ModelClass User Manual

Fernando Nóbel Santos Navarro (fersann1@upv.es)

March 11, 2021

Contents

1 Manual version

This manual was written for the version **v0.5.0** of ModelClass. Please check your local version of ModelClass using the following command:

```
1  ModelClass.version
```

and it should return something similar to this:

```
1  v0.5.0 250dd5d   -   Fernando Nóbel (fersann1@upv.es)
```

, then if your local version does not match, please update to the latest version in the GitHub repository.

2 Introduction

MATLAB is powerful piece of software with many specialized toolboxes, but it lacks tools that simplify the process of developing mathematical models. On the other hand, OpenModelica does a great job of defining models and making them reusable, however OpenModelica is restricted for simulation purposes only. ModelClass is a toolbox for MATLAB that replicates part of the OpenModelica functionality in native MATLAB, in this way, you can work effectively with models in MATLAB without dependencies to other pieces of software.

ModelClass is a MATLAB class which simplifies working with ODE models. The main objetive is to simplify the process of coding and simulating an ODE model, and therefore reducing the time spent in this task. With ModelClass one can program ODE models from the symbolic equations and then simulate directly. This class provides also some functionality like OpenModelica (i.e. extendable classes, simulation of DAE models, etc).

Apart from that, ModelClass provides us more classes for different tasks. For example, there is a class for mathematical analysis that can calculate equilibrium points, linearize the model, and calculate eigenvalues from the model defined in ModelClass. On other hand, there is class for a contractivity test to check whether a model is contractive or no.

Lastly it is even possible to define ModelClass models from chemical reactions directly and then perform QSSA analysis and simulate.

For more information please contact *fersann1@upv.es*.

3 Getting started

3.1 Installing from GitHub

The code of ModelClass is allocated in the following GitHub repository.

It is recommended to install the latest version of ModelClass, the available version of the software can be found in the releases in the GitHub repository.

Download the code and unzip it in the directory of your choice.

Then within MATLAB go to *HOME/ENVIROMENT » Set path* and add the directory of the repository and the *utils* and *parser* directories to the list (if they aren't already).

3.2 Keep the sowftware up-to-date

The ModelClass is currently in active development, so it is recommended to keep this software up to date. The master branch in the Github repository contains the latest stable version of the software

ModelClass has an integrated command to check if a newer version of it is available in the repository:

```
1  ModelClass.checkVersion();
```

and it should return the following message if the local code is updated,

```
1  The local version of ModelClass is up to date.
```

or the follwing message if if the local code is outdated:

```
1  The local version of ModelClass is outdated, please update to the ...
       latest version.
```

There are two ways to update your local code of ModelClass.

The first one is to manually download the latest release of the code in the repository in the master branch.

The second one is to execute the following command:

```
1  ModelClass.update();
```

and it will update the local code to the latest release in the repository automatically.

Note: if you have git installed locally, it is advisable to use git pull origin master to update your local repository as the ModelClass.updated(); could potentially mess up with the .git folder.

3.3 A minimal example

The following code show a minimal example of defining and simulating a model of the antithetic controller. It consists in two files: (i) the definition of the model as a .mc file and (ii) the main script that will simulate the model.

(i) model.mc

```
1  % Variables
2
3  variable x1;
4  variable x2;
5  variable x3;
6  variable ref(value = k3/d3);
7
8  % Parameters
9
10  parameter k1(value = 1.0);
11  parameter k2(value = 1.0);
12  parameter k3(value = 1.0);
13  parameter d1(value = 1.0);
14  parameter d2(value = 1.0);
15  parameter d3(value = 1.0);
16  parameter gamma12(value = 1.0);
17
18  % Equations
19
20  equation der_x1 == k1     - gamma12*x1*x2 - d1*x1;
21  equation der_x2 == k2*x3 - gamma12*x1*x2 - d2*x2;
22  equation der_x3 == k3*x1 - d3*x3;
```

