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

August 3, 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 manually and 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 search path can be more permenantly added by finding the "Set Path" option in the MATLAB toolbar/menu and then adding `pathname` using the "add with subfolders" option.

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 **??**.

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 **??**.

# 1 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).

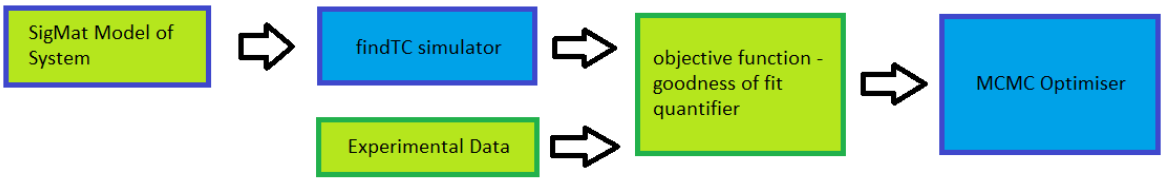


Figure 1.1: Illustration of the basic workflow in sigMat. Blue outline indicates a package defined syntax, blue fill indicates code included in the package. Green outline indicates custom syntax, green fill indicates custom code required.

## 1.1 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.

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. Comaprtments. 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';

%% Compartment definition
% spcComp = {'Compartment name', relative size};
modComp = {'Cytosol', 1;
           'Plasma_Membrane' 0.1};

%% Model species definition
% modSpc ={'State name', 'Compatment'  , 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.

### 1.1.1 Model Options

Various options "switches" may need to be declared at the top of the file. Currently only one such "switch" exists. This 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.

### 1.1.2 Model 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
% spcComp = {'Compartment name', relative size};
modComp = {'Cytosol', 1;
           'Plasma_Membrane' 0.1};
```

Note: compartment names MUST be unique.

### 1.1.3 Model 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 compartment. 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 defined in `modComp`*. Cells in the third column contain **parameters**. An example is:

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

### 1.1.4 Model 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,

Table 1.1: 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/ | 1 | ✗ | 1 | ✓ | ✗ | ✗ | $v = k[S]$ |
| Dissociation | | | $> 1$ | | | | |
| 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 | ✓ | ✗ | ✗ | $v = k[S_1][S_2]$ |

```
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.

### 1.1.5 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

Table 1.2: 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

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.

**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.

### 1.1.6 Default Boundaries

Model default boundaries are defined with the `Bnd` structure array. This is only a $1 \times 1$ 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];
```

### 1.1.7 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.

## 1.2 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.

### 1.2.1 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 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 species name allows you to specifically reference a species to perturb and

| | | 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` |

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* all perturbations applied using this syntax is 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.

### 1.2.2 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 proteins dissociated.

## 1.3 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.

### 1.3.1 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 into 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));
```

### 1.3.2 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

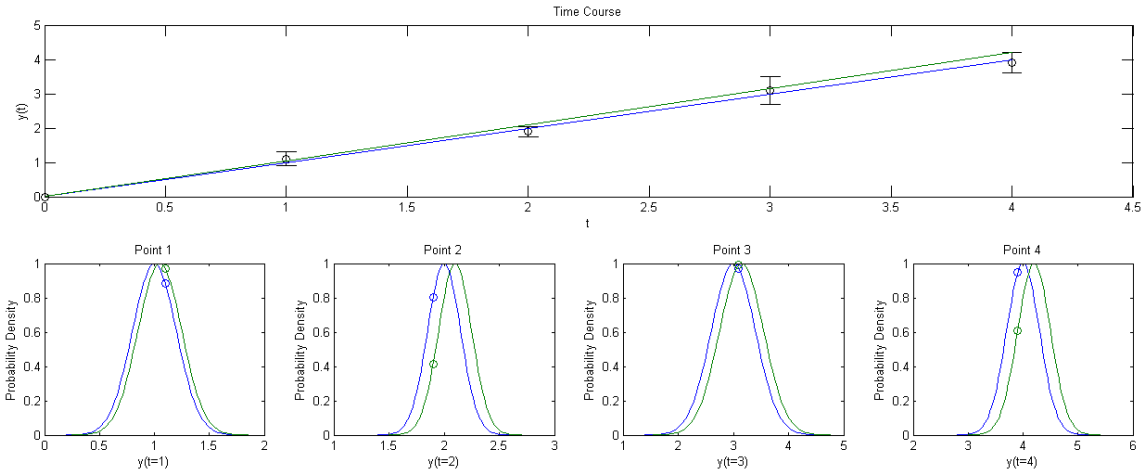


Figure 1.2: .

