ADAPT Modeling Framework (AMF) Manual

Contents

[1. Setup and preparation 4](#_Toc404254891)

[1.1 Required knowledge 4](#_Toc404254892)

[1.2 Preparing the ODEMEX toolbox 4](#_Toc404254893)

[1.3 Setting up a new project 5](#_Toc404254894)

[2. Setting up the model 6](#_Toc404254895)

[2.1 The model predictor 7](#_Toc404254896)

[2.2 Model constants 7](#_Toc404254897)

[2.3 Model inputs 8](#_Toc404254898)

[2.3.1 Data inputs 8](#_Toc404254899)

[2.3.2 Function inputs 8](#_Toc404254900)

[2.4 Model parameters 9](#_Toc404254901)

[2.5 Model states 9](#_Toc404254902)

[2.6 Model reactions 10](#_Toc404254903)

[2.6.1 Using reactions to define model outputs 11](#_Toc404254904)

[2.6.2 Conditional if-statements 11](#_Toc404254905)

[2.7 Restrictions 12](#_Toc404254906)

[2. Preparing experimental data 13](#_Toc404254907)

[3. Setting up the dataset 14](#_Toc404254908)

[3.1 Groups 14](#_Toc404254909)

[3.2 Data fields 15](#_Toc404254910)

[3.3 Data functions 16](#_Toc404254911)

[3.3.1 Custom functions 17](#_Toc404254912)

[4. Setting up the run file 18](#_Toc404254913)

[4.1 Creating the model and dataset objects 18](#_Toc404254914)

[4.2 Coupling the dataset to the model 18](#_Toc404254915)

[4.3 Configuring the model 18](#_Toc404254916)

[4.4 Computation 19](#_Toc404254917)

[4.5 Results analysis 20](#_Toc404254918)

[5. Model options 21](#_Toc404254919)

[6. Advanced configuration 22](#_Toc404254920)

[6.1 Accessing model components 22](#_Toc404254921)

[6.1.1 By reference 22](#_Toc404254922)

[6.1.2 By type 23](#_Toc404254923)

[6.2 Accessing observable data 24](#_Toc404254924)

[6.3 Customizable functions 25](#_Toc404254925)

[6.4 Regularization 27](#_Toc404254926)

[6.4.1 Defining additional penalties 28](#_Toc404254927)

[6.5.2 ADAPT regularization 29](#_Toc404254928)

[7. Saving and loading results 30](#_Toc404254929)

[7.1 Loading results 30](#_Toc404254930)

[8. References 31](#_Toc404254931)

# 1. Setup and preparation

The framework can be installed by running the setup function in the AMF root directory. This will add the AMF package and ODEMEX toolbox to the MATLAB search path, allowing these to be used from any location, and will attempt to start a parallel pool to enable computation using multiple cores (see help matlabpool). Any sub-directories will also be included in the search path.

## Required knowledge

Basic knowledge of MATLAB functions and data types is assumed. The following link explains the different data types supported by MATLAB:

<http://nl.mathworks.com/help/matlab/data-types_data-types.html>

While knowledge of object-oriented programming is not required for basic use, advanced configurations require some basic knowledge of how to use objects in MATLAB. Objects can be interpreted as structures with incorporated methods:

<http://nl.mathworks.com/company/newsletters/articles/introduction-to-object-oriented-programming-in-matlab.html>

Basic knowledge of the estimation of parameter trajectories using the ADAPT algorithm is required.1 In addition, the examples used in this manual make use of the toy model developed by Natal van Riel et al.2

## 1.2 Preparing the ODEMEX toolbox

The AMF package uses the ODEMEX toolbox to enable MEX compilation for computational files in order to improve computation times. This toolbox requires configuration in order to be used. Instructions on how to configure the ODEMEX toolbox can be found in the **Instructions.pdf** file located in the **odemex** folder.

## 1.3 Setting up a new project

It is recommended to create a new directory for each new AMF project. A project generally contains the following files:

* A model function;
* A dataset function;
* A mat-file containing the experimental data values;
* A run file;
* Custom functions (optional).

It is recommended to use unique names for these files to prevent naming conflicts that may occur when adding multiple projects to the MATLAB search path.

