the parameterisation of the model. Thus, parameters need to be defined as being "constrained" or "unconstrained", and if unconstrained, whether the boundary is "known", "unknown" or dependent on another unknown parameter.

Table 2.1: Rules relating to reaction rates are inferred from the structure of the reaction that is parsed by SigMat. Dashed line separates groups of reactions that use rate laws of the same form.

	Substrate	Enzymes	Products	k	K_m	n	rate law
Synthesis	✗	✗	> 1	✓	✗	✗	$v = k$
Enzymatic synthesis High Km	✗	✗	> 1	✓	✗	✗	$v = k[E]$
Degradation/ Conversion/ Dissociation	1	✗	✗ 1 > 1	✓	✗	✗	$v = k[S]$
General enzymatic	1	✓	> 0	✓	✓	✗	$v = \text{dQSSA}$
Hill function	1	✓	> 0	✓	✓	✓	$v = \frac{k[E]^n}{[E]^n+[Km]^n}$
High Km enzymatic	1	✓	> 0	✓	✗	✗	$v = k[S][E]$
Association	2	✗	1	✓	✗	✗	$v = k[S_1][S_2]$

Table 2.2: Rules relating to how parameters should be assigned and how they are interpreted by SigMat.

# vector elements	1	2	3	4
Parameter Interpretation	Unconstrained free or fixed	Grouped parameters related by a factor.	Ungrouped free but constrained	Constrained reference parameter of a group
Syntax	[NaN or $R+$]	[NaN or $R+$, N]	[NaN, $R-$, $R+$]	[NaN, N, $R-$, $R+$]

$R+$ is any real positive number (including zero), N is any positive integer

Parameters in this case, includes any part of the model definition that has been referred to as a **parameter**, and the following discussion and syntax will equally apply to all of them.

The nature of the parameter is defined by the vector that appears in the corresponding cell array or structure field. This is laid out in table 2.2.

The interpretations are, when the vector has the following number of elements:

1. The parameter is:
   - Fixed if it is a real positive number
   - Free, using the default parameter boundary if NaN
2. The parameter is multiplicatively related to another parameter. The second element indicates the group identification number. If the first element is:
   - NaN, then it is the reference parameter for that group.
   - A positive real number, it is a multiplicative factor of the reference parameter for that group. This number is the multiplicative factor.
3. The parameter is free and independent with a custom boundary. The second element is the lower boundary. The third element is the upper boundary.
4. The parameter is the reference group of a group of multiplicatively related parameters with a custom boundary. The second element is the group identification number, the third and fourth elements are the custom lower and upper bounds respectively.

With this convention, the program determines how many truly free parameters the model contains and the associated parameter boundaries of each parameter, which is used later in the workflow.

2.1.6 Default Boundaries

Model default boundaries are defined with the `Bnd` structure array. This is only a 1×1 array. The structure contains one field for each class of model **parameters**:

- k0: zeroth order rate constant. This is placed in the "k" field in a `rxn` structure.
- k1: first order rate constant. This is placed in the "k" field in a `rxn` structure.
- k2: second order rate constant. This is placed in the "k" field in a `rxn` structure.
- Km: Michaelis or dissociation constant. This is placed in the "Km" field in a `rxn` structure.
- n: Hill constant. This is placed in the "n" field in a `rxn` structure.
- r: Reaction geometry term. This is placed in the "r" field in a `rxn` structure.
- Conc: Species concentration. This is placed in the third column of `modSpc`.
- Comp: Compartment size. This is placed in the second column of `modComp`.

The boundary definition is important for model tuning at a later stage. We will discuss this more in a later section. But for now, just understand that the function of `Bnd` is, if the boundary of some model **parameters** are not given boundaries explicitly, then they will be taken from `Bnd` based on the class the **parameter** in question is.

Below is an example of how the `Bnd` structure is defined.

```
%% Features of default parameters
% Bnd.* = [lb ub]
% * is the parameter type

Bnd.k0   = [1e01 1e04];
Bnd.k1   = [5e-5 5e-1];
Bnd.k2   = [5e-5 5e-1];
Bnd.Km   = [1e-2 1e02];
Bnd.Conc = [1e-1 1e1];
Bnd.n    = [1 4];
Bnd.Comp = [0 1];
Bnd.r    = [0 1];
```

2.1.7 Parsing and compiling the model

Before beginning, it should be noted that parsing of the model is generally not required as the function in built in most sigMat routines. However, producing the compiled sigMat model object before passing it onto subsequent toolbox functions does have some advantages. These are:

- Model validation: Running the model can be validated using the parser and output errors related to errors in the model syntax.
- Summary of model details: The compiled sigMat model contains various fields and objects that contain useful labels. This may be important for automated programs that required isolating particular states or parameters for manipulation.

The parser can run using the following function:

```
>> compiled_model = parseModel(model_name)
```

This will be discussed in more detail in a later section.

2.2 The `findTC` (find time course) simulator

Once the model file has been constructed, it can be used for simulation purposes. The syntax for this function is:

```
[t,Y_Dis,Y_Ass,model] = findTC(model_name,tspan,...)
```

This is written with a similar syntax as Matlab ODE (ordinary differential equation) solvers.

2.2.1 Function Inputs

There are two mandatory inputs to this function. `model_name` is either passed as a string of the model name, or as a function handle (the location of the file must be added to Matlab's current seach path with the `addpath` command). `tspan` is a vector of time points to be simulated (output behaviour is similar to Matlab ode solvers). The remaining inputs are all optional and are specified with Name,Value pairs where the name is a string. The possible Name,Value pairs are:

- `p`: vector of parameter values to be substituted in place of the free parameters in the model
- `inp`: input into the system (such as dynamic perturbations). The input method will be dicussed after this list.
- `y0`: customised initial concentrations which overrides any that are defined by the model and/or parameter set. Note this input always implies a concentration.
- `errDir`: A string with the directory for outputing errors from the function.

Inputs have a number of inputs depending on the input type. Using for example a model with four states, 'A','B','C', and 'D', the various input methods are: By

		Input Type	
		Initial step	Time variable function
Reference	By species name	`{'A',0.1}`	`{'A',@(t) 0.1exp(-t)}`
	By full vector	`[0.1,0,0.5,0]`	`@(t)[0.1,0,0,0]*exp(-t)+ [0,0,0.5,0]*exp(-2t)*2`

species name allows you to specifically reference a species to perturb and the algorithm will automatically find the **one** correct species index to apply the perturbation to. However, if the full vector is referenced, the control of the indexation is ceded to the user. Although this is more complex, it provides the ability to make more complex combinations of inputs. This is not so important for *initial step* as all perturbations applied using this syntax are triggered at the beginning as a step like function. However, when perturbed using the *time variable function*, all states can have different time functions for the induced perturbations. If the output of the function is not long enough (i.e. does not have as many elements as species in the model), the algorithm automatically pads the **end** of the vector with zeros.

There are also two flags that do require an associated value (i.e. the name for the next option is entered as the next input argument). These are:

- **-r**: This means no initial ramping of the model for determining the quasi-steady state before actual simulation is performed. This can be used to increase efficiency of the system when the dQSSA model is not used.
- **-b**: This means no basaling of the model. This may be necessary when the initial condition of the model does not start at the system's steady state, and it is necessary to start the simulations at the system's actual steady state. Note, like the above, the basaling period is not stored.

2.2.2 Function Outputs

There are four outputs of the findTC simulator. These are:

- `t`: The time points associated with the simulation.
- `Y_Dis`: The simulation results of species concentrations *after enzymatic complexes are dissociated*.
- `Y_Ass`: The simulation results of species concentrations while *enzymatic complexes are still associated*.
- `model`: The compiled model object. The contents of this will be discussed in a later section, but it contains useful information that can be used to post-process the simulated data.

Two different simulation outputs are given due to the use of the dQSSA. The dQSSA implicitly accounts for the concentration of enzyme-substrate complexes, and these states are included in the Y_Ass output. However, most experiments measure protein concentrations of each individual protein. In other words, complexes are dissociated during the experimental quantification process. So the dissociated simulated results may be a more accurate comparison to experimental results. Due to how common this is, an explicit output was created for the time course with complexes dissociated.

2.3 The `MCMC` Markov Chain Monte Carlo optimisation routine

The MCMC optimisation routine performs a stochastic search of the free parameter space to find a parameter combination that minimises an objective function. **The objective function must be written by the user**, thus we will begin this section by giving a brief explanation of what an objective function is, to help the user determine the best way to construct this.

2.3.1 Introduction of objective functions

An objective function is a calculation, dependent on a number of variables (free parameters), that returns a number which is some value of success. Typically, the smaller the number the more successful the "parameter" is able to achieve the "objective", which is why it is called an objective function. Using this application as a concrete example, the "objective" is to match the `findTC` simulated result to the experimentally derived result.

