BOMBS Functions
Version 0.1.0

DAVID GOMEZ-CABEZA

David.Gomez@ed.ac.uk
January 27, 2021

1 Model Generation

1.1 defModStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the model generation section so only the values for each key have to be filled.
Outputs: Dictionary with keys NameF, nStat, nPar, nInp, stName, parName, inpName, eqns, Y0eqs, Y0Sim, tols and solver. Values for each key are empty lists [].

1.2 checkStruct(model_def)

Inputs: Dictionary with the keys specified in defModStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defModStruct()). If everything is correct, then some processing of the values will be done (mostly indexing and extraction of elements to ease further indexing of structures). If the value for some key has a wrong structure or content, the function will break (dictionary not returned) and a message will be printed to help the user identify where the issue is (mostly information about the key that had not passed the check).
Outputs: Same dictionary introduced with necessary modifications of some fields to aid it's use in further sections. If the value for any key is wrong, nothing will be returned.

1.3 GenerateModel(model_def)

Inputs: Dictionary with the keys specified in defModStruct() and filled values for each one.
Function: This function takes the information given by the user about the model and generates a Julia script with the necessary function to simulate the model.
The 4 functions contained in the script are:

- nameODE!(du,u,p,t): Function containing the ODEs of your model. For more information check `https://diffeq.sciml.ai/v2.0/`. du indicates the system of ODEs, u the value of the states, p parameters and t time.
- nameSteadyState(p,I): Function containing information about the steady state. This can be a set of equations or an empty function that returns the same y0 that you introduce. p is the parameter values and I the y0 values for each state (experimental values that need to be specified with the name of the state starting with exp).
- name_solvecoupledODE(ts, p, sp, inputs, ivss, pre=[]): This is the function that will allow you to solve the ODE system with different events (external inputs that change across the experiment, so no fixed parameters). ts is the time vector (from 0 to end time every 1), p is the vector of parameters, sp is the switching times for the external input, inputs is a matrix with the values for each external inducer for each step, ivss is the initial values for the model (y0) and pre is the concentration of the inducers for the steady state done before the start of the experiment (if any).
- name_SolveAll(ts, pD, sp, inputs, ivss, samps, pre=[]): This function allows you to simulate your ODEs using multiple instances for the parameters automatically. The inputs for the function are the same with the exception of pD (in this case this is the parameter matrix with all your theta samples to be used for the simulations) and samps (sampling time vector for the experiment, which might have a different resolution than 1).

Where name is the name you have given to the model in the key NameF.
Outputs: Same dictionary introduced with the extra key modelpath containing the path to the file generated.

2 Model Simulation

2.1 defSimulStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the model simulation section so only the values for each key have to be filled.
Outputs: Dictionary with keys Nexp, finalTime, switchT, y0, preInd, uInd, theta, tsamps, plot and flag. Values for each key are empty lists [].

2.2 checkStructSimul(model_def, simul_def)

Inputs: Dictionary containing the model information and dictionary with the keys specified in defSimulStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defSimulStruct()). If everything is correct, then some processing of the values will be done (mostly indexing and extraction of elements to ease further indexing of structures). If the value for some key has a wrong structure or content, the function will break (dictionary not returned) and a message will be printed to help the user identify where the issue is (mostly information about the key that had not passed the check).
Outputs: Same dictionary introduced with necessary modifications of some fields to aid it's use in further sections. If the value for any key is wrong, nothing will be returned.

2.3 fileStructInfo()

Inputs: None
Function: This function prints in the console information about the structure the CSV files need to have in order to be given to BOMBS so experimental details are extracted from it.
Outputs: None

2.4 defSimulStructFiles()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the model simulation section (if experimental details are given with CSV files) so only the values for each key have to be filled.

Outputs: Dictionary with keys ObservablesFile, EventInputsFile, theta, MainDir, plot, flag. Values for each key are empty lists [].