# 2. Setting up the model

A dynamical model is defined in a model file. This file contains a single function that returns a structure containing the specification of all components that are included in the model. This structure contains the following fields:

|  |  |
| --- | --- |
| DESCRIPTION | The model description. |
| PREDICTOR | Specification of the model predictor. |
| [CONSTANTS] | Specification of model constants (optional). |
| [INPUTS] | Specification of model inputs (optional). |
| PARAMETERS | Specification of model parameters. |
| STATES | Specification of model states. |
| [REACTIONS] | Specification of model reactions (optional). |

Start by defining the model function and adding a description for the model.

% toyModel.m  
function MODEL = toyModel()  
  
MODEL.DESCRIPTION = 'Toy model created by N.A.W van Riel.';

The remaining fields of the model structure are used to specify the model components. Components are grouped by type (i.e. predictor, constant, input, parameter, state and reaction) and are defined using cell matrices. Each type requires a single specification cell matrix of which one row defines a single model component. The columns of a specification matrix represent properties that can be used to customize components. While properties are specific to component types, the first and last property of components of all types correspond to the component name and the optional meta-data respectively.

|  |  |  |
| --- | --- | --- |
| Shared properties | | |
| name | char | **Unique** component identifier. |
| meta-data | cell (array) | Optional meta-data used for visualization of model results. meta-data should ether be an empty cell array {} or an array containing the component’s unit type, unit and plotting label: {‘unitType’ ‘unit’ ‘label’}. |

## 2.1 The model predictor

|  |  |  |
| --- | --- | --- |
| Properties | | |
| interval | double (vector) | Vector containing the start and end values of the (time) interval. |

The model predictor is used to define the independent variable of the model (usually time). The (time) interval defined is used for simulation and the computation of trajectories.

MODEL.PREDICTOR = {  
% name interval meta-data  
 't' [0 10] {'time' 'days' 'Time'}  
};

## 2.2 Model constants

|  |  |  |
| --- | --- | --- |
| Properties | | |
| value | double OR char | Constant value (double) OR name of a data component (char). |

Model constants can be defined as static values by using a number for the value property. In addition, constant values can be obtained from the dataset by using the name of the data component for the value property.

MODEL.CONSTANTS = {  
% name value meta-data  
 'mBW' 'BW' {'mass' 'kg' 'body weight'}  
};

In this example, the mean body weight of the subject group is defined in the dataset as ‘BW’. By using this syntax, the model constant ‘mBW’ obtains this value for use in the model ODE. The advantage of obtaining constant values from data is that these values are automatically updated when switching between datasets or subject groups.

## 2.3 Model inputs

|  |  |  |
| --- | --- | --- |
| Properties | | |
| type | char | Input type: ‘data’ for data inputs; ‘function’ for function inputs. |
| args | Cell (array) | Array of input arguments. |
| method | char | Interpolation method: ‘linear’ for linear interpolation; ‘spline’ for cubic spline interpolation. |

Dynamic model inputs can be obtained directly from data, or defined as a function of model parameters. Constant inputs are treated identical model constants, and should therefore be defined as model constants (2.2). The method property specifies the interpolation method to be used in the ODE solver.

### 2.3.1 Data inputs

For data inputs, args should contain a single element: the name of the data component.

MODEL.INPUTS = {  
% name type args method meta-data  
 'Insulin' 'Data' {'Insulin'} 'Linear' {'conc.' 'uU/ml' 'Insulin'}  
};

In this example, the insulin concentration for the current subject group is obtained from the dataset to be used as a model input.

### 2.3.2 Function inputs

For function inputs, the first element of args should contain the time interval. Additional elements need to be added for each time point in this interval, containing the names of the model parameters that compose the dynamic input.

MODEL.INPUTS = {  
% name type args method meta-data  
 'u1' 'Function' {[0 3 5 10],'p1','p2','p3','p4'} 'Linear' {}  
};  
  
MODEL.PARAMETERS = {  
% name fit value bnd meta-data  
 'p1' 1 1 [] {}  
 'p2' 1 1 [] {}  
 'p3' 1 1 [] {}  
 'p4' 1 1 [] {}  
};

This example demonstrates the definition of a piece-wise linear interpolated input function composed of 4 model parameters that should be fitted when optimizing the model.

