Compunetics approach v1.0.3 README

**GetStoichiometryConnor**

* *Calls: StoichiometryViaMILP2*
* *Inputs: Atomic*
* *Outputs: KeepV*

This function is the main function used for the black box generation of every possible reaction based on the ‘mass matrix’ of *Atomic*. *Atomic* is provided for the function as a matrix of the number of different atoms (such as carbon atoms, hydrogen atoms etc.) on each row, and each species on each column.

For example, for a following reaction where the identified species are:

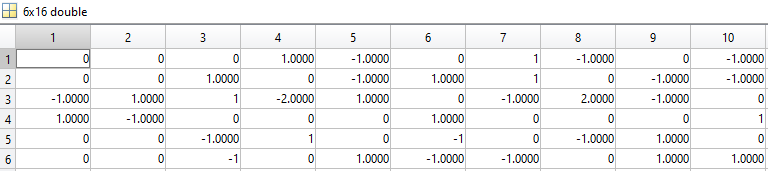


The following *Atomic* matrix would be correct:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** |
| **Carbon** | 4 | 4 | 8 | 8 | 12 | 0 |
| **Hydrogen** | 2 | 9 | 10 | 10 | 18 | 1 |
| **Nitrogen** | 2 | 1 | 3 | 3 | 4 | 0 |
| **Oxygen** | 0 | 1 | 1 | 1 | 2 | 0 |
| **Chlorine** | 2 | 0 | 1 | 1 | 0 | 1 |

After running the function, a *KeepV* output is generated, which is a matrix representing all of the possible reactions (based on the mass of the identified species) in an integer format, where:

* -X: ‘X’ number of reactant are used up in the reaction
* 0: The species does not participate in the reaction
* +Y: ‘Y’ number of product are formed in the reaction



For the above example, a 6x16 *KeepV* matrix is generated (6 species and 16 possible reactions). Reaction 1 is column 1, where the rows 1-6 represent A-F. Therefore in reaction 1, **C** reacts to form **D**. In reaction 8, **A** reacts with **E** to form two equivalents of **C**.

**OptimiseK\_2**

* *Calls: GenerateKineticLaws\_2, Reaction\_2, interuptFunfmincon*
* *Inputs: KeepV, Results2*
* *Outputs: SortedErrorTable, SortedKTable, SortedNoOfKTable*

This function is the main optimiser function which takes the experimental results in *Results2* (which is a matrix showing how the concentrations of each species change over time) and compares them to every combination of every possible reaction, where the K values for each combination are optimised to test the validity of that reaction model. This is explained in more detail below.

Firstly, the required structs are generated:

* ‘a’: an ‘alphabet struct’ initially containing letters and corresponding concentrations, which are overwritten later and used within the ODE solver.
* ‘Combinations’: generated as a placeholder for the code below which uses eval and nchoosek operators in a loop to generate all combinations of the columns. For example, Combinations.Comb3 for the above example will be a 560x3 array, where all combinations of 3 of the 16 *KeepV* columns are stored. An example combination could be “1, 5, 14”, meaning the reactions from columns 1, 5 and 14 in *KeepV* are combined and represent a particular reaction model. If for example you want the maximum
* ‘KV’: a ‘K Values’ struct, where K values are assigned during the fmincon optimisation process. This has been coded as a struct because as the reaction models change, you may need more or less K values that you needed previously, which cannot be used as an input for further functions. The least error-prone way of passing variable-length K amounts is by passing a struct.

*TotalCombinations* is also assigned a value, as is *Counter* in order for the following for-loops to work and *Conc* which is a vector of the known initial concentrations. ‘datafile.mat’ is deleted from any previous runs.

*GenerateKineticLaws\_2* is then called (which is explained in more detail within its own function README), which generates:

* ‘NumberofKValues’: this is a large vector that contains the required number of K values for each corresponding reaction model found in *TotalCombinationsFunctions*.
* ‘TotalCombinationsFunctions’: this is a large function array that contains every reaction model to try, where each row is one reaction model and each column represents how the concentrations are changed. Every function begins with ‘@(KV,a)0’ in order to use the inputs *KV* and *a* for the function, and 0 as the base function, so if that particular function isn’t necessary then it still runs but generates a 0. An example of how a reaction model within *TotalCombinationsFunctions* translates in the example reaction is shown below:

*TotalCombinationsFunctions notation:*



*Kinetic models notation:*

*Chemical reaction notation:*



A parallel pool is started (with the number of cores on your computer, default is 4) and the *ErrorTable*, *KTable* and *NoOfKTable* are defined here. This is because these tables will have values added to them via the parpool and so the correct amount of space must be pre-allocated in order to parpool to work.

*NumberofFunctionArrays* is defined as the ‘rounded-up’ number of 100-block reaction model chunks that can fit into the *TotalCombinationsFunctions* matrix. It was found when parallel computing that sending a large data file (*TotalCombinationsFunctions*) to each of the workers caused very large overheads and hence slow optimisations. Therefore it is much more efficient to splice this large data file into a smaller ‘block’, *CurrentCombinationsFunctions*, in order to send to the workers - this leads to a much faster optimisation. Each block is assigned during the for-loop where *CombNo* is the block number.

