BOMBS Functions
Version 0.1.0

DAVID GOMEZ-CABEZA

David.Gomez@ed.ac.uk
February 5, 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: This function allows the user to obtain the dictionary structure with all the keys for the Bayesian inference section (using Stan) so only the values for each key have to be filled.
Outputs: Dictionary with keys Priors, Data, StanSettings, flag, plot, runInf and MultiNormFit. Values for each key are empty lists [].

5.2 defBayInfDataStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for data structure necessary for the Bayesian inference section so only the values for each key have to be filled.
Outputs: Dictionary with keys Nexp, finalTime, switchT, y0, preInd, uInd, tsamps, Obs, DataMean and DataError. Values for each key are empty lists [].

5.3 defBayInfDataFromFilesStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for data structure introduced with CSV files necessary for the Bayesian inference section so only the values for each key have to be filled.
Outputs: Dictionary with keys Obs, Observables, Inputs and y0. Values for each key are empty lists [].

5.4 defBasicStanSettingsStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for Stan settings structure necessary for the Bayesian inference section so only the values for each key have to be filled.
Outputs: Dictionary with keys cmdstan_home, nchains, nsamples, nwarmup, printsummary, init, maxdepth, adaptdelta and jitter. Values for each key are empty lists [].

5.5 convertBoundTo2(x, bo, up)

Inputs: univariate samples for a parameter (x), lower (bo) and upper (up) bounds for the range to re-map the samples to.
Function: This function takes vector of samples for a parameter and re-maps it's values to a different numeric region specified by the upper and lower bounds.
Outputs: re-mapped samples to the numeric region specified.

5.6 fitPriorSamps(priorsamps, model_def)

Inputs: samples for a parameter or set of parameters and model definition structure.
Function: This function takes all the samples for each parameter in the model and fits a series of distributions (Beta, Exponential, LogNormal, Normal, Gamma, Laplace, Pareto, Rayleigh, Cauchy, Uniform) selecting the one with better likelihood. Re-parameterisation (in Stan meaning) of the distributions to be in the -2,2 region will be attempted.
Outputs: Dictionary with three entries (pars, transpars, pridis) with strings containing the information of the fitted priors in Stan format.

5.7 fitPriorSampsMultiNorm(priorsamps, model_def)

Inputs: samples for a parameter or set of parameters and model definition structure.
Function: This function takes all the samples for each parameter in the model and fits a Normal, LogNormal and Uniform distribution selecting the one with better likelihood. In this case, Normal and LogNormal distributions will result in a multi-variate Normal distribution for the prior fit. Re-parameterisation (in Stan meaning) of the distributions to be in the -2,2 region will be attempted.
Outputs: Dictionary with three entries (pars, transpars, pridis) with strings containing the information of the fitted priors in Stan format.

5.8 checkStructBayInf(model_def, bayinf_def)

Inputs: Dictionary with the model structure and dictionary with the keys specified in defBayInfStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced

(structure from defBayInfStruct()). 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.

5.9 checkStructBayInfData(model_def, data_def)

Inputs: Dictionary with the model structure and dictionary with the keys specified in defBayInfDataStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defBayInfDataStruct()). 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.

5.10 checkStructBayInfDataFiles(model_def, data_def)

Inputs: Dictionary with the model structure and dictionary with the keys specified in defBayInfDataFromFilesStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defBayInfDataFromFilesStruct()). 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.

5.11 checkStructBayInfStanSettings(model_def, stan_def)

Inputs: Dictionary with the model structure and dictionary with the keys specified in defBasicStanSettingsStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defBasicStanSettingsStruct()). 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.

5.12 genStanInitDict(samps, names, chains)

Inputs: samples for the parameters of your model, names for each parameter and number of chains that the inference will have.
Function: This function generates a dictionary structure for each chain with a sample for the parameters in the structure Stan requires so it can be given as initial guess for the inference.
Outputs: Array of dictionaries for the Stan chains initial guesses.

5.13 reparamDictStan(standict, bayinf_def)

Inputs: Stan initial guess array of dictionaries and bayesian inference main structure.
Function: This function computes a necessary reparametrisation (in Stan meaning) of the initial guess samples for the chains if reparameterisation of priors is required.
Outputs: Array of dictionaries for the Stan chains initial guesses with correct reparameterisation (re-mapping).