## 2.4 Model parameters

|  |  |  |
| --- | --- | --- |
| Properties | | |
| fit | double (bool) | Indicator for fitting: 1 for fit parameters; 0 for constant parameters. |
| value | double | Initial parameter value. |
| bnd | double | Lower and upper boundary of the estimated value. Will default to [0 inf] if left empty. |

Model parameters can be defined as constant parameters, or parameters that should be fitted during model optimization using fit. The boundary conditions for parameter estimation are specified in the bnd property.

## 2.5 Model states

|  |  |  |
| --- | --- | --- |
| Properties | | |
| init | double | Initial state value. |
| ode | char | The ODE describing the state time derivative. |

Model states define the initial model state, as well as the model ODE that describes the state time derivatives. The ode property contains the equation for the state time derivative (ODE), and specifies a function of other model components (constants, inputs, parameters, states and reactions).

MODEL.STATES = {  
% name init ode meta-data  
 's1' 0 'v1 - v3 - v4' {}  
 's2' 0 '-v1 + v2' {}  
 's3' 0 'v1 - v2' {}  
 's4' 0 'v4 - v5' {}  
};

In this example, the state time derivatives are defined as functions of model reactions (fluxes in this case).

## 2.6 Model reactions

|  |  |  |
| --- | --- | --- |
| Properties | | |
| expr | char | The expression describing the reaction as a function of other components. |

Model reactions are abstract model components that can represent any mathematical functions of other components (similar to state time derivatives). Generally, they are used to define intermediary component functions (i.e. fluxes or other physiological entities) which are in turn used to compose the model ODE (2.5).

MODEL.STATES = {  
% name init ode meta-data  
 's1' 0 'v1 - v3 - v4' {}  
 's2' 0 '-v1 + v2' {}  
 's3' 0 'v1 - v2' {}  
 's4' 0 'v4 - v5' {}  
};  
  
MODEL.REACTIONS = {  
% name expr meta-data  
 'v1' 'k1 \* u1 \* s2' {}  
 'v2' 'k2 \* u2 \* s3' {}  
 'v3' 'k3 \* s1' {}  
 'v4' 'k4 \* s1' {}  
 'v5' 'k5 \* s4' {}  
};

In this example, reactions are used to define fluxes, which are in turn used to define the model ODE in the model states.

### 2.6.1 Using reactions to define model outputs

Model reactions can be used to define any function of model components, even if they are not used to compose the model ODE. This can be useful since the framework will automatically compute and visualize any defined reaction, providing an easy way to analyze desired model outputs.

MODEL.REACTIONS = {  
% name expr meta-data  
 'v1' 'k1 \* u1 \* s2' {}  
 'v2' 'k2 \* u2 \* s3' {}  
 'v3' 'k3 \* s1' {}  
 'v4' 'k4 \* s1' {}  
 'v5' 'k5 \* s4' {}  
  
 'y1' 'v1 + v2' {}  
 'y2' 's1 \* 10^6' {}  
};

In this example, 2 model outputs are defined as additional model reactions. One computes the sum of fluxes v1 and v2, and the other performs a unit conversion on state s1.

### 2.6.2 Conditional if-statements

The use of conditional if-statements is supported in reaction expressions using a specific syntax and is compatible with MEX compilation.

MODEL.REACTIONS = {  
% name expr meta-data  
 'posG' 'if((G > 0), G, 0)' {} % single condition  
 'posS' 'if((G > 0) && (S > 0), 1, 0)' {} % multiple conditions  
};

In this example, S and G represent other model components (i.e. states). Reaction posG will have the value of G if G > 0, and 0 otherwise. Reaction posS will have a value of 1 if both G > 0 and S > 0, and 0 otherwise.

## 2.7 Restrictions

Several restrictions apply to the definition of model components:

* **All component names must be unique, even for components of different types;**
* **Only reaction expressions can contain conditional if-statements (2.6.2).**

To ensure compatibility with MEX compilation, several additional restrictions apply to component functions in reaction expressions (2.5) and state time derivatives (2.5):

* **MATLAB functions cannot be used in component functions;**
* **Powers should be defined using pow(A,B) or A^B. Exponent notation (i.e. 1e6) is not allowed;**

If these last conditions are not met, computational algorithms can still be executed without the optimized performance provided by MEX compilation.

# 2. Preparing experimental data

Experimental data has to be provided using a nested data structure that is restricted to a specific format. This structure needs to be prepared and saved only once, before defining the dataset.