(ii) main.m

```
1  % Initialize an object of the model.
2  m = ModelClass.load('./model/model.mc');
3
4  % Initialize a SimulationClass object with the model data.
5  s = SimulationClass(m);
6
7  % Simulation time span.
8  tspan = [0 10];
9
10  % Parameters of the model.
11  p = []; % They are already defined in "model.mc"
12
```

4

```
13  % Intial conditions of the model.
14  x0.x1 = 0.000000;
15  x0.x2 = 0.000000;
16  x0.x3 = 0.000000;
17
18  % Options for the solver.
19  opt = odeset('AbsTol', 1e-8, 'RelTol', 1e-8);
20
21  % Simulate the model.
22  [out] = s.simulate(tspan,x0,p,opt);
23
24  % Initialize a SimulationPlotClass object with the model data.
25  sp = SimulationPlotClass(m);
26
27  % Plot the result of the simulation.
28  sp.plotAllStates(out);
```

4 ModelClass

4.1 ModelClass methods

4.1.1 version()

4.1.2 checkVersion()

4.1.3 update()

5 ModelClass syntax

5.1 SimOptions

The keyword SimOptions allows us to define the default values for the configuration of the ode solver with odeset() and more general values as the time span for the simulation.

The syntax for using this command is as follows:

```
1  SimOptions parameter = value;
```

, where parameter is one of the simulation configuration propierty and value is the value we want to assign to it.

For example, the following code:

```
1  SimOptions AbsTol = 1e-3;
2  SimOptions RelTol = 1e-9;
```

```
3   SimOptions TimeSpan  = [0  10];
```

will set the value of the absolute tolerance AbsTol to 1e-3, the value of the relative tolerance RelTol to 1e-9 and the time span of the simualtion TimeSpan to [0 10].

In the following table is shown the list of simulation configuration propierties currently available:

Propierty	Description	Value
RelTol	Relative error tolerance	1e-3 (default) | positive scalar
AbsTol	Absolute error tolerance	1e-6 (default) | positive scalar | vector
TimeSpan	Simulation time span	[0 10] (default) | positive scalar vector

5.2   Extends

The keyword extends allows us to create new ModelClass models that extends the functionality of a base class.

When we use the keyword extends with the filename of a base model, the parser of the ModelClass will read all the data in the base model and it will add it to the model which we are working on. So it can be understood as pasting all the data from the base model to the new one.

This way we can reuse model for building new models that expand the original functionality, and it a change is made in the original model, it will be also done in the extended models. This helps the maintenance of the models and reduces the time spent programming.

5.2.1   Usage

When defining a ModelClass .mc, we can use the following structure for extending a base model:

```
1   extends /path/to/filename.mc;
```

, where /path/to/filename.mc it the path with the name of the model we want to extend (it is possible to use a relative path or a absolute path). For example:

This command will extend the file model.mc present in the working directory.

```
1   extends ./model.mc;
```

This command will extend the file model.mc present in the parent directory of the working directory.

```
1  extends ../model.mc;
```

This command will extend the file model.mc present in the path /home/user/MATLAB/.

```
1  extends /home/username/MATLAB/model.m;
```

NOTE: It is best to declare the extends at the beginning of the ModelClass model.

5.2.2 Example

In ./examples/ex1_import there is a example code for using the import keyword.

The example starts with a baseModel.mc and we would like to expand its functionality by adding a equation to show the reference dynamically.