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 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 initiate 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:

$$\frac{P(x')}{P(x)} > rand([0,1]) \tag{1.1}$$

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:

$$\left(\frac{P(x')}{P(x)}\right)^{1/T} > rand([0,1]) \tag{1.2}$$

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

### 1.3.3 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.

## 1.4 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.

# 2 Advanced 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 description, and an expanded diagram of the SigMat software architecture must be shown.

Before doing this though, its necessary to outline the overarching philosophy of the sigMat software architecture. It is based on two 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. This 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.

## 2.1 The Dynamic Equation

This package is written in MATLAB and MATLAB is short for Matrix Laboratory. It's traditional strength has been extremely efficient calculations of matrix operations. As such, it stands to reason that an efficient construction of the underlying ODE of the simulation is matrix based. This is the fundamental motivation behind the sigMat package.

So the primary question is, how can a dynamic equation, say $\dot{x}_i$, be constructed out of matrix operations? Sometimes, such as the case of a zeroth order rate law, is trivial since $\dot{x}_i = k_i$. In some cases, such as a first order rate law, it can be easily summarised with a matrix operations since $\dot{x}_i = k_{ij}x_j$. In other cases, such as a hill function, some preprocessing is required before a programmable matrix operation can be implemented, since $\dot{x} = kS[E/(E + K_m)]^n$ and the exponentiation and division cannot be easily notated in index notation.

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 $\kappa$ 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 package specific rules surrounding the construction of $\kappa$. As an example, $\kappa$ for the hill equation is calculated using:

$$\kappa_{ij} = k_{ijk} \left( \frac{E_k}{E_k + K_{ijk}^m} \right)^{n_{ijk}} \tag{2.1}$$

While for a second order rate equation, $\kappa$ is calculated using:

$$\kappa_{ij} = k_{ijk} S_k \tag{2.2}$$

Once $\kappa_{ij}$ 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 simple indexation can achieve this normally, the fact that $\kappa_{ij}$ typically collapses information about a multi-dimensional matrix into two simply the $i$ and $j$ index means that there may be multiple values of $\kappa$ for any unique $(i, j)$ pair of indicies. Because the collapse of the other dimensions is meant to simulate a matrix multiplication, values of $\kappa$ with repeated indices actually need to be summed together. This summation process is less easy to do using simple indexing but it innately performed using the `sparse` function.

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 $\kappa$ are:

- $\kappa_{11} = -k_{112} S_2$ which corresponds to $dx\_dt_1 = -k_a S_1 S_2$
- $\kappa_{21} = -k_{212} S_2$ which corresponds to $dx\_dt_2 = -k_a S_1 S_2$
- $\kappa_{41} = k_{212} S_2$ which corresponds to $dx\_dt_4 = k_a S_1 S_2$
- $\kappa_{11} = -k_{113} S_3$ which corresponds to $dx\_dt_1 = -k_b S_1 S_3$
- $\kappa_{31} = -k_{313} S_3$ which corresponds to $dx\_dt_3 = -k_b S_1 S_3$
- $\kappa_{51} = k_{513} S_3$ which corresponds to $dx\_dt_5 = k_b S_1 S_3$

There are two diferent values of $\kappa_{11}$ 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:

$$dx\_dt = M \times S \tag{2.3}$$

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:

$$dx\_dt_i = (I_{ik} + G_{ik})\backslash((V)kj * x_j + (W_{kj} + HillTerm_{kj}) * x_j + \sigma_k * V_k)/V_k); \tag{2.4}$$

As demonstrated in this section, the core component of the sigMat software package is in building ODE models using matrix equation which then allows the equation to be writte in a concise method. The rest of the package involves methods for constructing the required matrices, substituting in free parameters, and how to turn a human readable model of the system into the matrices that make up the mathematical equivalent of the system. With this in mind, we will begin exploring the deeper aspects of the package with the parsed sigMat model, which contains the matrix templates for constructing the matrices associated with a particular network architecture to be studied.