On the first level, the data structure contains the names of the separate subject groups (group names can be chosen arbitrarily). The second level contains the measurement times, means and standard deviations of the data fields (i.e. metabolite concentrations). The data structure used in the following examples represents a dataset of two subject groups toy and toy2 containing the measured means and standard deviations of metabolite concentrations s1 s2 s3 s4.

data =   
  
 toy: [1x1 struct]  
 toy2: [1x1 struct]

data.toy =   
  
 t: [5x1 double]  
 s1\_mean: [5x1 double]  
 s1\_std: [5x1 double]  
 s2\_mean: [5x1 double]  
 s2\_std: [5x1 double]  
 s3\_mean: [5x1 double]  
 s3\_std: [5x1 double]  
 s4\_mean: [5x1 double]  
 s4\_std: [5x1 double]

The measurement times for all measured metabolites must be included in the dataset. In this case, all measurements were done on the same time interval t. If data of multiple subject groups is available (as is the case here) the data field names must be identical between groups: toy2 must have the same field names for the measurement times, means and standard deviations of s1 s2 s3 and s4. Note that the data values do not have to be identical: these generally depend on the experiment and may be different between subject groups.

Once the data structure has been created, it should be saved in the project root directory.

save('toyData.mat', '-struct', 'data');

# 3. Setting up the dataset

A dataset is defined in a dataset function. This function returns a structure containing the data specification (similar to the definition of models). This structure contains the following fields:

|  |  |
| --- | --- |
| DESCRIPTION | The dataset description. |
| FILE | The name of the MAT file to which the experimental data structure is saved. |
| GROUPS | Subject group names. |
| FIELDS | Data fields. |
| FUNCTIONS | Data functions. |

Start by defining the dataset function and adding a description for the dataset.

% toyData.m  
function DATASET = toyData()  
  
DATASET.DESCRIPTION = 'Toy test data.';  
  
DATASET.FILE = 'toyData'; % toyData.mat

This dataset parses the data structure located in ‘toyData.mat’.

## 3.1 Groups

The GROUPS field specifies the names of the subject groups that may be used for model optimization. All of the specified subject groups must be available in the specified data structure.

DATASET.GROUPS = {  
 'toy'  
 'toy2'  
};

## 3.2 Data fields

|  |  |  |
| --- | --- | --- |
| Properties | | |
| name | char | Field name. |
| obs | double (bool) | Observable indicator. 1 for observables; 0 for regular data. |
| timeField | char | Name of the corresponding time field in the data structure. |
| meanField | char | Name of the corresponding mean value field in the data structure. |
| stdField | char | Name of the corresponding standard deviation field in the data structure. |
| unitConv | double | Unit conversion factor. |
| smooth | double | Smoothing parameter for the generation of random cubic smoothing splines (see help csaps for information on how this parameter affects the spline). |

Data fields can be defined to parse the experimental data values located in the supplied data structure. The obs property specifies whether or not the data should be considered observable. Observable data fields must have a name equal to the corresponding model state or reaction, and are automatically used to compute the model error during model optimization. The timeField , meanField and stdField specify the names of the corresponding fields in the data structure. Note that if the data field contains a single value (static data), the timeField property is not required and should be empty [].

DATASET.FIELDS = {  
% name obs timeField meanField stdField unitConv smooth  
 's1' 1 't' 's1\_mean' 's1\_std' 1 []  
 's2' 1 't' 's2\_mean' 's2\_std' 1 []  
 's3' 1 't' 's3\_mean' 's3\_std' 1 []  
 's4' 1 't' 's4\_mean' 's4\_std' 1 []  
};

Following the previous examples, the values for data fields s1 s2 s3 s4 are loaded from ‘toyData.mat’ upon loading a subject group (i.e. toy). In this dataset, data is available for all model states (2.5), therefore all defined data fields are observable.

Unit conversions can be specified using the unitConv property. Data values will be multiplied by this factor upon loading. In addition, the smoothing parameter for the generation of random cubic smoothing splines (i.e. for the ADAPT algorithm) can be specified using the smooth property.