The "parameters" are the kinetic parameters that determine reaction rates etc which when altered, changes the behaviour of the simulation. The "success" measurement is thus some quantitative difference between the simulated and experimental data, where a large value typically determines more difference between the curves. For some parameter value, "success" measurement will be the smallest possible. At this parameter value, the objective function is described as "minimised". The process of searching through different parameter sets to the find the one that minimising the objective function is known as "optimisation".

As an example, the "success" measurement between the experimental time course and the simulated time course might be calculated using least squares. This would begin by doing the simulation. Say a model called `Insulin_Sig.m` was simulated between 0 and 600s, using the parameter set `pTest` with an insulin stimulation of 1 nM:

```
[t,Y] = findTC('Insulin_Sig',[0,600],'p',pTest,'inp','Insulin',1)
```

We then want to extract the relevant state from the simulated data:

```
YSim = Y(:,5);
```

Next, we scale the data to the experimental data `YDat`, which is an N x n matrix with n replicates, using some custom scaling function `scaleDat`:

```
[YSim,YDat] = scaleDat(YSim,YDat)
```

Then we find the standard deviation of the experimental data across the rows:

```
Y_SD = std(YDat,[],2);
```

Finally, we calculate the least squares difference between the experimental and simulated data (duplicating the number of columns in the simulated data vector and standard deviation vector to take into account the number of replicates in the experimental data). The results are then summed to get the total least squares residual value which is then a measure of the success of the fit.

```
resid = sum(sum([(ones(1,n)*YSim-YDat)./(ones(1,n)*Y_SD)].^2/2));
```

So the full objective function, which might be called `objFun.m` could look like:

```
function resid = objFun(pTest)

[t,Y] = findTC('Insulin_Sig',[0,600],'p',pTest,'inp','Insulin',1)
YSim = Y(:,5);
[YSim,YDat] = scaleDat(YSim,YDat)
Y_SD = std(YDat,[],2);
resid = sum(sum([(ones(1,n)*YSim-YDat)./(ones(1,n)*Y_SD)].^2/2));
```

2.3.2 Background into Markov Chain Monte Carlo

While there are many procedures aimed at achieving optimisation, many of which are bundled with Matlab (fminsearch, fminbnd, ga, simulannealbnd just to name a few). Markov Chain Monte Carlo (MCMC) is a sophisticated optimisation technique that is not bundled with Matlab. This is a global optimisation technique (which, to cut a long story short, does not get trapped while parameter searching) that is motivated by probability theory. There are many excellent resources that describes this algorithm and the probability theory underpinning in detail. However, for completeness a short description is included here to provide the reader with a basic appreciation and understanding of the process.

Relationship to Probability

MCMC is based on the concept that the objective function is a measure of the probability that the simulated result reproduces the mean of the random distributions generated from the experimental data. Figure 2.2 illustrates this concept. In this example, there are four time points that were experimentally measured with the mean shown. Two single parameter (gradient) fit lines are tested for the likelihood that they correctly reproduce the mean behaviour of the data (blue and green lines in upper figures). To do this, a normal distribution centred about each fit line (green and blue) is constructed. The probability of sampling the experimental mean (circles in lower figures) is then determined from each of these distributions. The final probability is then determined by multiplying the probabilities from each distribution function together (i.e. all the geen circles and all the blue circles). In this example, we find the blue fit to have a probability of 0.648 and a lower 0.24 for the green fit. This relative goodness measure can be verified visually. This type of objective function has a special name, called the "maximum likelihood estimator" because it isn't just an arbitary measure of goodness of fit, but directed related to likelihood of the line predicting the system's mean behaviour in the probabilistic sense.

While this is the theory surrounding maximum likelihood estimation, the astute reader will notice that MLE required "maximisation" rather than "minimisation". Many

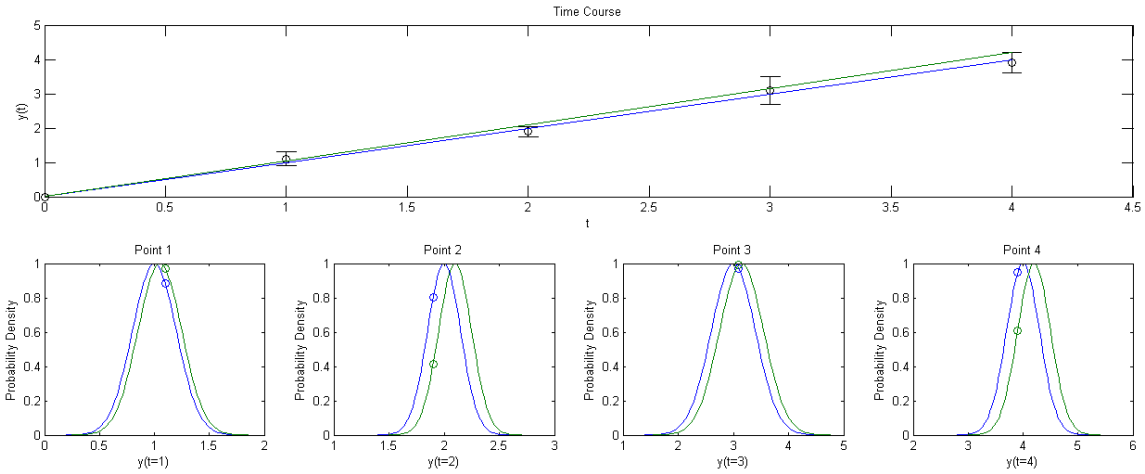

Figure 2.2: .

applications use $-log_e(P)$ in order to rectify this difference. By taking this transformation, large values correspond to low probability with a zero success measurement corresponding to a probability of 1 (the best fit possible) and allows it to conform to the "minimisation" convention commonly used for optimisation techniques. This is irrelevant for the MCMC algorithm though because it deals entirely with the probability rather than the success measurement.

Search Method and Metropolis Algorithm

MCMC, as previously stated, focuses on the probability of the objective function. The more probable a region is, the more often the algorithm searches in that area. Basically the result is, if the history of the movement of the algorithm is tracked, the step density in each region is proportional to the probability of the parameter fit in that area. To achieve this outcome, the algorithm repeatedly switches between two stages:

1. Step selection: a new candidate parameter set is chosen from the current parameter set

2. Step evolution: the likelihood produced by the new parameter set is calculated and used to determine if the algorithm proceeds to move to the candidate step

The step evolution stage can be considered the "exploration" stage. In simple terms, the MCMC routine (much like other optimisation routines) will begin at some arbitrary point (how this point is initialised is beyond the scope of this background) and move according to some criteria. The step evolution stage governs the movement direction of this point but is only a small part of the mechanics of the algorithm.

The step evaluation step is the "movement" stage. In this stage, the algorithm takes the parameter step found in the exploration stage and the probability of the objective function at that point to the probability at the current point in order to evaluate whether to proceed or not. The criteria used for this evaluation is known as the Metropolis-Hastings algorithm. This takes the ratio of the probability of the new step to the current step, and proceeds to the new step if the ratio is larger than 1. In other words, the algorithm always proceeds to improve the success measure. However, when the ratio is less than one, there is still a chance that the algorithm will proceed to the new point. When this happens, a random number between 0 and 1 is chosen, and compared to the ratio. If the selected number is smaller than the ratio, the new point is chosen, else the algorithm stays at the same place. Below is a summary of the Metropolis-Hastings criteria for acceptance of the candidate point:

$$\frac{P(x')}{P(x)} > rand([0,1]) \tag{2.1}$$

Recall that the step density taken by the algorithm is a reflection of the probability. The Metropolis-Hastings algorithm is what enables this. By accepting movement in a suboptimal direction at a rate equal to the ratio of the probabilities, and staying at the same spot otherwise, the eventual step density between the two points will have the same ratio, which leads to the proportionality between step density and probability.

Annealing to improve coverage

MCMC, with its probabilistic element, is supposed to have the potential to escape local minimas. However, when local minima are especially small and far apart even this stochastic behaviour is insufficient to prevent this trapping of the active point. In this case, the objective landscape can be "flattened" using an annealing strategy. This involves skewing the probabilities such that smaller probabilities appear larger.

Annealing is simply implemented by a slight alteration in the metropolis hastings criteria:

$$\left(\frac{P(x')}{P(x)}\right)^{1/T} > rand([0,1]) \tag{2.2}$$

Where T is the annealing temperature. When the temeprature is larger than one, the probability ratio is enlarged which increases its movement chance.