5.14 genStanModel(model_def, bayinf_def)

Inputs: model definition and Bayesian inference definition structures with filled values.
Function: This function takes all the necessary information of your model and generates a working Stan script to perform inference. Model ODEs

and steady state equations (if given) functions are generated, as well as all the required sections in a Stan script. High flexibility in definition of priors is allowed (see function infoAll or associated jupyter notebook for more information).
Outputs: Bayesian inference definition structure with the additional field ModelPath containing the path to the Stan script generated.

5.15 restructureDataInference(model_def, bayinf_def)

Inputs: model definition and Bayesian inference definition structures with filled values.
Function: This function re-structures all the information about the experimental data into a dictionary that Stan will understand as the data structure for the inference.
Outputs: dictionary containing the data and data structure for the Stan inference.

5.16 getStanInferenceElements(model_def, bayinf_def)

Inputs: model definition and Bayesian inference definition structures with filled values.
Function: This function checks the Stan settings selected (if any, defaults can be used) and processes all the contents for a Stan inference, however this will not be run, either due to lack of enough elements in the structures or because the user has selected not to run it.
Outputs: path to the Stan model (modelpath), Stan model generated (Model), structure containing all the stan settings and hyper-parameters (StanModel), data structure for the inference (inferdata), initial guesses for the chains if any (init) and model definition and Bayesian inference definition structures (model_def, bayinf_def).

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

Inputs: the 3 function outputs from a Stan inference call (containing inference results) and model definition and Bayesian inference definition structures with filled values.
Function: This function will allow you to save any Stan results run outside the package in the same structure (and generating the same plots) than if you run the inference from the package.

Outputs: matrix containing the posterior for parameters.

5.18 runStanInference(model_def, bayinf_def)

Inputs: model definition and Bayesian inference definition structures with filled values.
Function: Similar to getStanInferenceElements() but in this case inference will be run and results will be saved.
Outputs: stan inference results structure and model definition and bayesian inference definition structures with the additional fields containing scripts genretaed paths.

5.19 plotStanResults(staninf_res, model_def, bayinf_def)

Inputs: Bayesian inference results, model definition and Bayesian inference definition structures with filled values.
Function: This function generates and saves (in the results directory) the plots for the bayesian parameter inference 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. A bi-variate plot for the parameters will also be generated.
Outputs: None.

5.20 StanInfer(model_def, bayinf_def)

Inputs: model definition and Bayesian inference definition structures with filled values.
Function: This function is the main core to assess if structures provided by the user contain enough information to run inference or not and proceed calling the rest of the section functions.
Outputs: stan inference results structure and model definition and bayesian inference definition structures with the additional fields containing scripts genretaed paths.

6 Entropy Approximation

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

Inputs: model definition and Bayesian inference definition structures, number of samples wanted to be generated and mean and covariance if the sampling comes from a multivariate Normal distribution.
Function: This function reads the prior structure introduced in the bayinf_def structure and generates a sampling on the distributions specified automatically.
Outputs: matrix with the parameter samples generated.

6.2 H_Upper(w,E)

Inputs: weigts and covariances for the Gaussian mixture.
Function: This function computes an upper bound for the entropy approximation performed as a check. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: Upper bound for the Entropy approximation.

6.3 mvGauss(x, MU, E)

Inputs: sample, means and covariances for the Gaussian mixture.
Function: probability density function for a multivariate Gaussian. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs:

6.4 H_Lower(w, E, MU)

Inputs: weights, covariances and means for a Gaussian mixture.
Function: This function computes a lower bound for the entropy approx-

imation performed as a check. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: Lower bound for the Entropy approximation.

6.5 GaussMix(x, MU, E, w)

Inputs: sample, means, covariances and weights for a Gaussian mixture.
Function: This function computes the probability density function for a Gaussian mixture. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: probability of the Gaussian mixture for a given sample.

6.6 ZOTSE(MU, E, w)

Inputs: means, covariances and weights for a Gaussian mixture.
Function: This function computes the zero-order Taylor series expansion for a Gaussian mixture. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: Zero-order Taylor series expansion value for a Gaussian mixture.

