

TuneForLD_Qshape

Modelica package for benchmark testing of PID tuning rules
targeted to disturbance rejection

– minimal documentation –

Foreword

This is a *really* minimal documentation for the Modelica package TuneForLD_Qshape.mo. The package was written and tested with the OpenModelica translator, available as free software at <https://openmodelica.org/>, but in principle it should function correctly with any translator. To keep it as simple as possible no graphical annotations were used, hence the user is expected to interact with the package entirely in textual view.

The rest of this short document aims at putting the user in the position of

1. running all the simulations in the paper “Explicit model-based real PID tuning for efficient load disturbance rejection”, to which the package is companion,
2. running other simulation with modified parameters,
3. and expanding the package with additional model structures, model parametrisation methods, tuning rules, and controllers.

This document tries to provide the necessary explanations without requiring any Modelica knowledge. However, for the more articulated tasks (not for just reproducing the paper results or trying different parameters as per items 1 and 2 above) the interested user may take profit from acquiring some such knowledge. In this case, a useful reference is the book "Modelica by Example" by M.M. Tiller, available online at <https://book.xogeny.com/>.

Also, no theoretical stuff is treated herein. For this matter, please refer to the companion paper just mentioned.

Launching OpenModelica and loading the package

It is assumed that the user has installed OpenModelica as per the instructions available at <https://openmodelica.org/>, and decompressed the TuneForLD_Qshape.zip file in the folder of his/her choice.

The first action to take is launching the OMEdit application. This application allows one to load and edit Modelica packages (hierarchically organised collections of models), run

simulations, visualise and export results, and so forth. To launch OMEdit, locate the corresponding icon on the desktop, start menu, dock, or the like (the OpenModelica suite, including OMEdit, is available for Windows, MacOS and Linux, and details may differ) and double-click on it.

This action opens the application, from which the user has to select the menu entry File → Open Model/Library File(s), then navigate to the folder that contains the package TuneForLD_Qshape.mo, and finally click Open. Once this is done, the OMEdit window should have the aspect shown in Figure 1.

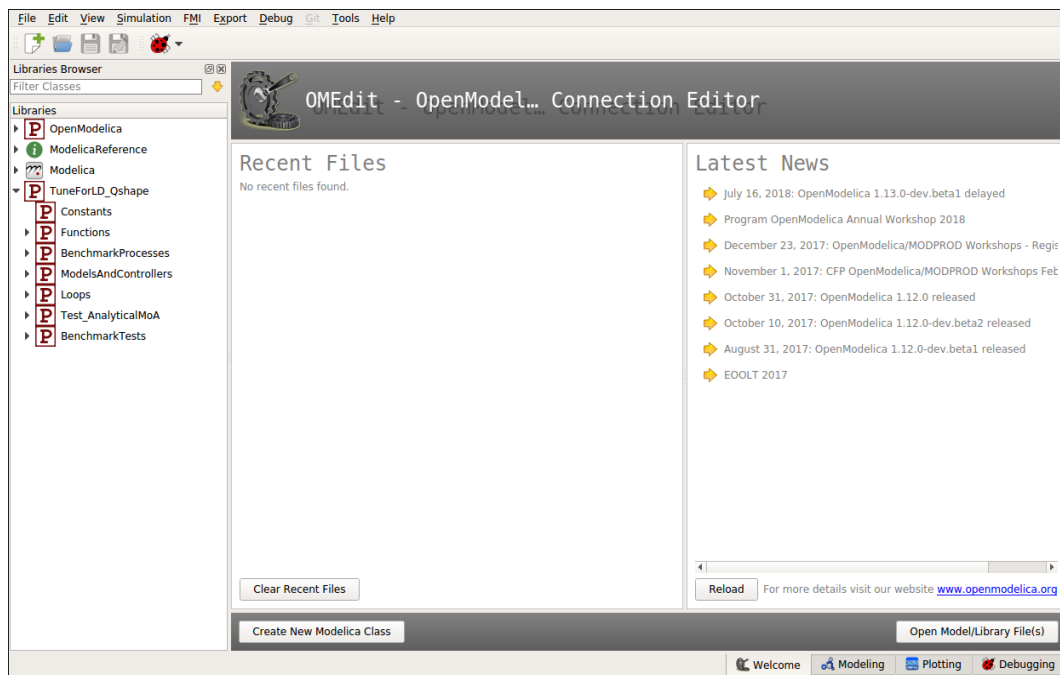




Figure 1 – the OMEdit window with TuneForLD_Qshape.mo loaded.