2.3.3 The `MCMC` function in SigMat

Implementation of this optimisation method is contained in the `MCMC` function in the SigMat package. It is designed to have a similar arguments as other Matlab optimisation functions. The function has the following input and output arguments:

```
[pts,logP,ptUnique] = MCMC(objfun,pt0,bndry,opts)
```

The function workflow

The `MCMC` is extremely complicated so in this section only the basic process will be explained. **Bolded** components of the run denote customisable options. The `MCMC` function performs the following stesp:

1. Processing of inputs and initialisation of basic features
2. Open **parallel computing**
3. Initialisation of advanced features
4. Select starting point (using pt0 if given, else selecting uniformly from the boundary)
5. Beginning of MCMC which ends until the run stores the required **number of unique points required** as set out in the run settings
    a) Choose new candidate active point after **resample** number of points (not applicable if no boundary or prior given)
        i. Select a point either by:
            - Randomly selecting from within search boundary
            - Randomly selecting from a **prior** with probably of selecting a point weighted by its probability
        ii. Accept point if Metropolic-Hasting criteria satisfied. Else use previous point.
    b) Advance the MCMC point by performing a step selection and then a step evolution process
    c) Store new step in the chain if it meets the **probability threshold** set for the run. Noting the parameter set (pts), the logP value (logP), and whether the point is new or not (ptUnique).
    d) Various progress check measures
6. Close parallel computing
7. Post processing of run outputs

The parallel computing portion is set up with a single master-node (typically node 1) and the remaining assigned as slave-nodes. The master node acts as the repository for the result obtained from all nodes, and slave-nodes periodically transfer their data to the master-node.

Input Arguments

The function has four input arguments. These are:

- objfun: Objective function to be explored. This must be a function with only one input argument and should either be a function handle or a function file.
- pt0: An N x 1 vector of the starting point for the MCMC chain. Can be left blank if bndry is passed for a random start point to be chosen automatically.
- bndry: Boundry within which the MCMC algorithm will be restricted within. This is an N x 2 matrix where N is the number of parameters to be fitted. The first column contains the lower bounds while the second column contains the upper bounds. Can be left blank if pt0 is passed.
- Opts: A struct that contains MCMC settings. It is advised that this be generated using the MCMCOptimset function. MCMCOptimset will be discussed in more depth later.

Output Arguments

The function has four output arguments. These are:

- pts: An M x N matrix containing all the stored points from the run.
- logP: An M x 1 numerical column vector of the corresponding $-log(P)$ value for the parameter set from the corresponding row in pts.
- ptUnique: An M x 1 logical column vector stating whether the parameter set from the corresponding row in pts is unique or not.
- status: A status flag that shows the reason why the run was terminated. 0 means normal termination. -1 means termination because the run took too long.

The `MCMCOptimset` MCMC setting creator function

Although `MCMC` is the primary function used to run the MCMC function, it contains many functionalities that can be customised. There are simply too many to be passed into `MCMC` as name, value pairs. As such, this function has been delegated to the `MCMCOptimset` function. Many of the options associated with `MCMC` are beyond the scope of this section, so we will only cover a small subset of it.

The syntax for this function is:

```
options = MCMCOptimset('Param1',Value1,...)
```

or

```
options = MCMCOptimset(oldopts,'Param1',Value1,...)
```

The first variant is for initialising the options. The second variant is for updating a previous set of options. Those familiar with Matlab will find that this is similar to matlab optimisation routine optimset functions, such as `optimset`, `saoptimset` or `gaoptimset`.

The basic option names (and their default values) are:

- PtNo: The number of unique points to be stored by the run. Value is a single integer. Defaults at 10,000.
- T: The run temperature (a determination of how much the objective landscape is flattened). Value is a single number. Defaults at 1 (no annealing).
- prior: A prior structure which the algorithm can draw seed points from. This structure must contain three fields: pts, logP and ptUn. pts is an M x N matrix with M parameters sets each with N parameters. logP is an M x 1 column vector of corresponding logP values of the corresponding row in pts. ptUn is an M x 1 logical column vector of whether the corresponding parameter set is unique or not. These are placed together as prior.pts, prior.logP and prior.ptUn. Empty by defaults.
- Pmin: The minimum probability value a parameter set needs to achieve before it is stored in the results. This probability IS modified by the annealing temperature. Value is a single number. Defaults at 0.
- Display: The type of display the program will produce as the run is going. Value is a string and defaults to “Termina”. The possible options are:

  - 'Off': Nothing will be displayed.
  - 'Terminal': Outputs will only be displayed on the terminal, and only from the master-node (node 1).
  - 'Text': Outputs will be printed into a text file called "Output-T_XX.txt" where XXX is the run temperature. Again outputs will only be printed from the master-node (node 1).
  - 'Debug': Outputs will be printed into a number of text files for debugging purposes:
    * An "Output-T_XXX-Slave Y.txt" where XXX is the run temperature and Y is the node number. This file contains the program outputs from each node in their own text files.
    * An "Output-T_XXX-Slave YcheckPoint.txt" where XXX is the run temperature and Y is the node number. This file contains a single number indicated what stage the program is currently up to.
    * A "DebugWorkspace-T_XXX-SlaveYY.mat" where XXX is the run temperature and YY is the node number. This file is an image of the workspace for each node at the beginning of current iteration of the MCMC loop.

- dir: The directory where display outputs will be stored. Value is a string. Defaults at '.' (i.e. current folder).
- DispInt: Run progress display interval. This determines the number of times the program will print its progress. Value is a single number. Defaults at 10.
- Walltime: Maximum time (in minutes) the program is allowed to be stuck in a single spot for. Value is a single number. Defaults at 60.
- ParMode: Switch for parallel computing functionality. Value is a number of logical. Defaults at true. The options are:
  - True: Parallel computing will be used and the maximum number of nodes available to MATLAB will be opened.
  - Number: Parallel computing will be used with the specified number of nodes connected.
  - False: Paralle computing (and all parallel computing functions) will NOT be used.

- PassNo: Number of points to be stored in each slave-node before they are scheduled to pass the data onward to the master-node. Value is a single number. Defaults at 500.

- Resample: The burn-in chain length. Or the number of times the MCMC step must be evolved since a seed point is chosen before reseeding can occur. Value is a single number. Defaults at 50.

2.4 The parsed sigMat model

The parsed sigMat model contains the following fields:

- name: model name

- rxnRules: function name containing the kinetic model rule set used to compile this model

- spcMode: switch indicating whether species quantities are an "amount" or "concentration"

- modSpc: description of model species

- modComp: description of model compartments

- params: matrices used to construct the dynamic equation. Customisable

- pFit: summary of parameter identities

These fields often contain complex sub-fields. Many of these govern how the model is mathematically constructed and is thus beyond the scope of this section. In spite of this, it does contain some information that is useful for performing analysis. Below will list some commands that can be used to extract various important information about the model.

- Species names ordered by their output from findTC. `model.modSpc.name`.

- Compartments that exist within the species. `model.modComp.name`.

- Compartment each species resides within. `model.modSpc.comp(model.modComp.name)`. The notation is as such because model.modSpc.comp gives the compartment index each species resides within.

- Number of free parameters in the model. `model.pFit.npar`.

- Identity of parameters. `model.pFit.desc`.
- Boundary of the parameters. `model.pFit.lim`.

These information should be sufficient to relate output matrices from either model simulations or parameter fitting routines with the biological concepts they represent.

3 Advanced Simulation Features

The features that have been discussed consists of the functional components of the SigMat package. However, many components of the describe workflow cannot be universally applied and any given algorithm or method may not be appropriate for certain circumstances. As such, SigMat was also designed to be highly customisable in areas that need to be made flexible. To explain this, a more thorough description of the parsed SigMat model needs to be described.

Before doing this though, its necessary to outline the overarching philosophy of the SigMat software architecture to understand how it can be manipulated for other applications. It is based on three principles:

1. The final ODE used to simulate the system is always based on matrix operations or be as close to emulating matrix operations as possible.

2. Construction of these matrices are performed using rules that are consistent in as many different contexts as possible in order to make them universal. Thus the algorithm needed for processing these can be hardcoded.

3. Non universal components are made as flexible as possible, and separated from hardcoded components so they can be customised.

With this in mind, we will begin a description of the matrix based design of the dynamic equation.

Note, it is highly recommended that those venturing into the next few sections have intermediate or advanced knowledge of MATLAB.

3.1 The Dynamic Equation

This package is written in MATLAB, short for Matrix Laboratory. It's traditional strength, as the name suggests, is in performing extremely efficient calculations of matrix operations. As such, it stands to reason that efficient construction of the underlying ODE would be achieved using a matrix based method. This is the fundamental motivation behind the SigMat package.

So the primary question is, how can a dynamic equation $\dot{x}_i$, be constructed out of matrix operations? Sometimes it is trivial since $\dot{x}_i = k_i$ in the case of a zeroth order rate law. In some cases it can be easily summarised with a matrix operations, such as first order rate law which can be implemented as $\dot{x}_i = k_{ij}x_j$. In other cases, some preprocessing is required before a programmable matrix operation can be implemented, such as the hill equation which looks like $\dot{x} = kS[E/(E + K_m)]^n$.