2.5 extractSimulCSV(model_def, simul_def)

Inputs: Dictionary with the keys specified in defSimulStructFiles() and filled values for each one.
Function: This function checks that the CSV files introduced exist and if so, proceeds to extract all the necessary information to populate the values of the dictionary structure defined in defSimulStruct.
Outputs: Dictionary with the structure defined in defSimulStruct where all the values for each key have been extracted from the CSV files.

2.6 plotSimsODE(simuls,model_def,simul_def)

Inputs: Simulation results, model definition and simulation definition dictionaries.
Function: This function generates and saves (in the results directory) the plots for the simulations done by the user. A separate plot for each experiment will be generated, where there will be a subplot for each state and each inducer of the system.
Outputs: None

2.7 simulateODEs(model_def, simul_def)

Inputs: Model definition and simulation definition dictionaries.
Function: This is the main function of the section, which takes all the information from the model and simulation structures, simulates the ODEs system, saves the simulation results in the results folder and generates the plots if the user has selected the option.
Outputs: Simulation results dictionary plus model definition and simulation definition dictionaries with the savepath key pointing to the saved files. The simulation definition dictionary has the path and file name split in savepath and savename.

3 Pseudo-Data Generation

3.1 defPseudoDatStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the model pseudo-data generation section so only the values for each key have to be filled.
Outputs: Dictionary with keys Nexp, finalTime, switchT, y0, preInd, uInd, theta, tsamps, plot, flag, Obs and Noise. Values for each key are empty lists [].

3.2 checkStructPseudoDat(model_def, pseudo_def)

Inputs: Dictionary containing the model information and dictionary with the keys specified in defPseudoDatStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defPseudoDatStruct()). If everything is correct, then some processing of the values will be done (mostly indexing and extraction of elements to ease further indexing of structures). If the value for some key has a wrong structure or content, the function will break (dictionary not returned) and a message will be printed to help the user identify where the issue is (mostly information about the key that had not passed the check).
Outputs: Same dictionary introduced with necessary modifications of some fields to aid it's use in further sections. If the value for any key is wrong, nothing will be returned.

3.3 defPseudoDatStructFiles()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the model pseudo-data generation section (if experimental details are given with CSV files) so only the values for each key have to be filled.
Outputs: Dictionary with keys ObservablesFile, EventInputsFile, theta, MainDir, plot, flag, Obs and Noise. Values for each key are empty lists [].

3.4 extractPseudoDatCSV(model_def, pseudo_def)

Inputs: Dictionary with the keys specified in defPseudoDatStructFiles() and filled values for each one.
Function: This function checks that the CSV files introduced exist and if so, proceeds to extract all the necessary information to populate the values of the dictionary structure defined in defPseudoDatStruct.
Outputs: Dictionary with the structure defined in defPseudoDatStruct where all the values for each key have been extracted from the CSV files.

3.5 PDatCSVGen(pseudo_res,model_def,pseudo_def)

Inputs: pseudo-data results, model definition and pseudo-data definition filled structure dictionaries.
Function: This function generates CSV files for each experiment simulated containing all the necessary information for the pseudo-data. Three files for each experiment are generated:

- Simulations: CSV file with the time vector and the simulation for all states for each theta given.
- Observables: CSV file with the time vector and the pseudo-data generated for each observable of the system (one trace for each theta vector introduced).
- Event_Inputs: CSV file containing information about the inputs and change over time (similar to the structure explained in fileStructInfo()).

Outputs: None (but CSV files generated).

3.6 plotPseudoDatODE(pseudo_res,model_def,pseudo_def)

Inputs: pseudo-data results, model definition and pseudo-data definition dictionaries.
Function: This function generates and saves (in the results directory) the plots for the pseudo-data generated by the user. A separate plot for each experiment will be generated, where there will be a subplot for each observable and each inducer of the system.
Outputs: None

3.7 GenPseudoDat(model_def, pseudo_def)

Inputs: Model definition and pseudo-data definition dictionaries.
Function: This is the main function of the section, which takes all the information from the model and pseudo-data structures (information about the experiments and osbservabels), simulates the ODEs system, generates pseudo-data for the observables, saves the simulation and pseudo-data results in the results folder and generates the plots if the user has selected the option.
Outputs: Pseudo-data results dictionary plus model definition and pseudo-data definition dictionaries with the savepath key pointing to the saved files. The pseudo-data definition dictionary has the path and file name split in savepath and savename.