## 3.3 Data functions

|  |  |  |
| --- | --- | --- |
| Properties | | |
| name | char | Field name. |
| obs | double (bool) | Observable indicator. 1 for observables; 0 for regular data. |
| func | char | Name of the corresponding time field in the data structure. |
| valArgs | char | Name of the corresponding mean value field in the data structure. |
| stdArg | char | Name of the corresponding standard deviation field in the data structure. |

Data functions are functions of data fields. The function specified in the func property is applied to the values of the data fields (or other data functions) specified in the valArgs cell array. Note that this function will be applied to the source values, current (randomized) values as well as the interpolated values. Data functions should therefore be used when fitting the model to interpolated data (i.e. the random splines generated during ADAPT), if the data of an observable corresponds to a function (i.e. sum, product) of data fields. In the current implementation, the corresponding standard deviations for the defined data function are copied from the data field specified in the stdArg property.

DATASET.FUNCTIONS = {  
 %name %obs %func %valArgs %stdArg  
 'dplasma\_C\_HDL' 1 @prod {'dplasma\_C', 'dplasma\_C\_HDL\_FPLC'} 'dplasma\_C\_HDL\_FPLC'  
 'dplasma\_C\_0' 0 @diff {'dplasma\_C', 'dplasma\_C\_HDL'} 'dplasma\_C'  
};

In this example, the values of 'dplasma\_C\_HDL' will be the product of the values of data fields 'dplasma\_C' and 'dplasma\_C\_HDL\_FPLC'. Its standard deviation will be copied from data field 'dplasma\_C\_HDL\_FPLC'.

Note that since no function is applied to the standard deviation, custom standard deviations need to be defined manually in the data structure, and defined in the dataset using a mock data field (this might change in future versions).

### 3.3.1 Custom functions

The function that is specified in the func property of a data function needs to be compatible with matrices. To illustrate this, consider the previous example. Before the function is applied, the following matrix is constructed:

dplasma\_C.val = [1 2 3 4 5];  
dplasma\_C\_HDL\_FPLC.val = [1 1 1 1 1];  
  
tmpVal = [dplasma\_C.val; dplasma\_C\_HDL\_FPLC.val]

tmpVal =  
  
 1 2 3 4 5  
 1 1 1 1 1

Then, the function is applied to this matrix to obtain the values of the data function. In the example, both prod and diff are native MATLAB functions that are compatible by default:

prod(tmpVal)

ans =  
  
 1 2 3 4 5

diff(tmpVal)

ans =  
  
 0 -1 -2 -3 -4

Any custom function can be used, as long as the desired operation is automatically applied to each column of the input matrix.

# 4. Setting up the run file

Once the model and dataset are defined, the model and dataset objects need to be created and configured in order to perform computational algorithms (i.e. model simulation, optimization or calculation of parameter trajectories using ADAPT). In order to provide a clear workflow, it is recommended to centralize these operations in a single run file for each separate task.

Start by importing the AMF package for easy access to its classes and methods.

% runToy.m  
import AMF.\*

## 4.1 Creating the model and dataset objects

The model and dataset objects can be created directly from their respective specification files.

model = Model('toyModel'); % toyModel.m  
data = DataSet('toyData'); % toyData.m

## 4.2 Coupling the dataset to the model

Data values need to be loaded by loading a subject group. If no group is loaded explicitly, the first defined subject group is loaded automatically. Therefore, explicit loading of a subject group is not required if only one group is defined. The dataset is then ready to be coupled to the model.

loadGroup(data, 'toy');  
initiateExperiment(model, data);

## 4.3 Configuring the model

The model can be configured by customizing the model options. These options control which files should be compiled and serve as parameters for computational algorithms. An overview of available model options is provided in chapter 5.

model.options.useMex = 1;  
model.options.odeTol = [1e-12 1e-12 100];  
model.options.numIter = 50;  
model.options.numTimeSteps = 100;  
model.options.parScale = [2 -2];  
model.options.seed = 1;  
model.options.SSTime = 1000;  
model.options.lab1 = .1;

This example configures the model for the execution of the ADAPT algorithm to compute 50 trajectories over 100 time steps, with MEX compilation enabled to drastically improve computation times.

The model needs to be parsed to resolve all interactions and dependencies between data and model components and compiled to generate the necessary computation files.

parseAll(model);  
compileAll(model);

## 4.4 Computation

Once the model is configured, computational algorithms can be performed. In the previous examples, the model was set up for the ADAPT algorithm, which can now be executed.

result = runADAPT(model);

The runADAPT method executes the ADAPT algorithm and returns a result object containing the computed trajectories and corresponding model errors.