6.7 GaussMix2(x)

Inputs: samples for a Gaussian mixture model.
Function: This function is used to compute the slope of the Gaussian mixture PDF for a given sample. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference

on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: probability of the Gaussian mixture for a given sample.

6.8 FMix(x, MU, E, w)

Inputs: samples, means covarainces and weights for a Gaussian mixture model.
Function: This function is used to compute the most computationally expensive part of the second order Taylor series expansion. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: Most computationally expensive part of the second order Taylor series expansion

6.9 SOTSE(MU, E, w)

Inputs: Means, covariances and weights for a Gaussian mixture model.
Function: This function computes the second-order Taylor series expansion for a Gaussian mixture. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
This function is not exported with the package.
Outputs: Second-order Taylor series expansion value for a Gaussian mixture.

6.10 computeH(sampl, model_def, tag)

Inputs: matrix containing the samples for the model parameters, model definition structure and string used to not over-write past files.
Function: This function computes the entropy approximation for the samples given using all the above functions. For more information, please check M. F. Huber, T. Bailey, H. Durrant-Whyte and U. D. Hanebeck, "On entropy

approximation for Gaussian mixture random vectors," 2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, 2008, pp. 181-188, doi: 10.1109/MFI.2008.4648062.
Outputs: Entropy approximation results for the given samples.

6.11 computeHgain(prior, posterior, model_def, tag)

Inputs: Prior (as samples or as Stan definition strings as used in the bayinf_def structure) and posterior samples, model definition structure and string used to not over-write past files.
Function: This function automatically computes the entropy approximation for prior and posterior and also gives the difference between the two (entropy information gain).
Outputs: Dictionary structure containing all the entropy approximations and information gain computed.

7 Optimal Experimental Design for Model Selection

7.1 defODEModelSelectStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the OED for model selection section so only the values for each key have to be filled.
Outputs: Dictionary with keys Model_1, Model_2, Obs, Theta_M1, Theta_M2, y0_M1, y0_M2, preInd_M1, preInd_M2, finalTime, switchT, tsamps, equalStep, fixedInp, fixedStep, plot, flag, uUpper, uLower and maxiter. Values for each key are empty lists [].

7.2 checkStructOEDMS(oedms_def)

Inputs: Dictionary with OED for model selection dictionary with the keys specified in defODEModelSelectStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defODEModelSelectStruct()). 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.

7.3 BhattacharyyaDist(mu1, mu2, sd1, sd2)

Inputs: mean vector for the first and second distributions and covariance matrices for the first and second distributions.
Function: This function computes the Bhattacharyya distance between first and second distribution given (closed at the second moment). Note that each distribution represents a full simulation for an observable where mean and covariance have been computed considering all the time points.
Outputs: Bhattacharyya distance value.

7.4 EuclideanDist(sm1, sm2)

Inputs: Simulation for model 1 and 2 (only one theta vector considered).
Function: This function computes the Euclidean distance between two different simulations (same observable but different models) across all the sampling time points.
Outputs: Euclidean distance value.

7.5 genOptimMSFuncts(oedms_def)

Inputs: OED for model selection structure dictionary with all the filled values for each key.
Function: This function extract all the information about the models and how the experiment will be designed (sampling times, switching times, observables, etc.) and generated the Utility/Cost function script for the optimisation.
Outputs: OED for model selection structure dictionary with some modifications in some key values to ease script generation (in observables) and checks (In this case, the check of the models, model scripts generation and check of the dictionaries is done in here).

7.6 plotOEDMSResults(oedms_res, oedms_def)

Inputs: OED for model selection results and OED for model selection definition structures with filled values.
Function: This function generates and saves (in the results directory) the plots for the Bayesian OED for model selection results. A plot for the designed experiment will be generated, where there will be a subplot for each observable and each inducer of the system. The convergence plot for the optimisation will also be generated.
Outputs: None.

7.7 settingsBayesOpt(oedms_def)

Inputs: OED for model selection definition structure with filled values.
Function: This function sets the bounds for the different inputs, utility/cost function to use and all the optimisation settings (defaults) according to the case running, generating a BOpt structure necessary for the BayesianOptimisation.jl package.