4 Maximum Likelihood Estimation

4.1 defMLEStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the maximum likelihood estimation section so only the values for each key have to be filled.
Outputs: Dictionary with keys Nexp, finalTime, switchT, y0, preInd, uInd, tsamps, plot, flag, thetaMAX, thetaMIN, runs, parallel, DataMean, DataError, Obs, OPTsolver, MaxTime, MaxFuncEvals. Values for each key are empty lists [].

4.2 SimToMle(mle_def, simul_def)

Inputs: Dictionaries with 2 separate structures from one of the package sections sharing one or more keys.
Function: Function that extracts all the values of the second that share keys with the first dictionary and places them in it. Even though in here we state as first dictionary mle_def and as second simul_def, this can be any two dictionaries (it is to follow the structure in the notebook examples).
Outputs: First dictionary (in this case mle_def) with the values of the second in any shared key.

4.3 checkStructMLE(model_def, mle_def)

Inputs: Dictionary with the model structure and dictionary with the keys specified in defMLEStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defMLEStruct()). If everything is correct, then some processing of the values will be done (mostly indexing and extraction of elements to ease further indexing of structures). If the value for some key has a wrong structure or content, the function will break (dictionary not returned) and a message will be printed to help the user identify where the issue is (mostly information about the key that had not passed the check).
Outputs: Same dictionary introduced with necessary modifications of some fields to aid it's use in further sections. If the value for any key is wrong, nothing will be returned.

4.4 selectObsSim_te(simul, Obs, stName)

Inputs: Simulation of one experiment, observables of the system (as vector of strings) and the state names of the model (as vector of strings).
Function: This function automatically extracts the observables of your model for a given simulation. Operations between states or others are allowed introduced in the input Obs.
Outputs: Matrix containing the simulation of an experiment for the observables specified for the system.

4.5 restructInputs_te(model_def, mle_def, expp)

Inputs: Dictionaries containing the information for the model and the MLE and index of the experiment (from mle_def) that you want to re-structure the inputs vector.
Function: This function takes the input values for a specified experiment (expp) and restructures the matrix into a single vector so the package can use it. This vector will group all the inputs for a specified step together. It is highly probable that you won't need to use this function.
Outputs: Vector containing the re-structured inputs. For example, if you have 2 inputs (inp1, inp2) and a 3 steps experiment, the vector will be organised as: inp1_stp1, inp2_stp1, inp1_stp2, inp2_stp2, inp1_stp3, inp2_stp3.

4.6 UVloglike(dats, mes, errs)

Inputs: Vector or matrix with experimental data, simulation (of one observable) and errors for your experimental data.
Function: This function computes the univariate Gaussian Log-Likelihood value between your data and one observable simulation. Sum across timepoints will be computed.
Outputs: Log-likelihood value for a specific observable.

4.7 MVloglike(dats, mes, errs)

Inputs: Vector or matrix with experimental data, simulation (of one observable) and array of co-variances for the data.
Function: This function computes the multi-variate Gaussian Log-Likelihood value between your data and one observable simulation. Note that a small number (0.1) is added in the diagonal of the co-variance matrix to ensure that this is positive definite. Also, since the determinant of these can get

extremely high or low, Infs have been set to the numerical limit of Julia (1e300).
Outputs: Log-likelihood value for a specific observable.