Alternatively, this segment of the run file can contain other algorithms. The AMF package currently supports the following algorithms:

|  |  |
| --- | --- |
| simulate | Simulate the model by computing the states (and all other components) for the time range of the current predictor. |
| fit | Fit the model using dataset and simulate the model using the optimized parameters. |
| runADAPT | Run ADAPT on the model to compute possible parameter trajectories. |

## 4.5 Results analysis

Once the computational algorithm has been executed, the returned result object can be used to visualize the computed trajectories.

plotAll(result, 'parameters', 'traj');  
plotAll(result, 'states', 'traj');

In this example, we plot the estimated parameter trajectories as well as the state trajectories. This will output 2 figures. Note that the meta-data provided to model components is automatically used for the visualization of results. In addition, any observable model component will automatically include its data (mean and standard deviations).

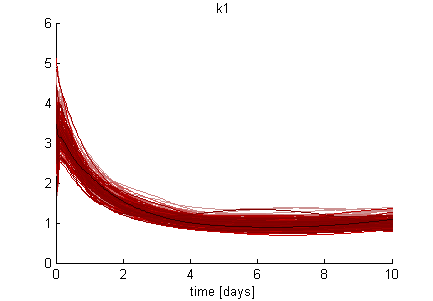


Figure 1: estimated parameter trajectories

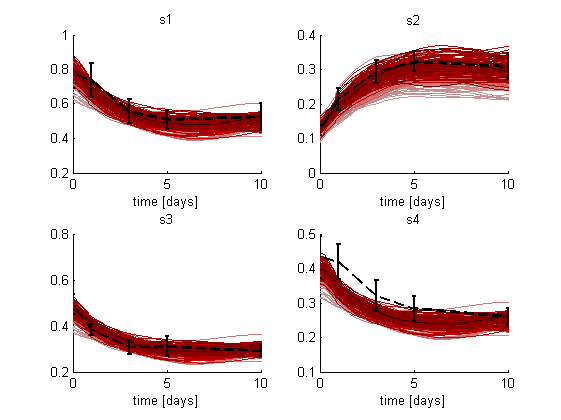


Figure 2: estimated state trajectories

# 5. Model options

Several options for computational algorithms can be configured on the model:

|  |  |  |
| --- | --- | --- |
| General options | | |
| optimset | struct | Optimization OPTIONS structure used by the optimization algorithm (lsqnonlin). See help optimset for more information. |
| useMex | double (bool) | Indicator that specifies whether or not the MEX compilation should be used to improve computation times (recommended). |
| timeSteps | double | The number of time steps used for simulation or trajectory estimation (ADAPT). |
| ADAPT options | | |
| parScale | double | 2-element vector containing the upper and lower boundaries for the generation of random parameters during the ADAPT algorithm. |
| numIter | double | The number of iterations (trajectories) to be calculated during the ADAPT algorithm. |
| lab1 | double | The regularization factor used during the ADAPT algorithm. |
| seed | double | The initial random seed (can be any arbitrary number). Using the same initial random seed will always generate identical trajectories during the ADAPT algorithm. Note that the generation of identical trajectories fails when parallel computing is enabled (parallel computing causes trajectory calculations in random order). |
| SSTime | double | The time used to fit the initial steady state. The specified value for SSTime will be prefixed to the current time vector prior to fitting the initial time step. |
| savePrefix | Char | A string that is used as prefix when saving model results. This option is used to prevent the automatic overwriting of result files. |

Note that since the model options are stored in a dynamic structure, any custom options can be defined for use in custom regularization functions (chapter 6). For example, a custom regularization function might require additional regularization factors:

model.options.labC = [.1 .01 .01];

# 6. Advanced configuration

Advanced configurations can be achieved by customizing several functions that are used during the execution of computational algorithms. However, using custom functions requires some knowledge of the internal workings of the framework, in particular of how model component and data field objects can be accessed to obtain their computed values.

## 6.1 Accessing model components

Once a model object is created from a specification file (4.1), there are several interfaces available on the object to access the individual components.