Since MATLAB is limited to performing matrix multiplication between 2-D matrices by vectors (and external packages that perform multidimensional matrix multiplications are highly inefficient), the optimal way to perform this is to pre-process the rate equation and collapse all excess components into a 2-D matrix called κ such that the equation looks like $\dot{x}_i = \kappa_i j x_j$ before the Matlab's matrix multiplication is used. There are no rules surrounding the construction of κ. As an example, κ for the hill equation is calculated using the following expression:

$$\kappa_{ij} = k_{ijk}\left(\frac{E_k}{E_k + K_{ijk}^m}\right)^{n_{ijk}} \tag{3.1}$$

While for a second order rate equation, κ is calculated using:

$$\kappa_{ij} = k_{ijk}S_k \tag{3.2}$$

Once κ_{ij} is calculated, the matrix can be constructed using the `sparse` function. What the `sparse` function does is to construct a 2-D matrix using a list of i and j indicies and places their corresponding matrix element into the relevant location into the final matrix. While it appears that indexation can achieve this, in practice it cannot due to the possibility of having multiple κ_{ij} with the same index. Duplicated indices need to be summed together. However, indexation replaces duplicated indicies instead. Fortunately, the `sparse` function essentially performs indexation but aggregates duplicate indices by summation.

For a more concrete example, consider the following second order rate equation case with two reactions with substrate 2 and 3 acting on substrate 1 to produce product 4 and 5. The relevant κ are:

- $\kappa_{11} = -k_{112}S_2$ which corresponds to $dx\_dt_1 = -k_aS_1S_2$
- $\kappa_{21} = -k_{212}S_2$ which corresponds to $dx\_dt_2 = -k_aS_1S_2$
- $\kappa_{41} = k_{212}S_2$ which corresponds to $dx\_dt_4 = k_aS_1S_2$
- $\kappa_{11} = -k_{113}S_3$ which corresponds to $dx\_dt_1 = -k_bS_1S_3$
- $\kappa_{31} = -k_{313}S_3$ which corresponds to $dx\_dt_3 = -k_bS_1S_3$
- $\kappa_{51} = k_{513}S_3$ which corresponds to $dx\_dt_5 = k_bS_1S_3$

There are two diferent values of κ_{11} in this case. If simple indexation is used to construct the matrix M, i.e.

```
M([i+(j-1)*size(M,2)]) = k_ij
```

then MATLAB will simply substitute $M(1,1)$ with the final value, which is $-k_{113}S_3$. However, what is correct is $M(1,1) = -k_{112}S_2 + -k_{113}S_3$. However, using the function:

```
M = sparse(i,j,k_ij)
```

will produce $M(1,1)$ with the correct summation. Once the final matrix is constructed, the ODE can then be constructed using the matrix equation:

$$dx\_dt = M \times S \tag{3.3}$$

The example provided only accounts for second order and other more complex rate equations. Other expressions may be necessary in some cases. The dynamic equation given in Wong et al. 2015 gives one example that covers first order reactions, second order reactions, the dQSSA enzyme kinetic model. The current dynamic equation used in sigMat also includes the hill function:

$$dx\_dt_i = (I_{ik} + G_{ik})\backslash((U_{kj}x_j + (W_{kj} + \text{HillTerm}_{kj})x_j + \sigma_k \circ V_k) \circ (1/V_k); \tag{3.4}$$

Where U is the first order reaction matrix, W is the second order matrix, HillTerm is the hill equation and G is the dQSSA prefactor term. All of these are different types of κ's. σ is a zeroth order term and finally V is a compartment size correction term. Note that $\circ$ is an element by element product known as the Hamadand product. Another thing to note is that κ's, when multiplied by a concentration vector, must produce an absolute species amount, so it must incorporate a volume correction. This will be discussed more later.

As demonstrated in this section, the crticial step in constructing the dynamic equation is in constructing the various κ's that make up the dynamic equation. In the next section we will cover how the κ's are constructed.

3.2 Dynamic Equation Matrices

The dynamic equation matrices κ are to put it simply, constructed using element by element operations after breaking the equation up into their separate components. Take for example the hill function. We had a hill function with reaction velocity:

$$v = kS\left(\frac{E}{E+K}\right)^n rV$$

r is a linear multiplicative geometric factors and V is the compartment volume. They will not be explained here, but see section 3.7 for more details. For completeness they will be carried on in the rest of the description. κ is then:

$$\kappa = \begin{bmatrix} -k\left(\frac{E}{E+K}\right)^n rV & 0 & 0 \\ k\left(\frac{E}{E+K}\right)^n rV & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This can be easily broken down into:

$$\kappa = \texttt{sparse}\left(\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix} -k\left(\frac{E}{E+K}\right)^n rV \\ k\left(\frac{E}{E+K}\right)^n rV \end{bmatrix}\right)$$

$$\kappa = \texttt{sparse}\left(\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}-k\\k\end{bmatrix} \circ \left[\begin{bmatrix}E\\E\end{bmatrix} \circ \left(\begin{bmatrix}E\\E\end{bmatrix} + \begin{bmatrix}K\\K\end{bmatrix}\right)^{\circ -1}\right]^{\circ \begin{bmatrix}n\\n\end{bmatrix}} \circ \begin{bmatrix}r\\r\end{bmatrix} \circ \begin{bmatrix}V\\V\end{bmatrix}\right)$$

So we now have the rule for constructing κ, which to be explicit, is:

$$\kappa = \begin{bmatrix}-k\\k\end{bmatrix} \circ \left(\begin{bmatrix}E\\E\end{bmatrix} \circ \begin{bmatrix}K\\K\end{bmatrix}^{\circ -1}\right)^{\circ \begin{bmatrix}n\\n\end{bmatrix}} \circ \left(\left[\begin{bmatrix}E\\E\end{bmatrix} \circ \begin{bmatrix}K\\K\end{bmatrix}^{\circ -1}\right]^{\circ \begin{bmatrix}n\\n\end{bmatrix}} + 1\right)^{\circ -1} \circ \begin{bmatrix}r\\r\end{bmatrix} \circ \begin{bmatrix}V\\V\end{bmatrix}$$

Or more concisely:

$$\kappa = C \circ \left[A \circ B^{\circ -1}\right]^{\circ D} \left[\left[A \circ B^{\circ -1}\right]^{\circ D} + 1\right]^{\circ -1} \circ E \circ F \tag{3.5}$$

Where

$$\begin{aligned} A &= \begin{bmatrix}E\\E\end{bmatrix} & B &= \begin{bmatrix}K\\K\end{bmatrix} & C &= \begin{bmatrix}-k\\k\end{bmatrix} \\ D &= \begin{bmatrix}n\\n\end{bmatrix} & E &= \begin{bmatrix}r\\r\end{bmatrix} & F &= \begin{bmatrix}V\\V\end{bmatrix} \end{aligned}$$

Here, A is a species index, while B, C, D and E are parameters. If we concatenate the 2×1 matrices, we achieve the prematrix equivalent ($\bar{\kappa}$) of κ.

$$\bar{\kappa} = \begin{bmatrix} 1 & 1 & 3 & r & -k & K & n \\ 2 & 1 & 3 & r & k & K & n \end{bmatrix} \tag{3.6}$$

These species and parameters now correspond with the 3rd to 7th column of the prematrix. Note that we've replaced A with its equivalent species index. Using this convention, the Hill function has been condensed into a single pre-matrix (Eqn. 3.6) and a rule associated with how this matrix is to be processed (Eqn. 3.5).

For those paying close attention, notice that the V's are not included in the $\bar{\kappa}$ matrix. This is because V is handled in a different part of the code. This is addressed in section 3.9.

3.3 Insertion of Free Parameters

Continuing the example of the Hill function, we now note that we have distilled the function down into a rule and a pre-matrix. However, a key component of model testing is in varying parameters. Additionally, some parameters are factors of other parameters. So how can we dynamically change parameters under this regime.

The SigMat solution to this is to duplicate the $\bar{\kappa}$ matrix, producing the parameter index matrix. The parameter index matrix, as the name suggests, contains the parameter index in the free parameter vector, which contains the value required in the corresponding matrix position in the $\bar{\kappa}$ matrix. This logic is visually demonstrated in Fig. 3.1. By using this scheme the κ and parameter matrices do not need to change in order to change the parameter set. As the numbers in the parameter vector change, the parameter matrix fetches the relevant free parameter value (by indexation) and places the fetched values into the corresponding position in the $\bar{\kappa}$ matrix.