4.8 plotMLEResults(mle_res,model_def,mle_def)

Inputs: MLE results, model definition and MLE definition dictionaries.
Function: This function generates and saves (in the results directory) the plots for the maximum likelihood estimation results (convergence plots and simulations against data plots). A separate plot for each experiment will be generated, where there will be a subplot for each observable and each inducer of the system.
Outputs: None

4.9 defCrossValMLEStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the cross-validation of the maximum likelihood estimation results so only the values for each key have to be filled.
Outputs: Dictionary with keys Nexp, finalTime, switchT, y0, preInd, uInd, tsamps, plot, flag, thetaM, DataMean, DataError, Obs. Values for each key are empty lists [].

4.10 checkStructCrossValMLE(model_def, cvmle_def)

Inputs: Dictionary with the model structure and dictionary with the keys specified in defCrossValMLEStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defCrossValMLEStruct()). If everything is correct, then some processing of the values will be done (mostly indexing and extraction of elements to ease further indexing of structures). If the value for some key has a wrong structure or content, the function will break (dictionary not returned) and a message will be printed to help the user identify where the issue is (mostly information about the key that had not passed the check).
Outputs: Same dictionary introduced with necessary modifications of some fields to aid it's use in further sections. If the value for any key is wrong, nothing will be returned.

4.11 plotCrossValMLEResults(cvmle_res,model_def,cvmle_def, simul_def)

Inputs: cross-validation of MLE results, model definition, cross-validation of MLE definition and simulations dictionaries.
Function: This function generates and saves (in the results directory) the plots for the cross-validation of the maximum likelihood estimation results . A separate plot for each experiment will be generated, where there will be a subplot for each observable and each inducer of the system.
Outputs: None

4.12 CrossValMLE(model_def, cvmle_def)

Inputs: Model definition and cross-validation of MLE results definition dictionaries.
Function: This is the main function to run the cross-validation of the MLE results, which takes all the information from the model and cross-validation structures (information about the experiments and osbservabels), simulates the ODEs system, computes the log-likelihood for each theta sample given, selects the best theta vector having seen the new set of data and generates the plots if the user has selected the option.
Outputs: Cross-validation results dictionary plus model definition and cross-validation definition dictionaries with the savepath key pointing to the saved files. The cross-validation definition dictionary has the path and file name split in savepath and savename.

4.13 finishMLEres(mle_res, model_def, mle_def)

Inputs: MLE results, model definition and MLE definition dictionaries.
Function: This function saves the MLE results and plots in the common BOMBS structure. This function might be needed if parallelisation is desired since in that case the user needs to set the optimisation (not fully automated yet).
Outputs: MLE results dictionary plus model definition and MLE definition dictionaries with the savepath key pointing to the saved files. The MLE definition dictionary has the path and file name split in savepath and savename.

4.14 MLEtheta(model_def, mle_def)

Inputs: Model definition and maximum likelihood estimation definition dictionaries.

Function: This is the main function of the section, which takes all the information from the model and MLE structures (information about the experiments and osbservabels), generates the necessary scripts (cost-function related), runs the optimisation (for as many runs as the user has selected and in parallel if selected), extracts the best results and generates the plots if the user has selected the option.
Outputs: MLE results dictionary plus model definition and MLE definition dictionaries with the savepath key pointing to the saved files. The MLE definition dictionary has the path and file name split in savepath and savename.

5 Stan Inference of Parameters

5.1 defBayInfStruct()

Inputs: None
Function:
Outputs:

5.2 defBayInfDataStruct()

Inputs: None
Function:
Outputs:

5.3 defBayInfDataFromFilesStruct()

Inputs: None
Function:
Outputs:

5.4 defBasicStanSettingsStruct()

Inputs: None
Function:
Outputs:

5.5 convertBoundTo2(x, bo, up)

Inputs:
Function:
Outputs:

5.6 fitPriorSamps(priorsamps, model_def)

Inputs:
Function:
Outputs:

5.7 fitPriorSampsMultiNorm(priorsamps, model_def)

Inputs:
Function:
Outputs:

5.8 checkStructBayInf(model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.9 checkStructBayInfData(model_def, data_def)

Inputs:
Function:
Outputs:

5.10 checkStructBayInfDataFiles(model_def, data_def)

Inputs:
Function:
Outputs:

5.11 checkStructBayInfStanSettings(model_def, stan_def)

Inputs:
Function:
Outputs:

5.12 genStanInitDict(samps, names, chains)

Inputs:
Function:
Outputs:

5.13 reparamDictStan(standict, bayinf_def)

Inputs:
Function:
Outputs:

5.14 genStanModel(model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.15 restructureDataInference(model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.16 getStanInferenceElements(model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.17 saveStanResults(rc, chns, cnames, model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.18 runStanInference(model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.19 plotStanResults(staninf_res, model_def, bayinf_def)

Inputs:
Function:
Outputs:

5.20 StanInfer(model_def, bayinf_def)

Inputs:
Function:
Outputs:

6 Entropy Approximation

6.1 genSamplesPrior(model_def, bayinf_def, nsamps, mu,coo)

Inputs:
Function:
Outputs:

6.2 H_Upper(w,E)

Inputs:
Function:
Outputs:

6.3 mvGauss(x, MU, E)

Inputs:
Function:
Outputs:

6.4 H_Lower(w, E, MU)

Inputs:
Function:
Outputs:

6.5 GaussMix(x, MU, E, w)

Inputs:
Function:
Outputs:

6.6 ZOTSE(MU, E, w)

Inputs:
Function:
Outputs:

6.7 GaussMix2(x)

Inputs:
Function:
Outputs:

6.8 FMix(x, MU, E, w)

Inputs:
Function:
Outputs:

6.9 SOTSE(MU, E, w)

Inputs:
Function:
Outputs:

6.10 computeH(sampl, model_def, tag)

Inputs:
Function:
Outputs:

6.11 computeHgain(prior, posterior, model_def, tag)

Inputs:
Function:
Outputs:

7 Optimal Experimental Design for Model Selection

7.1 defODEModelSelectStruct()

Inputs: None
Function:
Outputs:

7.2 checkStructOEDMS(oedms_def)

Inputs:
Function:
Outputs:

7.3 BhattacharyyaDist(mu1, mu2, sd1, sd2)

Inputs:
Function:
Outputs:

7.4 EuclideanDist(sm1, sm2)

Inputs:
Function:
Outputs:

7.5 genOptimMSFuncts(oedms_def)

Inputs:
Function:
Outputs:

7.6 plotOEDMSResults(oedms_res, oedms_def)

Inputs:
Function:
Outputs:

7.7 settingsBayesOpt(oedms_def)

Inputs:
Function:
Outputs:

7.8 mainOEDMS(oedms_def)

Inputs:
Function:
Outputs:

8 Optimal Experimental Design for Model Calibration

8.1 defODEModelCalibrStruct()

Inputs: None
Function:
Outputs:

8.2 checkStructOEDMC(oedmc_def)

Inputs:
Function:
Outputs:

8.3 genOptimMCFuncts(oedmc_def)

Inputs:
Function:
Outputs:

8.4 plotOEDMCResults(oedmc_res, oedmc_def)

Inputs:
Function:
Outputs:

8.5 settingsBayesOptMC(oedmc_def)

Inputs:
Function:
Outputs:

8.6 mainOEDMC(oedmc_def)

Inputs:
Function:
Outputs:

9 Others

9.1 printLogo()

Inputs: None
Function:
Outputs: None

9.2 versionBOMBS()

Inputs: None
Function:
Outputs: None

9.3 infoAll(woo)

Inputs:
Function:
Outputs: None
Prints:

Please, if you want information about one of the package structures, type:
1) "model" for the model generation section
2) "simulation", "simulation" or "simul" for the model simulation section
3) "pseudo-data", "pseudodata", "pseudo data" for the model pseudo-data generation section
4) "mle", "likelihood" for the maximum likelihood estimation section 5) "inference", "stan", "stan inference", "staninference" for the Bayesian parameter inference section
6) "oedms", "model selection", "modelselection", "oed model selection" for the optimal experimental design for model selection section
7) "oedmc", "model calibration", "modelcalibration", "oed model calibration" for the optimal experimental design for model calibration section