Outputs: BOpt structure necessary for the BayesianOptimisation.jl package.

7.8 mainOEDMS(oedms_def)

Inputs: OED for model selection definition structure with filled values.
Function: This is the main function of the section, which takes all the specifications from the user, checks that everything is correct and if so, generates the model and utility/cost function scripts, sets up the Bayesian optimisation and runs it saving all the results after.
Outputs: OED for model selection results and definition structure with filled values.The OED for model selection definition structure will also contain the fields savepath and savename.

8 Optimal Experimental Design for Model Calibration

8.1 defODEModelCalibrStruct()

Inputs: None
Function: This function allows the user to obtain the dictionary structure with all the keys for the OED for model calibration section so only the values for each key have to be filled.
Outputs: Dictionary with keys Model, Obs, Theta, y0, preInd, finalTime, switchT, tsamps, equalStep, fixedInp, fixedStep, plot, flag, uUpper, uLower, maxiter and util. Values for each key are empty lists [].

8.2 checkStructOEDMC(oedmc_def)

Inputs: Dictionary with OED for model calibration dictionary with the keys specified in defODEModelCalibrStruct() and filled values for each one.
Function: This function checks the contents of the dictionary introduced (structure from defODEModelCalibrStruct()). 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.

8.3 genOptimMCFuncts(oedmc_def)

Inputs: OED for model calibration structure dictionary with all the filled values for each key.
Function: This function extracts all the information about the model and how the experiment will be designed (sampling times, switching times, observables, etc.) and generated the Utility function script for the optimisation.

8.4 plotOEDMCResults(oedmc_res, oedmc_def)

Inputs: OED for model calibration results and OED for model calibration definition structures with filled values.
Function: This function generates and saves (in the results directory) the

plots for the Bayesian OED for model calibration results. A plot for the designed experiment will be generated, where there will be a subplot for each observable and each inducer of the system. The convergence plot for the optimisation will also be generated.
Outputs: None.

8.5 settingsBayesOptMC(oedmc_def)

Inputs: OED for model calibration definition structure with filled values.
Function: This function sets the bounds for the different inputs, utility function to use and all the optimisation settings (defaults) according to the case running, generating a BOpt structure necessary for the BayesianOptimisation.jl package.
Outputs: BOpt structure necessary for the BayesianOptimisation.jl package.

8.6 mainOEDMC(oedmc_def)

Inputs: OED for model calibration definition structure with filled values.
Function: This is the main function of the section, which takes all the specifications from the user, checks that everything is correct and if so, generates the model and utility function scripts, sets up the Bayesian optimisation and runs it saving all the results after.
Outputs: OED for model calibration results and definition structure with filled values.The OED for model calibration definition structure will also contain the fields savepath and savename.

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. If the model has no inputs, set it to 0.

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. If the model has no inputs just give an empty vector.

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.
Note that models without external inputs are also supported except for the optimal experimental design sections.

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" –>

CALL defBayInfStruct()

MAIN STRUCTURE

bayinf_def["Priors"] = []; Five options for this:
1) A 2*N array containing the bounds for the parameters. Order of parameters will be assumed as the one introduced in model_def["parName"]. As a prior a truncated Normal covering 2 standard deviations in the bounds given will be generated as prior.
2) Path to a CSV file containing samples for the parameters. Fitting of the samples to different type of distributions will be done to generate the priors. Order of parameters will be assumed as the one introduced in model_def["parName"]. You can also introduce a 2D array of floats with the samples (ArrayFloat64,2).
3) Dictionary with fields pars, transpars and pridis defining the parameters, subsequent desired transformations and prior distributions. In the field pars, parameters have to be defined with the same names and order as in model_def["parName"], otherwise the script will not proceed.
4) An empty array. If this is the case, the stan model will be generated with nothing in the parameters and transformed parameters section. The path to

the stan file will be given so the user can fill these sections.
5) Path to a Stan model file if you already have one. For example, if first you introduce an empty array, then you can fill the parameters section and run from that script instead of having to copy the things in here.