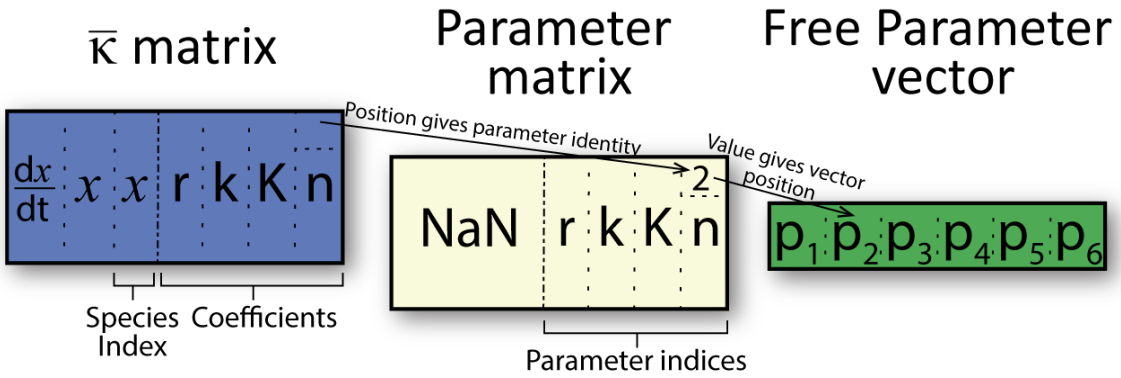

Figure 3.1: How parameters are inserted into $\bar{\kappa}$ matrices

As we mentioned earlier, some parameters are multiplicative factors of others. In order to account for this, the $\bar{\kappa}$ matrix is prepopulated with the multiplicative factor. When the required free parameter vector value is fetched, it is simultaneously multiplied by the multiplicative factor within the $\bar{\kappa}$ matrix.

If the parameter in the $\bar{\kappa}$ matrix was never defined to be free in the first place, then the parameter matrix in that position should contain an NaN, which tells SigMat to not fetch any values and so the original value within the $\bar{\kappa}$ matrix is preserved (notice that the entire species index side of the parameter matrix is NaN for this reason).

Although the focus of this discussion has been on the $\bar{\kappa}$ matrix, this method of parameter insertion is universally used in the SigMat algorithm. This also applies to any other parameter type values such as concentrations and compartment sizes (obviously with no species index in concentration and compartment size matrices because the matrix element denotes the species or compartment identity in those cases).

3.4 Parameter Description

With the construction of the parameter matrix, it is possible to link the κ matrix with the free parameter vector. In order to give meaning to the free parameter vector, a number of attributes also need to be defined. First is the parameter description and second is the parameter type, which is essentially used to determine parameter boundary for optimisation purposes.

The parameter description is stored as a cell vector that has the same length as the free parameter vector. The description cell contains a description of the parameter in a corresponding cell in the free parameter vector.

The parameter boundary on the other hand is a $2 \times n$ matrix with the same height as the free parameter vector. The first column contains the lower boundary and the second column contains the upper boundary and the row number contains these boundaries for the corresponding parameter in the free parameter vector with the same index. The default values that get placed into the rows is identified using the fieldnames of the boundary structure.

3.5 Construction of Dynamic Equation Matrices

In this section, we will provide a detailed workflow of the steps that need to be conducted to construct the required matrices used by the SigMat algorithm. Note that this is not designed to be performed manually, but the description is given to merely provide the reader with an appreciation of the methods employed by the algorithm. This appreciation is necessary for the reader when they come to design their own rule sets for the algorithm.

This section will be divided into two components. The first section takes a broader overview of the matrix construction process. This is mainly focused at describing what the required matrices will look like after the parsing process. The second section will dig deeper at the algorithmic way SigMat constructs these matrices and what components of the rule set function is drawn on to achieve this.

3.5.1 Matrix Construction Example

The prerequisite parts of the parameter definition (excluding the top level free parameter vector input) include the following:

- $\bar{\kappa}$ matrix

- Parameter matrix
- Parameter description cell vector
- Parameter boundary array

For a distinct reaction, the κ matrix is well defined. However, it also changes from reaction type to reaction type. So this is something that must be customisable and manually defined. In this walkthrough, we will continue our use of the Hill function. Starting with a bare bones reaction structure from a SigMat model files of a Hill function reaction:

```
rxn.desc = 'S -> P | E';
rxn.sub = {'S'};
rxn.prod= {'P'};
rxn.enz = 'E';
rxn.r   = 1;
rxn.k   = NaN;
rxn.Km  = NaN;
rxn.n   = 1.3;
```

Using equation 3.6 as a template for the required $\bar{\kappa}$. The species index stays the same (assuming the global index of the species, product and enzymes for this reaction are 1, 2 and 3 respectively). For the parameters, we note that two are fixed and two are free. The fixed parameters are 'r' and 'n'. These belong to the 4th and 7th columns respectively. These matrix elements are populated with their fixed values. The free parameters, 'k' and 'Km', belong to the 5th and 6th columns. Since these are free parameters, based on the implementation convention, the $\bar{\kappa}$ columns are populated with multiplicative factors instead. Since no multiplicative factors have been explicitly set, these are by default given values of 1. So the $\bar{\kappa}$ matrix for this reaction looks like:

$$\bar{\kappa} = \begin{bmatrix} 1 & 1 & 3 & 1 & -1 & 1 & 1.3 \\ 2 & 1 & 3 & 1 & 1 & 1 & 1.3 \end{bmatrix} \tag{3.7}$$

Next is the construction of the parameter matrix, which we recalled is a mirror of the κ matrix. Recall that this matrix is designed to contain the free parameter vector indicies which the $\bar{\kappa}$ matrix extracts parmeters from, and NaN everywhere else. To cut a long story shore, SigMat copies the structure of the $\bar{\kappa}$ matrix but changes its content algorithmically such that non-free parameter elements are set to NaN. Free parameters on the other hand, are set the automatically assigned integer value of the index in the parameter vector. In this case 'k' and 'Km' are the 1st and 2nd free parameters to they have been assigned index 1 and 2. Thus, the matrix parameter now looks like:

$$\text{Param Matrix} = \begin{bmatrix} \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 1 & 2 & \text{NaN} \\ \text{NaN} & \text{NaN} & \text{NaN} & \text{NaN} & 1 & 2 & \text{NaN} \end{bmatrix} \tag{3.8}$$

Then there is the parameter description. The parameter description is algorithmically generated by combining the parameter tag with the reaction description.

The reaction description is constructed by scanning the reaction structure and inferring the reaction that is occuring. Substrates appear on the left with each species separated with a '+', then an arrow '-¿' is printed, followed by the products with each species again separated with '+'. Enzymes, if involved, are the denoted on the right, separated with a '—' from the products. In the case of the Hill reaction, the constructed reaction description looks like:

```
S -> P | E
```

The parameter tag consistes of two components. First is the parameter index, which is automatically assigned. Second is the parameter type, which is manually but categorically assigned. For example the 'Km' in Hill functions can be given the type `K_Hill`. This will give parameter 2 a tag of `K_Hill`. However, all "Half occupation concentrations" in all subsequent Hill reactions will be given the same parameter type. The parameter index is also included automatically. So the two parameter tags generated are:

```
1| k :
2| K_hill :
```

Combining these two pieces gives the following descriptions:

```
1| k : S -> P | E
2| K_hill : S -> P | E
```

Which is simply assigned to the corresponding entry in the description cell vector.

Finally is the boundary array. The required default boundary is determined by the user defining which field in the boundary structure to extract the boundary from. Suppose the default boundary structure (defined in the model file) looks like:

```
Bnd.k0   = [1e01 1e04];
Bnd.k1   = [5e-5 0.5];
Bnd.k2   = [5e-5 5e-1];
Bnd.Km   = [1e-2 1e02];
Bnd.Conc = [1e-1 1e1];
Bnd.n    = [1 4];
Bnd.r    = [0 1];
Bnd.Comp = [0 1];
```

We can then choose 'k1' and 'Km' as the default boundaries for parameters of type k and K_Hill in Hill functions respectively. The SigMat algorithm will then fetch and apply these defaults everytime these parameters types are encountered with no custom boundary defined.

3.5.2 Algorithmically implementing this convention

The algorithmic method (lines 335-386 in parseModelm) for implementing the matrix construction convention by iterating through each reaction in the model and applying three computational steps on them:

1. Modifying reaction structure values (testPar function).
2. Generating reaction specific components, i.e. matrices and other description/boundaries (rxnRules('rxnrules',...))
3. Appending reaction specific components into their corresponding global components (internal to parseModelm)

These steps will be describe using an example. Take the simplest case of a reaction structure, one with no free parameters. I.e.

```
rxn.desc = 'S -> P | E';
rxn.sub = {'S'};
rxn.prod= {'P'};
rxn.enz = 'E';
rxn.r   = 1;
rxn.k   = 2;
rxn.Km  = 3;
rxn.n   = 4;
```

We can directly make the following matrix as well as where the matrix needs to go:

```
reqTens  = 'Hill';

       -                                  -
mat = |Si Si Ei rxn.r -rxn.k rxn.Km rxn.n|
      |Pi Si Ei rxn.r  rxn.k rxn.Km rxn.n|
       -                                  -
```

Where Si, Pi and Ei are the index of the substrates, product and enzyme respectively. The construction of this matrix is performed entirely by step 2.