"model" –>
CALL defModStruct()
model_def["NameF"] = [];
String containing the name of the model. Scripts and results will be stored

using this name

model_def["nStat"] = [];
Integer indicating the total number of steps of the model.

model_def["nPar"] = [];
Integer indicating the total number of parameters of the model.

model_def["nInp"] = [];
Integer indicating the total number of stimuli (inducers) of the model

model_def["stName"] = [];
Vector of strings indicating the name of all the states of the model (without a d letter in front).

model_def["parName"] = [];
Vector of strings indicating the name of all the parameters of the model.

model_def["inpName"] = [];
Vector of strings indicating the name of all the stimuli (inducers) of the model

model_def["eqns"] = [];
Vector of strings containing all the equations for the model (left and right-hand sides).
If an equation represents a state, the left-hand side has to be one of the strings contained in stName but with a d in front (example: Prot -> dProt = ...).
Equations that are not states of ODEs are also allowed. Same as Julia expressions (println, for, if, etc.)
If you want to include a condition (if) for a state variable, each one of the if statement has to be written in a separate string. Be careful with this, since then your Stan model will not work. To make it work you have to write it in Stan language, but then the Julia code will not work. We recommend that if so, first generate the stan code and then apply all the necessary modifications there. However, if you do not care for the Julia code and just want the Stan code go on. Just know that in this case, if there is a condition in one of the states, in stan you need to type the full if statement in one same string.

model_def["Y0eqs"] = [];
Vector of strings containing the steady-state equations of the model if desired (if not, just leave it as an empty vector). These equations will be used

to compute y0 assuming steady-state reached before the experiment.
All the states must appear (identified in the left-hand side, but this time without the d in front), however other equations are allowed.
If some element of the equation requires an experimental value for the calculation, please add exp at the beginning of the state (example: Cmrna -> expCmrna).
Please, do not use the name alp for anything, since this is reserved.

model_def["Y0Sim"] = [];
If analytical solution of the model at steady state is not accurate enough and you want to add an Over-Night simulation before the actual experiment simulation.
Allowed vales are true, false, "Yes", "yes", "No", "no".
Default value is false.
Time-scale for the simulation is assumed in minutes (1440 min). If this wants to be changed, what should be introduced here is a number for the time conversion (e.g. 1/60 if to convert to days or 60 if to convert to seconds)

model_def["tols"] = [];
Vector of 2 floats containing the relative and absolute tolerances for the solver (in this order). If left empty, 1e-6 will be assumed for both.

model_def["solver"] = [];
IVP solver to solve the ODEs. If nothing specified, the default will be Tsit5(). For more info check `https://diffeq.sciml.ai/v2.0/tutorials/ode_example.html`

"simul" –>

CALL defSimulStruct()
MAIN STRUCTURE

simul_def["Nexp"] = [];
Integer indicating the number of experiments to be simulated

simul_def["finalTime"] = [];
Vector of final times for each simulation (initial time will always be assumed as 0, so please consider that).

simul_def["switchT"] = [];

Array with the switching times of the inducer in the simulation (time 0 and final time need to be considered)

simul_def["y0"] = [];
Array (single simulation) or matrix (multiple simulations) of Y0s for the simulations for each experiment. If you are computing the steady-state this vector might not be used, however, you still need to introduce it with some random numbers.

simul_def["preInd"] = [];
Vector of numbers with the values for the stimuli (inducer) in the over-night. It might be the case that this entry is not required since only the y0 vector is considered for the initial point of the simulation. However, you still need to introduce a random value for it to avoid future issues.

simul_def["uInd"] = [];
Array containing the values for the stimuli at each step for each experiment. In each sub-array, columns indicate a step and rows indicate an inducer (if multiple ones are considered).

simul_def["theta"] = [];
Vector/Matrix with the parameter samples or directory and file location of CSV file with them.

simul_def["tsamps""] = [];
Array of sampling time vectors for the experiments.

simul_def["plot"] = [];
Boolean or yes/no string to save the resulting simulations in the results directory (false will be considered as default).

simul_def["flag"] = [];
String to attach a unique flag to the generated scripts and result files so they are not overwritten. If empty, nothing will be added.