The baseModel.mc defines the following variables and equations:

```
1  % Variables
2
3  variable x1;
4  variable x2;
5  variable x3;
6
7  % Parameters
8
9  parameter k1;
10 parameter k2;
11 parameter k3;
12 parameter d1;
13 parameter d2;
14 parameter d3;
15 parameter gamma12;
16
17 % Equations
18
19 equation der_x1 == k1    - gamma12*x1*x2 - d1*x1;
20 equation der_x2 == k2*x3 - gamma12*x1*x2 - d2*x2;
21 equation der_x3 == k3*x1 - d3*x3;
```

And the extendeModel.mc is defines the variable and equation needed for the reference:

```
1  % I would like to represent the reference of the baseModel.mc dynamically.
2
3  % First, extend the functionality defined in baseModel.mc.
4  import ./model/baseModel.mc;
5
6  % Then, add a variable for the reference.
```

```
7  variable ref;
8
9  % And add the equation to calculate the reference value.
10 equation ref == k3/d3;
```

Notice how the equation for the reference ref == k3/d3 makes use of parameters previously defined in the baseModel.mc. The extended model has access to all the information defined in the base model.

Then if we execute the following command:

```
1  m = ModelClass.load('./model/importModel.mc');
```

, the file importModel.m is autogenerated, and it will contain all the model information defined by baseModel.mc and extendedModel.mc. Here is shown the contents of importModel.m:

```
1  classdef importModel < ModelClass
2      % This code was generated by ModelClass v0.4.7 a5ec716   -   ...
           Fernando Nóbel (fersann1@upv.es)
3      methods
4          function [obj] = importModel(opts)
5              v = VariableClass(obj,'x1');
6              obj.addVariable(v);
7
8              v = VariableClass(obj,'x2');
9              obj.addVariable(v);
10
11             v = VariableClass(obj,'x3');
12             obj.addVariable(v);
13
14             p = ParameterClass(obj,'k1');
15             obj.addParameter(p);
16
17             p = ParameterClass(obj,'k2');
18             obj.addParameter(p);
19
20             p = ParameterClass(obj,'k3');
21             obj.addParameter(p);
22
23             p = ParameterClass(obj,'d1');
24             obj.addParameter(p);
25
26             p = ParameterClass(obj,'d2');
27             obj.addParameter(p);
28
29             p = ParameterClass(obj,'d3');
30             obj.addParameter(p);
31
32             p = ParameterClass(obj,'gamma12');
```

```
33                 obj.addParameter(p);
34
35                 e = EquationClass(obj,'');
36                 e.eqn = 'der_x1 == k1      - gamma12*x1*x2 - d1*x1';
37                 obj.addEquation(e);
38
39                 e = EquationClass(obj,'');
40                 e.eqn = 'der_x2 == k2*x3 - gamma12*x1*x2 - d2*x2';
41                 obj.addEquation(e);
42
43                 e = EquationClass(obj,'');
44                 e.eqn = 'der_x3 == k3*x1 - d3*x3';
45                 obj.addEquation(e);
46
47                 v = VariableClass(obj,'ref');
48                 obj.addVariable(v);
49
50                 e = EquationClass(obj,'');
51                 e.eqn = 'ref == k3/d3';
52                 obj.addEquation(e);
53
54                 obj.checkValidModel();
55             end
56
57         end
58         methods(Static)
59             function [out] = isUpToDate()
60                 dependenciesPath = {...
61                     './model/importModel.mc'...
62                     './model/baseModel.mc'...
63                 };
64                 out = importModel.checkUpToDate(dependenciesPath);
65             end
66         end
67     end
```

, where we can confirm that the base model has been successfully imported.

Finally, at this point we can work as usually with the model importModel.mc, and we could even extended it to create a importImportModel.mc!

6 SimulationClass

6.1 Default values for simulating

It is possible to define the default values for parameters, initial conditions, simulation time and ode options in the definition of a model using the ModelClass syntax.

In the following example, it is shown how we can set the intial condition of Variables setting the start propierty, the default value of a Parameter setting the value propierty or

the simulation options using the SimOptions command.

```
1  simOptions AbsTol = 1e-3;
2  simOptions RelTol = 1e-9;
3  simOptions TimeSpan  = [0  10];
4
5  variable x(start = 0);
6  parameter k(value = 1);
7
8  equation der_x == k - x;
```

This is very handy because it allows us to have a single contained file with the structure of the model and everything we need to simulate it.