Let us consider a slightly more complicated case, where all parameters are free variables:

```
rxn.desc = 'S -> P | E';
rxn.sub = {'S'};
rxn.prod= {'P'};
rxn.enz = 'E';
rxn.r   = NaN;
rxn.k   = NaN;
rxn.Km  = NaN;
rxn.n   = NaN;
```

If this reaction structure is passed into step 2, then we essentially generate a matrix that is the inverse of the parameter matrix. NaN for free parameters and integers for the species indices. To achieve the correct form for the parameter matrix, like equation , we instead modify the values within the reaction structure with:

```
rxnPar.desc = 'S -> P | E';
rxnPar.sub = {'S'};
rxnPar.prod= {'P'};
rxnPar.enz = 'E';
rxnPar.r   = 1i;
rxnPar.k   = 2i;
rxnPar.Km  = 3i;
rxnPar.n   = 4i;
```

Where $1i$,...,$4i$ are imaginary numbers. The matrix generated by step 2, using this structure, can then be converted into equation by first making all real numbers NaN, and then turning all imaginary numbers into real numbers. This process is performed by step 3. The process of converting `rxn` into `rxnPar` is performed by step 1. Then to construct the $\bar{\kappa}$ matrix, we can construct the following reaction structure.

```
rxnKap.desc = 'S -> P | E';
rxnKap.sub = {'S'};
rxnKap.prod= {'P'};
rxnKap.enz = 'E';
rxnKap.r   = 1;
rxnKap.k   = 1;
rxnKap.Km  = 1;
rxnKap.n   = 1;
```

Here, all 1's are the multiplicative factors. Conversation of the NaN to their multiplicative factors are again performed by step 1. Step 2 will construct the required $\bar{\kappa}$ matrix.

What is important to note here, is step 1 and 3 as described are entirely automated. It can construct `rxnKap` and `rxnPar` as the required reaction structures for generating $\bar{\kappa}$ and the parameter matrix respectively, without any functional changes to step 2. All step 2 needs to do, is to assume that the input reaction structure describes a reaction with no free parameters.

This procedure covers both the $\bar{\kappa}$ and parameter matrices. However, the algorithm must still generate and store the parameter descriptions and boundaries. These are also primarily performed by step 2. Here, the parameter description and default boundary are placed inside the corresponding field of the reaction structure. As an example, to define the description and parameter type (which is used to extract default boundary) for the parameter 'k', we assign the description and boundary into `rxn.k`. This would look like:

```
rxn.k = {param_Type,Desc}
```

Step 2 then outputs the modified reaction structure for use by step 3.

What may become evident is, much focus is placed on making step 2 as simple but generalisable as possible. This is because step 2 is the customisable component of the algorithm and ultimately controls how the $\bar{\kappa}$ matrix, and hence κ matrix is constructed. Note that step two also has the ability to add new species to the model if that is required by the reaction. This will be discussed in more depth in a later section.

3.6 Designing the reaction rule file

As a summary for this chapter, the reaction rule file has four components. These are:

1. 'ini': the initialisation component
2. 'rxnrules': the longest component, the matrix construction rules for each reaction type
3. 'compile': pre-construction or compilation of matrices before simulation
4. 'dynEqn': pre-construction of matrices and dynamic equation construction during simulation

The rxnRules file must look like the below (with the given input arguments and output arguments). All lines need to be kept intact except the lines with `<Insert *>`:

```
function varargout = rxnRules(method,varargin)
```

```
switch lower(deblank(method))

case 'ini'
%<Insert parameters>
varargout = {param};

case 'rxnrules'
[rxn,modSpc,flag,ii] = varargin{:};
%<Insert reaction rules>
varargout = {reqTens,matVal,modSpc,rxn};

case 'compile'
if length(varargin) == 2
[model,tspan] = varargin{:};
elseif length(varargin) == 1
model = varargin{:};
tspan = [0 1]; %no non-dimensionalise
end
%<Insert pre-compilation steps>
varargout = {model};

case 'dyneqn'
[t,x,model] = varargin{:};
%<Insert compilation steps and dynamic equation calculation>
varargout = {dx_dt};
end
```

A clean reaction rule file can be created by running the following command:

```
>> mkRule(filename)
```

The next subsections will look at the requirements of each section of the rule set.

3.6.1 'ini'

In the previous sections we have covered generation of the $\bar{\kappa}$ matrix for the Hill function. However, many other dynamic equations exist with a different (or even multiple) $\bar{\kappa}$'s. The different types of $\bar{\kappa}$ required by the model needs to be predefined. This initialisation component of the rule set and has no inputs. It defines all of the matrices required in the rule set. The matrix definitions simply need to described the number of columns in

each matrix. Matrices are placed within the `param` structure with unique fieldnames. The fieldnames then become the unique identifier for the matrix. These matrices must be constructed with NaN.

For example, the default rule set has:

```
%           P_i k0
param.k0 = [NaN NaN]
%           P_i S_i  r  k1
param.k1 = [NaN NaN NaN NaN]
%           P_i S1_i S2_i  r  k1
param.k2 = [NaN NaN  NaN  NaN NaN]
```

These correspond to the $\bar{\kappa}$ matrices that needs to be made for a zeroth, first and second order reaction. The commented parts annotate the meaning of each column. Param is then outputted back to the model parser.

3.6.2 'rxnRules'

This is defines the population rule for the matrices that have previously been initialised. It takes input arguments:

```
rxnRules('rxnrules',rxn,modSpc,flag,ii)
```

`rxn` is the reaction structure, `modSpc` is the parsed model species structure, `flag` is an assortment of flags for modifying behaviour (not really used currently), and ii is the index of the reaction tested (mainly used for error outputs). Thus, the key input is `rxn` and the other three are simply made available for the use if required.

At its heart is the construction of a $\bar{\kappa}$ matrix given some characteristic of the input reaction structure. For example if `rxn.k` is the only parameter with a value, and there are two substrates, the reaction may be classified as a bimolecular reaction. It may be useful to programa classifier into the rxnRules part. However, the optimal method of choosing a matrix structure given a `rxn` structure is left entirely to the user.

For construction of the matrix, three steps are compulsory. These are firstly the param field identification step (where the matrix will go), second is the matrix population step, and finally is the description definition step. These correspond with three of the output arguments, which are:

```
[reqTens,matVal,modSpc,rxn] = rxnRules('rxnRules',...)
```

reqTens stores each param fieldname in a cell vector. matVal contains the required reaction specific matrices, each matrix within a cell in a cell vector. The cell vector index in matVal corresponds with the cell vector index in reqTens. For example, for a dQSSA model implementation (which involves two matrices):

```
reqTens  = {'k1','Km'};
matVal   = {[subIndx     compIndx          [r -k];
             prodIndx  prodVec*compIndx prodVec*[r k]];
             [compIndx  subIndx enzIndx   r -Km;
              compIndx  enzIndx subIndx   r -Km;
              subIndx   subIndx enzIndx   r  Km;
              subIndx   enzIndx subIndx   r  Km;
              enzIndx   subIndx enzIndx   r  Km;
              enzIndx   enzIndx subIndx   r  Km;
             ]};
```

Recall that for this step, we can assume that all parameter values, where present, can be placed within the matrix as if they're fixed.

Finally for the parameter description, the required parameter description and the type of parameter it is is stored in the reaction structure and passed back to the calling function. The cell structure is {'paramType ,'description'}.

'paramType' is manually defined for each parameter. However, a helper function exists for creating 'description'. This is the `makeDesc` command. The syntax for the function is:

```
outputDescription = makeDesc(subList,prodList,enzList,paramDesc)
```

`subList`, `prodList` and `enzList` are all cell arrays containing the substrate, product and enzymes of the reaction (note that there is at most one enzyme in any reaction). `paramDesc` is a string that describes the parameter to be manually included, for example `'K_Hill'`. Empty substrates and product lists will produce a 'phi'.

So with the description covered, the boundary (if the default to be used is 'k1') and description can be inserted with the following syntax in the reaction rules:

```
rxn.k = {'k1',makeDesc(subList,prodList,enzList,'k')}
rxn.k = {'r' ,makeDesc(subList,prodList,enzList,'r')}
rxn.k = {'Km',makeDesc(subList,prodList,enzList,'Km')}
```

Note that `'modSpc'` can also be modified. This includes all subfields within it. If there is a need to add a new species based on a reaction (perhaps it is a stable intermediate),

note that modSpc.name is the list of species names, modSpc.matVal is equivalent to the $\bar{\kappa}$ convention, modSpc.comp is modComp index of the home compartments of the interacting species, and modSpc.pInd is equivalent to the parameter matrix convention of parameters. Generally the safest way to add new species is to add them at the end of the list to prevent the index of existing species from shifting (and hence disrupting the model parsing process).