CALL defSimulStructFiles()

STRUCTURE IF CSV FILES ARE USED

simul_def["ObservablesFile"] = [];
Vector of strings containing the name of the files that have the information

about the sampling times and y0 values.

simul_def["EventInputsFile"] = [];
Vector of strings containing the name of all the files that have the information about the stimuli and events of the experiment.

simul_def["theta"] = [];
Vector/Matrix with the parameter samples or directory and file location of CSV file with them.

simul_def["MainDir"] = [];
Main directory path for the files in ObervablesFile and EventInputsFile.

simul_def["plot"] = [];
Boolean or yes/no string to save the resulting simulations in the results directory (false will be considered as default).

simul_def["flag"] = [];
String to attach a unique flag to the generated scripts and result files so they are not overwritten. If empty, nothing will be added.

"pseudodata" –>

CALL defPseudoDatStruct()

MAIN STRUCTURE

pseudo_def["Nexp"] = [];
Integer indicating the number of experiments to be simulated

pseudo_def["finalTime"] = [];
Vector of final times for each simulation (initial time will always be assumed as 0, so please consider that).

pseudo_def["switchT"] = [];
Array with the switching times of the inducer in the simulation (time 0 and final time need to be considered)

pseudo_def["y0"] = [];
Array (single simulation) or matrix (multiple simulations) of Y0s for the

simulations for each experiment. If you are computing the steady-state this vector might not be used, however, you still need to introduce it with some random numbers.

pseudo_def["preInd"] = [];
Vector of numbers with the values for the stimuli (inducer) in the over-night. It might be the case that this entry is not required since only the y0 vector is considered for the initial point of the simulation. However, you still need to introduce a random value for it to avoid future issues.

pseudo_def["uInd"] = [];
Array containing the values for the stimuli at each step for each experiment. In each sub-array, columns indicate a step and rows indicate an inducer (if multiple ones are considered).

pseudo_def["theta"] = [];
Vector/Matrix with the parameter samples or directory and file location of CSV file with them.

pseudo_def["tsamps"] = [];
Array of sampling time vectors for the experiments.

pseudo_def["plot"] = [];
Boolean or yes/no string to save the resulting simulations in the results directory (false will be considered as default).

pseudo_def["flag"] = [];
String to attach a unique flag to the generated scripts and result files so they are not overwritten. If empty, nothing will be added.

pseudo_def["Obs"] = [];
States of the model that are observables. This is either a vector of strings, a vector of integers indicating which entries from model_def["stName"] are observables. If a vector of strings is given, these could also be an expression combining states (Only +,-,*,/ and ŵill be considered).

pseudo_def["Noise"] = [];
Percentage of heteroscedastic noise (introduced as value from 0 to 1). If empty 10% will be assumed. This has to be a vector of noise values for each observable.

CALL defPseudoDatStructFiles()

STRUCTURE IF CSV FILES ARE USED

pseudo_def["ObservablesFile"] = [];
Vector of strings containing the name of the files that have the information about the sampling times and y0 values.

pseudo_def["EventInputsFile"] = [];
Vector of strings containing the name of all the files that have the information about the stimuli and events of the experiment.

pseudo_def["theta"] = [];
Vector/Matrix with the parameter samples or directory and file location of CSV file with them.

pseudo_def["MainDir"] = [];
Main directory path for the files in ObervablesFile and EventInputsFile.

pseudo_def["plot"] = [];
Boolean or yes/no string to save the resulting simulations in the results directory (false will be considered as default).

pseudo_def["flag"] = [];
String to attach a unique flag to the generated scripts and result files so they are not overwritten. If empty, nothing will be added.

pseudo_def["Obs"] = [];
States of the model that are observables. This is either a vector of strings, a vector of integers indicating which entries from model_def["stName"] are observables. If a vector of strings is given, these could also be an expression combining states (Only +,-,*,/ and ŵill be considered).

pseudo_def["Noise"] = [];
Percentage of heteroscedastic noise (introduced as value from 0 to 1). If empty 10% will be assumed. This has to be a vector of noise values for each observable.