However sometimes we would want to change some values of the model to perform different simulations. For this, we can override the default values without modifing the model. This can be done by calling the simulate() method with the arguments we want to use in this particular simulation.

```
1  %% Init the model and the tools for simulating
2  m = ModelClass.load('./model/model.mc');
3  s = SimulationClass(m);
4  sp = SimulationPlotClass(m);
5
6  %% Simulate with default values.
7  [out] = s.simulate();
8  sp.plotAllStates(out);
9
10 %% Simulate with user defined values.
11 tspan = 0:10;
12 p.k = 2;
13 x0.x = 1;
14 opt = odeset('AbsTol',1e-1);
15 [out] = s.simulate(tspan,x0,p,opt);
16 sp.plotAllStates(out);
```

6.2 Steady state

6.2.1 Simulate until steady state

Sometimes it is handy to just simulate until the steady state is reached. We can set manually an Event in the ODE options for this. However, the following command will do that for us:

```
1  opt = s.optSteadyState(opt,p,tol);
```

10

,where opt is the options for the ODE solver, s is the SimulationClass object of the model, p is the struct of paramters used in the simulation and tol is the tolerance to determine the steady state. The steady state is reached when the absolute sum of all the derivatives of the model is less than tol. If tol is not defined, it will be set to 1e-6.

6.2.2 Get the steady state

The following method calculates the steady state value of the model directly:

```
1  [out] = s.simulateSteadyState(x0,p);
```

, s is the SimulationClass object of the model, x0 is the struct of initial conditions, p is the struct of paramters used in the simulation and tol is the tolerance to determine the steady state. The steady state is reached when the absolute sum of all the derivatives of the model is less than tol. If tol is not defined, it will be set to 1e-6.

6.2.3 Example

Here is a minimal example of how to use the optSteadyState function:

(i) the model.mc code:

```
1  variable x(start = 0);
2  parameter A(value = 1);
3
4  equation der_x == A - x;
```

and (ii) the main.m code:

```
1  %% Init the model and the tools for simulating
2  m = ModelClass.load('./model/model.mc');
3  s = SimulationClass(m);
4  sp = SimulationPlotClass(m);
5
6  % Define intial state, parameters.
7  x0 = [];
8  p = [];
9
10  % Define the options for the simulator.
11  opt = odeset('AbsTol', 1e-3, 'RelTol', 1e-6);
12
13  %% Simulate normally the model.
14  % This way, we need to define a long enough time span to reach the steady
15  % state.
16
```

```matlab
17  % Simulation time span.
18  tspan = [0 10];
19
20  % Simulate.
21  [out_1] = s.simulate(tspan,x0,p,opt);
22
23  %% Simulate until steady state is reached.
24  % We are going to define an event to stop the simulation when the steady
25  % state is reached.
26
27  % We do not need to worry about simulation time span, as it will stop due
28  % to the event.
29  tspan = [0 +inf];
30
31  % Define the tolerance to determine the steady state.
32  % Try changing this value to see its effect.
33  tol = 1e-1;
34
35  % Set the event for ending the simulation when steady state is reached.
36  opt = s.optSteadyState(opt,p,tol);
37
38  % Simulate.
39  [out_2] = s.simulate(tspan,x0,p,opt);
40
41  %% Calculate the steady state.
42  % We can use this method to calculate the steady state directly.
43
44  % Get the steady state.
45  [out_3] = s.simulateSteadyState(x0,p);
46
47  %% Plot the result and see that the simulation has been stop way before ...
         the
48  % defined time span.
49  figure(1);
50  clf(1);
51
52  hold on;
53  grid on;
54  plot(out_1.t,out_1.x);
55  plot(out_2.t,out_2.x,'--','LineWidth',1.5);
56  plot(out_1.t(end),out_3.x,'o');
57  legend('simulate','optSteadyState','simulateSteadyState','Location','SouthEast');
```