### 6.1.1 By reference

A model object has a ref property that contains a structure in which references to all component objects are stored by name. Following the previous examples, state s1 can be accessed using this structure.

model.ref.s1

ans =   
  
 AMF.ModelState handle  
 Package: AMF  
  
 Properties:  
 derivedODE: []  
 compiledODE: []  
 index: 1  
 name: 's1'  
 expr: 'v1 - v3 - v4'  
 data: [1x1 AMF.DataField]  
 time: []  
 init: 0  
 prev: 0  
 curr: 0  
 val: []  
 unitType: []  
 unit: []  
 label: []

While the properties of the component objects are different for each type, all components have identical properties for storing computed values, which can be used to create alternative error and regularization functions (6.3-6.5):

|  |  |  |
| --- | --- | --- |
| Component object properties that store computed values | | |
| init | The initial value. |
| curr | The current value. This value updates after fitting each time step during ADAPT. |
| prev | The previous value. Equals curr of the previous time step. |
| val | Values of the resulting trajectory. |
| data | Only used for observable components: contains a reference to the corresponding data object. |

### 6.1.2 By type

Additionally, components are grouped by type and stored in properties of the model object as arrays. The following component arrays are available.

|  |  |  |
| --- | --- | --- |
| Model properties containing component arrays | | |
| constants | Model constants. |
| inputs | Model inputs. |
| parameters | Model parameters. |
| fitParameters | Fit parameters (subset of parameters). |
| states | Model states. |
| reactions | Model reactions. |

These component arrays can be used to access multiple values simultaneously. For example, a single statement can retrieve the current values for parameters:

pcurr = [model.parameters.curr]

pcurr =  
  
 1.0000 1.0000 0.1000 0.5000 1.0000

## 6.2 Accessing observable data

For observable model components, the data property contains a reference to the corresponding data object (6.1).

s1 = model.ref.s1;  
s1.data

ans =   
  
 AMF.DataField handle  
 Package: AMF  
  
 Properties:  
 name: 's1'  
 obs: 1  
 index: 1  
 timeField: 't'  
 valField: 's1\_mean'  
 stdField: 's1\_std'  
 unitConv: 1  
 smooth: []  
 source: [1x1 struct]  
 time: [5x1 double]  
 val: [5x1 double]  
 std: [5x1 double]  
 options: []  
 fitIdx: [5x1 double]

The following properties of a data object contain computed values.

|  |  |  |
| --- | --- | --- |
| Data object properties that store computed values | | |
| time | Time interval used for interpolation. |
| val | Interpolated values. |
| std | Interpolated standard deviations. |
| source.time | Measurement time of the experimental data values. |
| source.val | Measured experimental values. |
| source.std | Measured experimental standard deviations. |
| fitIdx | The indices of the source.time locations in the complete fit time interval. |

For regular model optimization, the framework simulates the model only on the time points for which observable data is available. Since different observables can have different measurement times, the indices of the bigger fit time interval that correspond to the measurement times are stored in the fitIdx property.

## 6.3 Customizable functions

The functions that are used for computational tasks are stored in the functions property of the model object. By default, this property contains a structure with function handles to the relevant functions.

model.functions

ans =   
  
 ODE: @C\_toyModel\_ODE  
 ODEMex: @C\_toyModel\_ODEMEX  
 ODEC: @C\_toyModel\_ODEC  
 reactions: @C\_toyModel\_REACTIONS  
 inputs: @C\_toyModel\_INPUTS  
 reg: @AMF.regFun  
 err: @AMF.errFun  
 errStep: @AMF.errFunStep

To realize advanced configurations for computational algorithms, these functions are fully customizable and can be replaced with alternative custom functions in the run file. However, it is advisable to only change the functions that are not automatically generated when compiling the model.

|  |  |
| --- | --- |
| Compiled model functions | |
| ODE | The compiled ODE function used by the solver algorithm to compute the dynamic model states. |
| ODEMex | The compiled function that is used as in input to the ODEMEX toolbox in order to compile the MEX function (this function contains a syntax specific to the toolbox and cannot be run directly). |
| ODEC | The compiled MEX ODE function. This function will only be compiled when MEX compilation is enabled in the model options (useMex). |
| reactions | The reactions function that computes the dynamic reactions for any given time. |
| inputs | The inputs function that computes (interpolates) the dynamic inputs for any given time. |
| Customizable model functions | |
| reg | The regularization function. This function is used for ADAPT regularization, but can be used to define any additional model errors (penalties) that should be considered during optimization. |