## 2.2 Complete description of the parsed sigMat model

In order to understand the complete sigMat package architecture, it is necessary to first explain the details and motivation surrounding the model structure of a parsed sigMat model as it forms the motivation of the way the software architecture was designed. As described previously, 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
- pFit: summary of parameter identities

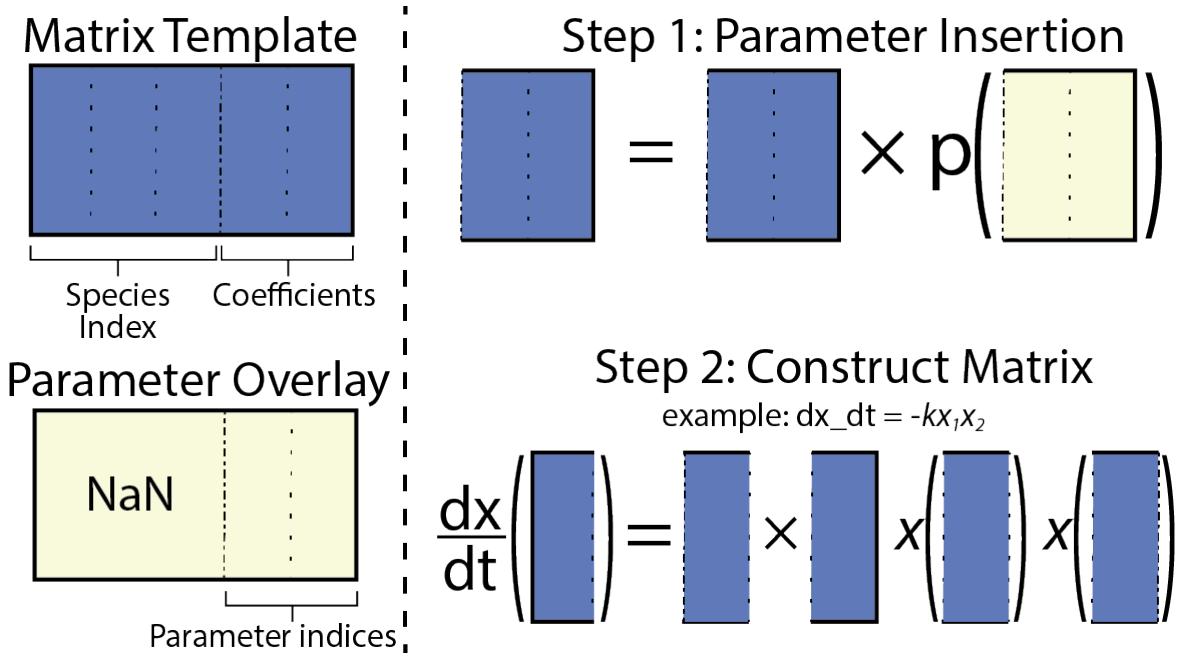


Figure 2.1: Illustration of universal matrix construction algorithm used.

The first three fields provide rough descriptions of the model. `Name` provides the "filename" of the unparsed model while `rxnRules` provides the name of the reaction rule function used to parse the model. This is annotated for reproducibility. `spcMode` is an annotation that denotes if species quantity given in the parsed model are to be treated as absolute quantities (e.g. as moles of X) or concentrations (e.g. as molar concentrations of X). This changes the behaviour of the model.

The remaining are complex structures that describe how the model is to be mathematically constructed. In spite of their complexity, they are designed to have a similar logic. That is parameters are described such that they can be be compiled into matrices using a universal algorithm. Matrix compiliation is composed of two components: the matrix template and the parameter index overlay. This is true for `modComp`, `modSpc` and all components of `params`. The universal algorithm used is described in Fig. 3.1 where each row forms one complete unit of information. The left pane describes the makeup of the matrix template and the parameter overlay. The right pane then describes how these are modified to produce the matrix that is used in the dynamic equation (in this case for a bimolecular rate constant).

The matrix template, as the name suggests, provides information regarding how the matrix is to be constructed. Broadly speaking, this matrix is divided into the species index component and the coefficient component. In Fig. 3.1, the division is illustrated with the dot-dash line with the former on the left and latter on the right.

We will begin by discussing the species index component. The species index component contains information about which index of the ODE the row in question applies to, and the indicies of species which are involved with controlling the rate of the reaction.

More specifically, the first column gives the index to be used in the ODE (e.g. index of the rate of change of the substrate or product) while the next few columns the species indices which the rate equation requires (e.g. index of the enzyme or substrates for example). Given the species index, it is not difficult to use the species index component of the matrix template as an index to substitute in actual species concentrations in the dynamic equation.