To summarise, the `rxnrules` component of the rule set has at least the following steps:

1. Classification of reaction structure into reaction type
2. Definition of param field to place the matrices into reqTens
3. Creation and population of matrices into matVal
4. Creation of parameter descriptions and parameter type back into rxn

These are then outputted alongside `modSpc`.

3.6.3 'compile'

Compile is called after free parameters are placed within $\bar{\kappa}$ matrices. Many of the model data within the parsed model has also been stripped away for brevity. Typically what remains is a single vector for `modSpc` of the concentrations for each species and `modComp` of the compartment size of each species, while only `matVal` and `name` remain within `model.param`. It performs two functions.

1. Non-dimensionalises matrices
2. Manipulates and processes as many of the $\bar{\kappa}$ matrices as possible and to get them to appear as like κ as possible (to minimise calculations during the ode simulation stage).

Non-dimensionalisation simply involves taking the free parameters which are time dependent and rescales them such that the simulation is run between the time points $[0, 1]$. Currently no non-dimensionalisation is performed for the species concentrations/amounts.

A helper function is available for this tasks, known as `nonDim`.

```
nonDimMat = nonDim(rawMat,tspan,normInd)
```

This function takes in a matrix `rawMat`, and normalises it to `tspan` for columns `normInd`. This function needs to go inside the manually set rule set because `normInd` is unknown.

Next is the manipulaion and preprocessing of $\bar{\kappa}$. Sometimes $\bar{\kappa}$ can be preprocessed if the subsequent κ does not depend on a species state until the matrix multiplication stage. Or some matrices can be combined now that their parameters have been inserted. This can all be done before running the simulation to save on resources.

It is possibly a good idea to at this point hard code the identity of `model.param.matVal` based on their index in `model.param` because it can be computationally expensive to search for the correct index within `'dynEqn'`.

Note that all operations must apply directly to `model` as it is also the output for this part of the rule set.

3.6.4 'dynEqn'

Here we have now come full circle, ending at the dynamic equation again. The input arguments of this part of the rule set conforms to the function required by native ODE solvers in MATLAB. That is:

```
dx_dt = rxnRules('dynEqn',t,x,model)
```

`model` here is identical to that produced by the `rxnRules('compile',...)` process with a few minor differences. `model.sigma` is a function handle that is included within model and signifies a synthesis rate of change term. `model.time` is also a new field that contains the handle to a tic command. `toc(model.time)` can be used to monitor the run time of the simulation.

All that needs to be done in this part of the script is to finish compiling and uncompiled κ matrices, and then construct the differential equation as described in section 3.2.

3.7 Discretised model of compartmentalisation

Compartmentalisation can be implemented and accounted for in SigMat. However, it is suggested that this be performed not in the $\bar{\kappa}$ matrix, but at a later stage. This will become apparent as we walkthrough the technical background behind SigMat's implementation of compartmentalisation.

Compartmentalisation models generally start with a number of assumptions. Firstly, compartments are internally well mixed with a high concentration, allowing it to be modelled as a deterministic process. Secondly, transport between compartments is

slow compared to the mixing rate within compartments. When both assumptions are satisfied, transport between compartments can be modelled as elementary reactions.

Transport between compartments can be modelled by discretising Fick's law of diffusion. The net flux moving from compartment 2 to compartment 1 is described by the following expressions:

$$J = D\frac{\partial[X]}{\partial x} \tag{3.9}$$

$$\frac{1}{A}\frac{dX_1}{dt} = P_s\left([X_2] - [X_1]\right) \tag{3.10}$$

$$\frac{dX_1}{dt} = P_s A\left([X_2] - [X_1]\right) \tag{3.11}$$

where J is the flux into compartment 1, D is the diffusion constant, X is the absolute number of molecules, $[X]$ is the concentration, x is the spatial dimension, P_s is the permeability coefficient, A is the interface area between the two compartments.

These equations show that diffusion determines the number of molecules transported as a function of the concentration. In order to determine the rate of change of concentration in each compartment due to transport, the expressions need to be further manipulated by normalising to the volume of the compartment of interest. In other words:

$$\frac{d[X_1]}{dt} = \frac{P_s A}{V_1}\left([X_2] - [X_1]\right) \tag{3.12}$$

$$\frac{d[X_2]}{dt} = \frac{P_s A}{V_2}\left([X_1] - [X_2]\right) \tag{3.13}$$

Where V_1 is the size of the compartment which X_1 is in. Similarly for V_2.

We note that $P_s[X]$ is similar to the rate equation of a unimolecular reaction. The difference is P_s has units ms^{-1} while the unimolecular rate constant k has units s^{-1}. Noting this, we can separate out P_s as well as factor out V_1 and V_2, and rewrite Eqns. 3.12 as:

$$\frac{d[X_1]}{dt} = \frac{1}{V_1}\left(\frac{k_a\hat{z}A}{V_2}V_2[X_2] - \frac{k_a\hat{z}A}{V_1}V_1[X_1]\right) \tag{3.14}$$

This generates the apparent rate constant k_a and a combination of geometry terms. We can break up V_1 into z_1A_1 where z_1 is a characteristic depth of compartment V_1. This then allows A_1 to be be cancelled out from both the numerator and denominator. Subsequently, $\hat{z}$, the transport depth, can be combined with z_1 to form the geometry term r that is the ratio of $\hat{z}$ and z_i, which describes the percentage suppression of the reaction due to suboptimal interface area to compartment volume ratio.

$$\frac{d[X_1]}{dt} = \frac{1}{V_1}\left(rV_2k_s[X_2] - rV_1k_s[X_1]\right) \tag{3.15}$$

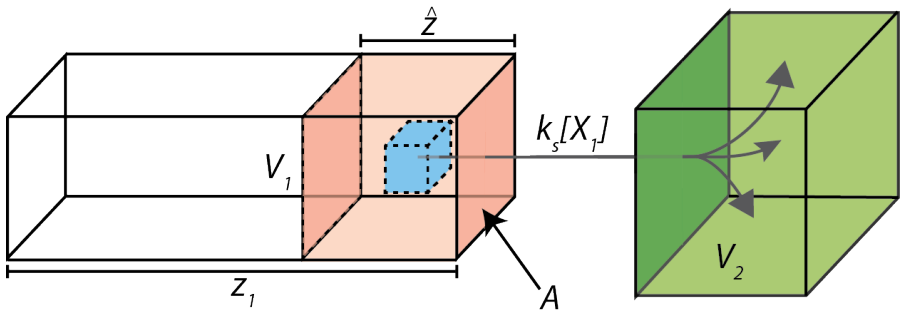

Figure 3.2: Figure showing the physical interpretation of the discretised model of transport between V_1 to V_2. X_1 within a unit volume (dash bordered cube) is transported to V_2 at a rate of k_s and disperses throughout V_2. All X_1 within the coloured volume of V_1 undergoes this transport process to form the total transported amount. The coloured volume is only a subset of the total volume of V_1 because this is restricted by interface area to volume ratio of the compartment, which affects the accessibility of X_1 available to be transported.

This gives the rate of change due to transport between compartments modelled as a unimolecular reaction. The interpretation of this is illustrated in Fig. 3.2. Using this interpretation, k_s can be considered the diffusion rate per unit volume under its most efficient conditions.

Compartmentalisation is implemented in SigMat essentially based on equation 3.15 If we recall equation 3.4 in section 3.1 ends with a $1/V$ term. This is the same as the leading $1/V_1$ term in equation 3.15. The r terms introduced in section 3.2 is in fact the geometric term r we have introduced just now.

The remaining component in equation 3.15 is the V_1 and V_2 terms next to their unimolecular rate like terms (i.e. $k[X]$). To explain how this is implemented in SigMat will require slightly more theoretical detail to be given in the next section.

3.8 Compartmentalisation of reactions SigMat

It is possible to expand the concept of transport to other rate laws using the following general expression:

$$\frac{d[X_j]}{dt} = \frac{\sum_i \dot{N}_i}{V_j} \tag{3.16}$$

Where $\dot{N}_i$ is the rate equation in terms of absolute molecules number from reaction i:

$$\dot{N}_i = r_i V_i v_i \tag{3.17}$$

and can be applied to Eqn. 3.16 to construct the rate equation for the species concentration, where v_i is any standard rate equation.

An example of a cross compartment unimolecular reaction is the dissociation of the plasma membrane localised insulin receptor - insulin receptor substrate complex. The elementary rate equation would look like Eqn. 3.15 with k_s instead being the unimolecular rate constant, r_i would be one (as the reaction is never limited by geometry) and V_1 would be the volume where the complex resides. The insulin receptor would remain in the plasma membrane while the insulin receptor substrate would dissociate into the cytosol. In this case, V_i is always the volume of the reactant.