bayinf_def["Data"] = []; Two options:
1) A dictionary containing 3 fields. Observables, for the path to the files containing the experimental data. Inputs, for the path to the files containing the input profiles for the experiments. Obs, a string vector containing the observables of the experiments. The format of the files has to be the same one as the ones generated in the pseudo-data section. For each of the 2 entries, more than one file can be given if a multi-experimental inference wants to be done. Obs file also needs to be given. A y0 in a field with the same name has to be given for each experiment. The field is compulsory, even if not used.
WARNING: For now, this option does not include extraction of covariance matrix for data.
2) A dictionary containing the same structure as the simul_def plus the fields DataMean and DataError containing the experimental data. You can call the function defBayInfDataStruct() to obtain the empty structure of the dictionary.

bayinf_def["StanSettings"] = []; Two options:
1) A dictionary with the basic fields from a stan run. The structure of the dictionary can be extracted calling the function defBasicStanSettingsStruct().
2) An empty array. If this is the case, no inference will be done after calling the main function. Instead, you will be given a StanModel file and data structure and you will have to use this to run a call of Stan by yourself. An example will be provided.

bayinf_def["flag"] = [];
String to attach a unique flag to the generated files so it is not overwritten. If empty, nothing will be added.

bayinf_def["plot"] = [];
true, flase, "Yes", "yes", "No", "no" or [] indicating if plots with the results will be generated. Default is false.

bayinf_def["runInf"] = [];
true, flase, "Yes", "yes", "No", "no" or [] indicating if inference wants to be performed or only stan structure is given. Default is true (if all the necessary

elements are present)

bayinf_def["MultiNormFit"] = [];
true, false, "yes", "no", "Yes", "No" or [] indicating that if samples are given to fit, the prior will be fitted as a MultiNormal distribution or not. the default is false. If some parameter is better fit with a Log-Normal distribution, this will be used with a distribution reparameterisation to begin to be included in the multinormal. If for some parameter a Uniform distribution is better, this parameter will be excluded from the multinormal and defined as a separate parameter. This can be used as an example of how to set your Stan Model to use multi_normal priors.

CALL defBayInfDataStruct()

DATA DICTIONARY STRUCTURE

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

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

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

data_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.

data_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.

data_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).

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

data_def["DataMean"] = [];
Array containing the vector of means for each experiment.

data_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.

data_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).

CALL defBayInfDataFromFilesStruct()

DATA DICTIONARY STRUCTURE IF FILES ARE GIVEN

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

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

data_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).

data_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.

CALL defBasicStanSettingsStruct()

STAN SETTINGS DICTIONARY STRUCTURE

stan_def["cmdstan_home"] = [];
String containing the full path to the installation directory for cmdstan

stan_def["nchains"] = [];
Number of chains for the inference (integer)

stan_def["nsamples"] = [];
Number of post-warmup samples for each chain (integer)

stan_def["nwarmup"] = [];
Number of warm-up samples for each chain (integer)

stan_def["printsummary"] = [];
Print sumary of inference at the end. This can be either true or false. Default will be true.

stan_def["init"] = [];
Initial point for the parameters in each chain. See output of MLE results. This field can be empty. Please introduce the parameter values in their true parameter range.

stan_def["maxdepth"] = [];
Maximum tree-depth. This field can be empty. Check Stan documentation for more information.

stan_def["adaptdelta"] = [];
Delta value between 0 and 1. This field can be empty. Check Stan documentation for more information.

stan_def["jitter"] = [];
Jitter value between 0 and 1. This field can be empty. Check Stan documentation for more information.

"oedms" –>

CALL defODEModelSelectStruct()

oedms_def["Model_1"] = [];
Model structure for Model 1. Dict. See Model Generation Section.
Note that the order of the inputs is the one defined in this model. If there is a chance that Model_2 has an input that does not exist in this model, these will be appended at the end of the ones in Model_1.

oedms_def["Model_2"] = [];
Model structure for Model 2. Dict. See Model Generation Section.

oedms_def["Obs"] = [];
States of the model that are observables. This is a vector of strings.These could also be an expression combining states (Only +,-,*,/ and ŵill be considered).

oedms_def["Theta_M1"] = [];
Theta vector (frequentist OED or model with 1 parameter) or matrix (Bayesian OED) for model 1. Path to file with samples is also allowed.

oedms_def["Theta_M2"] = [];
Theta vector (frequentist OED or model with 2 parameter) or matrix (Bayesian OED) for model 2. Path to file with samples is also allowed.

oedms_def["y0_M1"] = [];
For Model 1:
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.

oedms_def["y0_M2"] = [];
For Model 2:
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.

oedms_def["preInd_M1"] = [];
For Model 1:
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.

oedms_def["preInd_M2"] = [];
For Model 2:
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.

