ModelClass User Manual

Fernando Nóbel Santos Navarro (fersann1@upv.es)

September 29, 2020

Contents

1 Manual version

This manual was written for the version **v0.4.3** of ModelClass. Please check your local version of ModelClass using the following command:

```
ModelClass.version();
```

and it should return something similar to this:

```
v0.4.3 4c898f7   -   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* directory 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:

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

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

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

or the follwing message if if the local code is outdated:

```
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:

```
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

```
% Variables

Variable x1;
Variable x2;
Variable x3;
Variable ref(value = k3/d3);

% Parameters

Parameter k1(value = 1.0);
Parameter k2(value = 1.0);
Parameter k3(value = 1.0);
Parameter d1(value = 1.0);
Parameter d2(value = 1.0);
Parameter d3(value = 1.0);
Parameter gamma12(value = 1.0);

% Equations

Equation der_x1 == k1    - gamma12*x1*x2 - d1*x1;
Equation der_x2 == k2*x3 - gamma12*x1*x2 - d2*x2;
Equation der_x3 == k3*x1 - d3*x3;
```

(ii) main.m

```
% Initialize an object of the model.
m = loadModelClass('model');

% Initialize a SimulationClass object with the model data.
```

```
s = SimulationClass(m);

% Simulation time span.
tspan = [0 10];

% Parameters of the model.
p = []; % They are already defined in "model.mc"

% Intial conditions of the model.
x0.x1 = 0.000000;
x0.x2 = 0.000000;
x0.x3 = 0.000000;

% Options for the solver.
opt = odeset('AbsTol', 1e-8, 'RelTol', 1e-8);

% Simulate the model.
[out] = s.simulate(tspan,x0,p,opt);

% Initialize a SimulationPlotClass object with the model data.
sp = SimulationPlotClass(m);

% Plot the result of the simulation.
sp.plotAllStates(out);
```

4 ModelClass

4.1 ModelClass methods

4.1.1 version()

4.1.2 checkVersion()

4.1.3 update()

5 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.

5.1 Usage

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

```
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.

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

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

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

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

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

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

5.2 Example

In `./examples/ex1_extends` there is a example code for using the `extends` 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:

```
% Variables

Variable x1;
Variable x2;
Variable x3;

% Parameters

Parameter k1;
Parameter k2;
Parameter k3;
Parameter d1;
Parameter d2;
Parameter d3;
Parameter gamma12;

% Equations

Equation der_x1 == k1    - gamma12*x1*x2 - d1*x1;
Equation der_x2 == k2*x3 - gamma12*x1*x2 - d2*x2;
Equation der_x3 == k3*x1 - d3*x3;
```

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

```
% I would like to represent the reference of the baseModel.mc dynamically.

% First, extend the functionality defined in baseModel.mc.
extends ./baseModel.mc;

% Then, add a variable for the reference.
Variable ref;

% And add the equation to calculate the reference value.
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:

```
m = loadModelClass('extendedModel');
```

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

```
classdef extendedModel < ModelClass
    methods
        function [obj] = extendedModel()
            v = VariableClass('x1');
            obj.addVariable(v);

            v = VariableClass('x2');
            obj.addVariable(v);

            v = VariableClass('x3');
            obj.addVariable(v);

            p = ParameterClass('k1');
            obj.addParameter(p);

            p = ParameterClass('k2');
            obj.addParameter(p);

            p = ParameterClass('k3');
            obj.addParameter(p);

            p = ParameterClass('d1');
            obj.addParameter(p);

            p = ParameterClass('d2');
            obj.addParameter(p);

            p = ParameterClass('d3');
            obj.addParameter(p);

            p = ParameterClass('gamma12');
            obj.addParameter(p);
```

```
            e = EquationClass('');
            e.eqn = 'der_x1 == k1    - gamma12*x1*x2 - d1*x1';
            obj.addEquation(e);

            e = EquationClass('');
            e.eqn = 'der_x2 == k2*x3 - gamma12*x1*x2 - d2*x2';
            obj.addEquation(e);

            e = EquationClass('');
            e.eqn = 'der_x3 == k3*x1 - d3*x3';
            obj.addEquation(e);


            v = VariableClass('ref');
            obj.addVariable(v);

            e = EquationClass('');
            e.eqn = 'ref == k3/d3';
            obj.addEquation(e);

        obj.checkValidModel();
        end
    end
end
```

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

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

6 SimulationClass

6.1 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:

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

,where `opt` is the options for the ODE solver, `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 `0.001`.

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

(i) the `model.mc` code:

```
Variable x(start = 0);
Equation der_x == 1 - x;
```

and (ii) the `main.m` code:

```
% Init the model and the tools for simulating
m = loadModelClass('model');
s = SimulationClass(m);
sp = SimulationPlotClass(m);

% Define intial state, parameters.
x0 = [];
p = [];

% Simulation time span (note that we have set a huge time span).
tspan = [0 100000];

% Define the options for the simulator.
opt = odeset('AbsTol', 1e-10, 'RelTol', 1e-10);

% Define the tolerance to determine the steady state.
% Try changing this value to see its effect.
tol = 0.01;

% Set the event for ending the simulation when steady state is reached.
opt = s.optSteadyState(opt,p,tol);

% Simulate the model.
[out] = s.simulate(tspan,x0,p,opt);

% Plot the result and see that the simulation has been stop way before the
% defined time span.
sp.plotAllStates(out);
```