"mle" –>

CALL defMLEStruct()

mle_def["Nexp"] = [];
Integer indicating the number of experiments to be simulated

mle_def["finalTime"] = [];
Vector of final times for each simulation (initial time will always be assumed as 0, so please consider that).

mle_def["switchT"] = [];
Array with the switching times of the inducer in the simulation (time 0 and final time need to be considered)

mle_def["y0"] = [];
Array (single simulation) of Y0s for the simulations for the experiment. If you are computing the steady-state this vector might not be used, however, you still need to introduce it with some random numbers.

mle_def["preInd"] = [];
Vector of numbers with the values for the stimuli (inducer) in the over-night. It might be the case that this entry is not required since only the y0 vector is considered for the initial point of the simulation. However, you still need to introduce a random value for it to avoid future issues.

mle_def["uInd"] = [];
Array containing the values for the stimuli at each step for each experiment. In each sub-array, columns indicate a step and rows indicate an inducer (if multiple ones are considered).

mle_def["tsamps"] = [];
Array of sampling time vectors for the experiments.

mle_def["plot"] = [];
Boolean or yes/no string to save the resulting simulations in the results directory (false will be considered as default).

mle_def["flag"] = [];
String to attach a unique flag to the generated scripts and result files so they are not overwritten. If empty, nothing will be added.

mle_def["thetaMAX"] = [];
Vector containing the maximum bounds for theta (no files can be introduced)

mle_def["thetaMIN"] = [];
Vector containing the minimum bounds for theta (no files can be introduced)

mle_def["runs"] = [];
Integer indicating how many runs of Optimisation will be done. You will get as many theta vectors as runs selected.

mle_def["parallel"] = [];
Boolean or yes/no string indicating if the different runs want to be done in parallel (true) or series (false). Default is false.

For the two following fields, you can introduce a string pointing to the observable files (same strings in) both fields having the same structure as the ones generated in the PseudoData section. If multiple theta are considered in the file, then the covariance matrix will be taken.
mle_def["DataMean"] = [];
Array containing the vector of means for each experiment.

mle_def["DataError"] = [];
Array containing the vector or matrices (covariance included) of errors for the data for each experiment.

IMPORTANT!!!!

Whilst each entry (experiment) of DataMean can be a matrix where each column is an observable of the system, for DataError this is not the case. Each entry of the array (experiment) will have as many entries as observables, where it would be a vector of errors or a matrix. This is done this way to generalise the presence of both options.

mle_def["Obs"] = [];
States of the model that are observables. This is either a vector of strings, a vector of integers indicating which entries from model_def["stName"] are observables. If a vector of strings is given, these could also be an expression combining states (Only +,-,*,/ and ŵill be considered).

mle_def["OPTsolver"] = [];

For now we only use the package BlackBoxOptim, so any of their options can be used. The default is adaptive_de_rand_1_bin_radiuslimited. Please, introduce it as a string.

mle_def["MaxTime"] = [];
Integer indicating the maximum number of time allowed for the optimisation as a stop criterion. If this is selected MaxFuncEvals has to be empty (only 1 stop criterion). If both are empty, the default will be to do 1000 function evaluations

mle_def["MaxFuncEvals"] = [];
Integer indicating the maximum number of function evaluations allowed for the optimisation as a stop criterion. If this is selected MaxTime has to be empty (only 1 stop criterion). If both are empty, the default will be to do 1

"inference" –>

"oedms" –>

"oedmc" –>