The reverse case of complex association deserves some additional attention due to the choice required for the characteristic parameters. The two reactants insulin receptor and insulin receptor substrate originate from different compartments with different volumes. So what is the appropriate choice for V_i and r_i for this association reaction? Considering the nature of bimolecular reaction, the reaction can only occur in some volume where both species are present, and this is physically a thin layer surrounding the interface between the two compartments. From this consideration, $r_i V_i$ could define the thin volume surrounding the interface. Using the defintion of r we introduced previously, we can define the volume V_i to be the reacting volume when the geometric layout of the system is optimised, i.e. $r = 1$. This means the entire volume of one of the two compartments is in the overlapping region. This can only be the smaller of the two compartment volumes. Thus for bimolecular reactions, $V_i = \min(V_1, V_2)$ and $0 \leq r \leq 1$.

3.9 Implication for SigMat Reaction Rule

Implementation of compartmentalisation has two implications on the reaction rule set in SigMat. First is the modification of the reaction rate, and second is the choice of compartments in a dynamically generated reaction species (e.g. in the dQSSA enzyme kinetic model generated enzyme-substrate complex). This will be discussed in the following subsections.

3.9.1 Modification of reaction rate

Because the generic rule for the selection of V for some reactions requires a minimisation of two potentially unknown values, assignment of this cannot be encoded by the $\bar{\kappa}$/parameter matrix pair. This is because construction of the $\bar{\kappa}$/parameter matrix requires a parameter index to be explicitly defined. However, in this case, the parameter index required can change depending on the values in the free parameter vector.

To illustrate this, consider an example where $V_1 \rightarrow p_1$ and $V_2 \rightarrow p_2$. Now the volume V_i required for the bimolecular reaction i is $V_i = \min(V_1, V_2)$. If $p_1 < p_2$ then $V_i = p_1$. However, if $p_1 > p_2$ then $V_i = p_2$. This kind of conditional parameter assignment is very difficult to implement in the $\bar{\kappa}$/parameter matrix implementation, especially because the basic tenet of this implementation is that all columns are used to construct the final κ matrix.

To overcome this, note that the information required for choices available for V_i actually already exists in the $\bar{\kappa}$/parameter matrix pair. Since the indices of the reactants in the reaction are already encoded in the $\bar{\kappa}$ matrix, it is possible to infer their compartment sizes based on this information. So it is possible to wait until the free parameter vector has been defined before the algorithm makes the decision as to which V to choose for V_i. In fact, the `rxnRule('compile',...)` stage is the perfect time to perform this step.

Thus, it is recommended that modification of the reaction rate by the reaction volume is performed by taking the minimum of the reactant volumes, which can be extracted from a vector of compartment sizes each species resides in given by modComp during the compile stage.

3.9.2 Dynamically generated reaction species

Recall from section 3.6.2 that modSpc can be changed by the rxnRule part of the model rule set. Part of this includes defining what compartment index to assign the new species (defining the compartment the species belongs in) within the modSpc.comp field. Typically, it's possible to just make the new species reside in the same compartment as an existing species (possibly one of the reactants). However, sometimes a new species may come from a reaction between species residing in two different compartments. In such a case, it may be necessary for the algorithm to know the indices of both originating compartments to decide which one to pick once the volume sizes have actually been assigned. This is similar to the problem faced in the "reaction rate modification" part discussed previously.

In such a case, it is possible to have multiple compartment indices in each row of the modSpc.comp matrix. Each row in this case corresponds with the species occupying the same row in modSpc.name. Later in the algorithm, when the parameter values are inserted (and the compartment volume is explicitly defined), SigMat will take the minimum of volumes in each row and use that as the new species compartment size.

4 Advanced MCMC Features

The MCMC algorithm shipped with the SigMat package presents a reasonably simple implementation algorithm. This is because some of the "customisable" components of the algorithm have been implemented in a very simple way. As you may recall from section 2.3, the MCMC algorithm contains an "exploration" stage. This is the stage that takes the existing seed point of the MCMC algorithm, and generates a candidate point that is then passed through the "evaluation" stage. The candidate point at the exploration stage has no bearing on the probability of the function to be optimised, and so really the exploration can be any process as long as it meets two criterion:

- The exploration method as the ability to traverse the entire parameter landscape at some point in the optimisation process.
- A biased term can be calculated by the search method. The biase is ratio of the probability of selecting the candidate point from the seed point and the probability of selecting the seed point from the candidate point.

This exploration algorithm is termed the "proposal distribution" in the MCMC algorithm (as it proposes a candidate point) and is a "customisable" method in the SigMat package.

A proposal distribution usually has two key features:

- The proposal distribution shape
- The proposal distribution range

The shape denotes the shape of the probabiltiy distribution function used to select a candidate point. In the default example, an n-dimensional gaussian function with zero covariance is used. The range denotes how far the probabiltiy distribution function will search in the parameter space. In the default example. the range is the standard deviation of the gaussian function. Note that typically, the range of the probability distribution function will change dynamically in order to maintain a failure rate in the evolution stage of the algorithm. This failure rate is defined in the SigMat MCMC algorithm using the "AcceptRatio" flag in `MCMCOptimset`.

Both of these components are contained within the SigMat proposal distribution function. A proposal distribution function looks like:

```
function varargout = propName(fcn,runVar,opts)

switch lower(deblank(fcn))
    case 'newpt'
        %<Generate pt1 and pdfBias based on a custom proposal distribution>
        varargout = {pt1,pdfBias};

    case 'adapt'
        %<Modify runVar.step. Information from opts.stepi is available>

        varargout = {runVar};

end

end
```

This can be generated using the command:

```
mkProp(filename)
```

Each of the parts of the proposal distribution function will be discussed in more detail in the following subsections.

4.0.3 Candidate point selection with proposal distribution

The main purpose of the proposal distribution is to select a new candidate point. The information available to this function is all information within `runVar`. `runVar` contains the seed point, the parameter space boundaries and the current step size (akin to sampling range).

The key question in designing the proposal distribution is the probability distribution function which is to be sampled from. This can be absolutely anything distribution as long as it sample a single test point and a pdfbias as an output (this latter requirement almost always requires the proposal function to be analytically tractible).

Note however, that the point of MCMC is to move the search point to an optimal point whilst being reasonably exploratory. Thus, some proposal distributions will be more

efficient than others at achieving this goal. For example, for a narrow gaussian proposal distribution with a covariance would be more efficient than a uniform distribution.

The proposal distribution can also be dynamic, such that its shape changes as the algorithm evolves. New attributes can be created in runVar to facilitate this as long as it does not override an existing runVar field.

If you type:

```
>> edit propDis
```

The default proposal distribution will appear. This function uses independent gaussian functions, noting log-linear or linear scale variables, and ensures that parameters fit within the boundary. The pdfbias is also calculated (usually unbiased) especially noting cases where some of the proposal distribution falls outside of the boundary, which creates a pdfbias because the pdf that falls outside the boundary becomes invalid, increasing the probability of hitting points that lie within the boundary.

4.0.4 Proposal distribution range modification

Update of the proposal distribution is performed after the evolution stage and determines how the proposal distribution changes given the success/failure of the candidate point previously selected by the proposal distribution. There are a number of key factors that need to be considered when defining how the proposal distribution range is modified.

- The search range
- Rate of change of the search range with success/failure
- Change in the shape of the proposal distribution

The other component of the proposal distribution modification algorithm is in changing the range of the proposal distribution, which is known as the step size. This is one way the proposal distribution can be dynamically changed as the run progresses. This is necessary because as the algorithm approaches poorly fitting areas, it may need to reduce its step size in order to boost its success rate (bvy confining its search in a good area). Alternately in a good area, the search algorithm may need to increase its step size in order to increase the failure rate (making the algorithm more exploratorya dn to prevent it from becoming stuck in a local minima). The `opts.stepi` value is also an important point of reference in this algorithm as that denotes the minimum step size for the algorithm. If the algorithm is in a particularly poorly fitting region, it may

force the step size to be so small that no evolution occurs. The `opts.stepi` exists to prevent this.

The rate of change of the step size is also an important consideration. This is because updating the step size slowly causes the algorithm to be stiff and adapt poorly. Alternately a very fast rate of change can cause the algorithm to be erratic and prematurely classfiy an area as a local minima and escape out of it.

A third change that is possible is the shape of the proposal distribution. This is possible the most advanced of the modification methods. This may involve skewness or covariance of the proposal distribution such that the algorithm focuses on searching within certain beneficial dimensions.

In order to decide the proposal distribution modification regime, a number of fields in runVar are available. A particularly useful field is the `runVar.ptTest` field, which records a number of past successfully recorded points. Again any new data can be stored within `runVar` to enhance the proposal distribution function as long as it does not override existing fields.