*interupt\_time\_fmincon* is a variable where the number of seconds allowed for each optimisation is defined, as default it is 200 seconds - it is very likely that if fmincon has not successfully optimised the K values for this reaction model within this time, then the reaction model is not a viable one. This variable plugs into the *interuptFunfmincon* function, which is called after every iteration of fmincon. This output function is simple, if the ‘state’ of the algorithm is ‘init’ is means it has just started and therefore the time is recorded. If the state is not ‘done’ then it is therefore ‘iter’ and it is iterating, meaning the times can be compared and if the elapsed time is greater than the *interupt\_time\_fmincon* variable then the optimisation will stop.

The parfor is then run. For each optimisation the number of K variables to be optimised will be different, so this is defined here as *CurrentNoofKValues* from the *NumberofKValues* large vector. This variable is then used to define the lower and upper bounds (*lb* and *ub*) for the optimisation as well as *x0* which are the starting K values that will be attempted.

*Reaction\_2* is then optimised using fmincon, specifically the K values for each reaction model within the block (100 models). The optimisation suitability criteria is defined within the *Reaction\_2* function README.

After the optimisation for the current model has ran, the outputs from the (fmincon) optimisation are compiled into their respective matrices and cells. After each block has been optimised, the following variables are saved in *datafile.mat* in case of a code (or power) failure:

* ‘ErrorTable’: A matrix showing the value that has been optimised for, which is the sum of squared error between the experimental results and the simulated ODEs. The lower this value, the more accurate the K values for the particular reaction model are.
* ‘KTable’: A cell array showing the K values found from the optimisation for the particular reaction model.
* ‘NoOfKTable’: A matrix showing the number of K values that were optimised for each reaction model, and hence the number of reactions the model is comprised of. This is important for analysis by Akaike’s Information Criterion (AIC), which determines the suitability of a model based on the accuracy of the model as well as the number of model terms. This analysis is important as not to identify the ideal model based only on the fitting to the data, as a very large reaction model could be made to fit any dataset with the correct K values. This stops the most ideal model from simply being the model with the largest amount of model terms.

After all optimisations have been ran, the *ErrorTable*, *KTable* and *NoOfKTable* are reshaped into long vectors from their respective matrices and cells, meaning AIC analysis can be run. After the AIC is ran, *SortedKTable*, *SortedErrorTable* and *SortedNoOfKTable* represent the AIC ranking of each model in descending order. For example, *SortedKTable*(1) will represent the best K values that represent the best ranked (by AIC) reaction model that gave the optimisation output value shown in *SortedErrorTable*(1).

**GenerateKineticLaws\_2**

* *Calls:*
* *Inputs: KeepV, a, TotalCombinations, Combinations*
* *Outputs: TotalCombinationsFunctions, NumberofKValues*

(Note: edit function with caution. Note: This cannot be parallelised due to *eval* operators.)

(Tip: For debugging main code, save *TotalCombinationsFunctions* and *NumberofKValues* to a file that can be re-opened instead of running this function again.)

Firstly, the required variables are assigned for use later in the function. Including defining the required *Alphabet* terms and defining *TotalCombinationsFunctions* as a cell array of the number of required terms, for the example reaction this number is 6.

The first set of embedded for-loops look through, in order, every combination of reaction terms (found in KeepV) in every field within the *Combinations* struct, referring to the differing amount of reaction model terms.

For each run, the relevant columns (containing the model terms of interested) are pulled from *KeepV* into a new matrix called *NewKeepV*. The model terms are then split up into the ones that decrease the concentration of a particular species, *Negatives*, and the ones that increase the concentration of a particular species, *Positives*. These model terms are then compiled into separate cell arrays of *EquationsNeg* and *EquationsPos* which contain all of the concentration-decreasing and concentration-increasing terms respectively for each species.

‘@(KV,a)0’ is first added to every function term within the function array. Then each function (reaction model term) is built by combining the concentration-decreasing and concentration-increasing terms for each of the species. For the example reaction, for each reaction model there should be 6 reaction model terms that each have separate reactions that affect the corresponding species’ concentration.

The reason a large function array is created as the method of storing all possible reaction combinations is because it is much faster to evaluate functions rather than use *eval* operators with strings found in a cell array, which is another possible route.

**Reaction\_2**

* *Calls: ReactionKineticLaws\_2, interuptFunODE*
* *Inputs: input, KV, Results2, KeepV, a, CurrentCombinationsFunctions, I, TotalCombinations*
* *Outputs: Error*

Firstly, the ‘*input’* input from the optimiser is assigned to the relevant K values in the model terms and these are the values that are optimised based on the output of this function. The *Time* variable is also assigned here with all of the times for the ODE solver to evaluate (in seconds).

The *interupt\_time* is used in the *interuptFunODE* function, which is an output function for the ODE solver (ode15s), and is the amount of time that can pass before the ODE solver is stopped, which is 4 seconds as default. This stops time-related problems arising in cases where the solver may need to integrate in very small steps, leading to lengthy integration times. Generally speaking, ODE solvers cannot be stopped once started - therefore this function works in a similar way to the aforementioned *interuptFunfmincon* function, although a coding error is purposely generated to exit the solver after a certain elapsed time (4 seconds by default). This is why the ODE solver is within a try-catch operation, as well as the sum of squared error (sse) operator as when the ODE solver is stopped there are no simulated results to compare to the experimental results.

**ReactionKineticLaws\_2**

* *Calls:*
* *Inputs: Time, Conc, KV, a, CurrentCombinationsFunctions, I*
* *Outputs: dydt*

This function assigns the ODEs and evaluates them at different times. fHandle is assigned as the function representing the reaction model for that particular species, then evaluated using *KV* and *a* as the inputs.