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

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

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

oedms_def["fixedInp"] = [];
If more than 1 inducer for the system exists, but only 1 input can be dynamic, give a vector of strings indicating which inputs are going to be optimised but as a constant input instead of dynamic. If none, just give an empty vector.
If there is only 1 stimuli in the model, this field will be ignored.

oedms_def["fixedStep"] = [];
If you want any of the steps to be fixed to a value. This has to be an empty

array if none is fixed or an array of tuples, where each tuple is a step to be fixed. The first entry of the tuple is the index of the step (as an Integer), and the second and array of values for each inducer. Note that the fixed inputs will be ignored, so do not take them into account here.
The example type should be ArrayArrayInt,1,1 or an empty array ([]) if not used.

oedms_def["equalStep"] = [];
If you want a series of steps to have the same optimised value (for example if you want to design a pulse experiment) you can introduce inside this array different arrays with the indexes of the steps that will have the same value. The values introduced in each array need to be integers.

oedms_def["plot"] = [];
true, flase, "Yes", "yes", "No", "no" or [] indicating if plots with the results will be generated. Default is false.

oedms_def["flag"] = [];
String to attach a unique flag to the generated files so it is not overwritten. If empty, nothing will be added.
The order of uUpper and uLower will be taken as the order of inducers defined in Model 1. If Model 2 has an aditional input, this will be added after, so consider this when introducing the bounds for them.

oedms_def["uUpper"] = [];
Vector indicating the upper bounds for the inducers

oedms_def["uLower"] = [];
Vector indicating the lower bounds for the inducers

oedms_def["maxiter"] = [];
Maximum number of iterations for the Bayesian Optimisation. If nothing is introduced a default of 100 iterations will be taken

"oedmc" –>

CALL defODEModelCalibrStruct()

oedmc_def["Model"] = [];
Dict with Model. See Model Generation Section.

oedmc_def["Obs"] = [];
States of the model that are observables. This is a vector of strings.These could also be an expression combining states (Only +,-,*,/ and ŵill be considered).

oedmc_def["Theta"] = [];
Theta matrix (Bayesian OED) for the model. No single vectors will be allowed. Path to file is also allowed.

oedmc_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.

oedmc_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.

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

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

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

oedmc_def["fixedInp"] = [];
If more than 1 inducer for the system exists, but only 1 input can be dynamic, give a vector of strings indicating which inputs are going to be optimised but as a constant input instead of dynamic. If none, just give an empty vector.
If there is only 1 stimuli in the model, this field will be ignored.

oedmc_def["fixedStep"] = [];
If you want any of the steps to be fixed to a value. This has to be an empty

array if none is fixed or an array of tuples, where each tuple is a step to be fixed. The first entry of the tuple is the index of the step (as an Integer), and the second and array of values for each inducer. Note that the fixed inputs will be ignored, so do not take them into account here.

oedmc_def["equalStep"] = [];
If you want a series of steps to have the same optimised value (for example if you want to design a pulse experiment) you can introduce inside this array different arrays with the indexes of the steps that will have the same value. The values introduced in each array need to be integers.

oedmc_def["plot"] = [];
true, flase, "Yes", "yes", "No", "no" or [] indicating if plots with the results will be generated. Default is false.

oedmc_def["flag"] = [];
String to attach a unique flag to the generated files so it is not overwritten. If empty, nothing will be added.

oedmc_def["uUpper"] = [];
Vector indicating the upper bounds for the inducers

oedmc_def["uLower"] = [];
Vector indicating the lower bounds for the inducers

oedmc_def["maxiter"] = [];
Maximum number of iterations for the Bayesian Optimisation. If nothing is introduced a default of 100 iterations will be taken

oedmc_def["util"] = [];
String indicating entropy or perc (or percentile) as the core of the utility function to compute the uncertainty of the model simulations. The default will be to use percentiles.