## 6.4 Regularization

The regularization function is used to define any additional errors that should be considered during model optimization. The complete model object is passed to this function, allowing the use of any model or data components for the computation of errors. Model components can be accessed using the methods described in 6.1. Data components can be accessed by using the dataset model property, which contains a reference to the current dataset.

model.dataset

ans =   
  
 AMF.DataSet handle  
 Package: AMF  
  
 Properties:  
 name: 'toyData'  
 description: 'Toy test data.'  
 specification: [1x1 struct]  
 data: [1x1 struct]  
 groups: {'toy'}  
 activeGroup: 'toy'  
 fields: [1x4 AMF.DataField]  
 functions: []  
 ref: [1x1 struct]  
 list: {4x1 cell}

A dataset object contains a ref property that can be used to access individual data objects, similar to the ref property of the model. Generally, only observable data is used which can be accessed directly from the observable model components (6.2).

### 6.4.1 Defining additional penalties

The regularization function can be used during regular optimization to define additional errors or penalties. For example, if the AUC of a metabolite concentration is to be remained constant, a penalty can be defined in a custom regularization function:

% minGlucReg.m  
% Regularization function of the minimal  
% glucose model by Dalla Man et al.  
function E = minGlucReg(model)  
  
Ra = model.ref.Ra\_g;  
Gtot = model.ref.Gtot;  
  
E = AUC(Ra) - Gtot.val;

In this example, total glucose intake Gtot (grams) is a model constant and is believed to be equal to the area under the curve (AUC) of the glucose rate of appearance in grams Ra\_g. The AUC of Ra\_g is approximated using cumulative trapezoidal numerical integration:

% AUC.m  
% Approximates the area under curve (AUC)  
% using cumulative trapezoidal numerical  
% integration.  
function val = AUC(comp)  
  
tmp = cumtrapz(comp.time, comp.curr);  
val = tmp(end);

To use this regularization function, it needs to be assigned to the model.

model.functions.reg = @minGlucReg;

### 6.5.2 ADAPT regularization

When executing the ADAPT algorithm, a default regularization function is used that penalizes erratic changes in parameter values, controlled by model option lab1:

% AMF.regFun  
function reg = regFun(model)  
  
t = getTime(model);  
p = model.fitParameters;  
  
lab1 = model.options.lab1;  
  
dt = t(end) - t(1);  
  
if t(end) == 0  
 reg = 0;  
else  
 reg = ([p.curr] - [p.prev]) ./ [p.init] ./ dt \* lab1;  
end

As seen in the default regularization function, any of the model object functions and properties can be used to compute additional errors. This allows for the creation of complex custom regularization functions that use any model or data components.

Note that if a custom regularization function is assigned to the model, this supplements the default ADAPT regularization function. To bypass default regularization, simply set the model option lab1 to 0.

# 7. Saving and loading results

After running the ADAPT algorithm on the model, the resulting trajectories are automatically saved in the ‘results’ sub-directory of the project root. Results are saved in the following format:

{options.savePrefix}\_{name}\_{options.numIter}\_{options.numTimeSteps}.mat

Note that options.savePrefix should be set in the run file to prevent overwriting of existing result files ( a suggestion would be to use the current date as prefix). An example results file might look like this:

13112015\_toyModel\_100\_50.mat

For other computational algorithms, results can be saved manually:

% fit the model  
fit(model);  
  
% save result structure in the results directory  
saveResult(model);

## 7.1 Loading results

Previously saved results can be loaded into a model result object.

resultStruct = load('13112015\_toyModel\_100\_50.mat');  
  
% create the ModelResult object  
result = AMF.ModelResult(resultStruct);

After the results are loaded in the model result object, it can be used to visualize trajectories. Several options for visualization include:

% parameter trajectories  
plotAll(result, 'parameters', 'traj');  
  
% parameter trajectory histogram  
plotAll(result, 'parameters', 'hist');  
  
% state trajectories  
plotAll(result, 'states', 'traj');

# 8. References

1. Tiemann, C. A. *et al.* Parameter Trajectory Analysis to Identify Treatment Effects of Pharmacological Interventions. *PLoS Comput Biol* **9,** e1003166 (2013).

2. Riel, N. A. W. van, Tiemann, C. A., Vanlier, J. & Hilbers, P. A. J. Applications of analysis of dynamic adaptations in parameter trajectories. *Interface Focus* **3,** 20120084 (2013).