The TuneForLD_Qshape package is organised in sub-packages, as explained below, and can be navigated by expanding/collapsing branches in the tree view visible in the left pane of the window, like any file manager: (sub)packages – the equivalent of (sub)folders in the file manager metaphor – have a  icon, while models – the equivalent of files – an  one. Double-clicking on a model opens it in the right pane, that switches to "Modeling" (see the bottom part of the OMEdit window). Multiple models can be opened, and in this case each one of them gets its own tab in the right pane of the OMEdit window.

Running the first simulation

We start by running the paper test corresponding to the process class P_2 of the benchmark proposed in (Åström and Hägglund, 2000). To this end, open the LoadDistRes_Loop_P2 model in the BenchmarkTests package (in the left pane double-click on BenchmarkTests and when this expands, on LoadDistRes_Loop_P2). This leads to the situation shown in Figure 2; the black arrow indicates that the Modelling view is active. It may be necessary to also click on the Text view icon in the right pane sub-menu (indicated in Figure 2 by a red arrow).

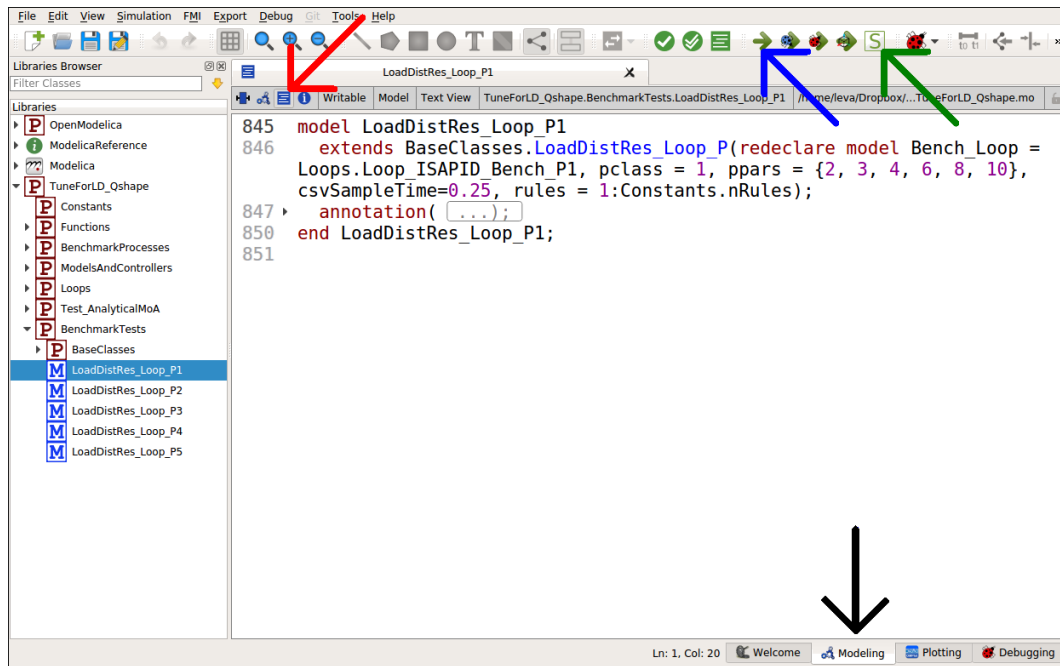


Figure 2 – opening the model LoadDistRes_Loop_P2.

To simply run the test in the paper, click on the Simulate icon, indicated in Figure 2 by a blue arrow. This compiles and runs the model – taking a bit of time because all the compared rules and all the model parameter values are treated in a single simulation run, so please be patient – and then moves to the Plotting view (indicated by the tab at the bottom right of the window).

The Plotting window has a variable browser on the right. There are a lot of variables in the model, but for the scope of this paragraph we are only interested in the $y[i,j]$ ones, that are found by navigating to the bottom of the variable list.

By ticking/unticking the box right of the variable name, the corresponding curve is added/removed to the current plot. The first index is that of the parameter value and the second that of rule, both in the sets selected in the model. To exemplify, since the model text – see again figure 2 – says

`ppars = {2,3,4,6,8,10};`

and

`rules = 1:Constants.nRules;`

– that is, all the available rules

then $i=2$ and $j=5$ mean selecting the result corresponding to the model with the parameter set to 3 (the second value in `ppars`) and yielded by rule number 5. If for example the user modified the model to make it read

`ppars = {2,5,6,8,10,12};`
`rules = {1,2,4,5,8,9};`

then $i=2$ and $j=5$ would select the result of rule number 8 applied to the model with the parameter set to 5. Figure 3 shows the Plotting view with some results. There are a

number of controls to filter variables, create plot windows, zoom, pan, and so forth. Most of these interface elements are intuitive and therefore we don not spend further words on them; for full details refer to the OpenModelica documentation and help.

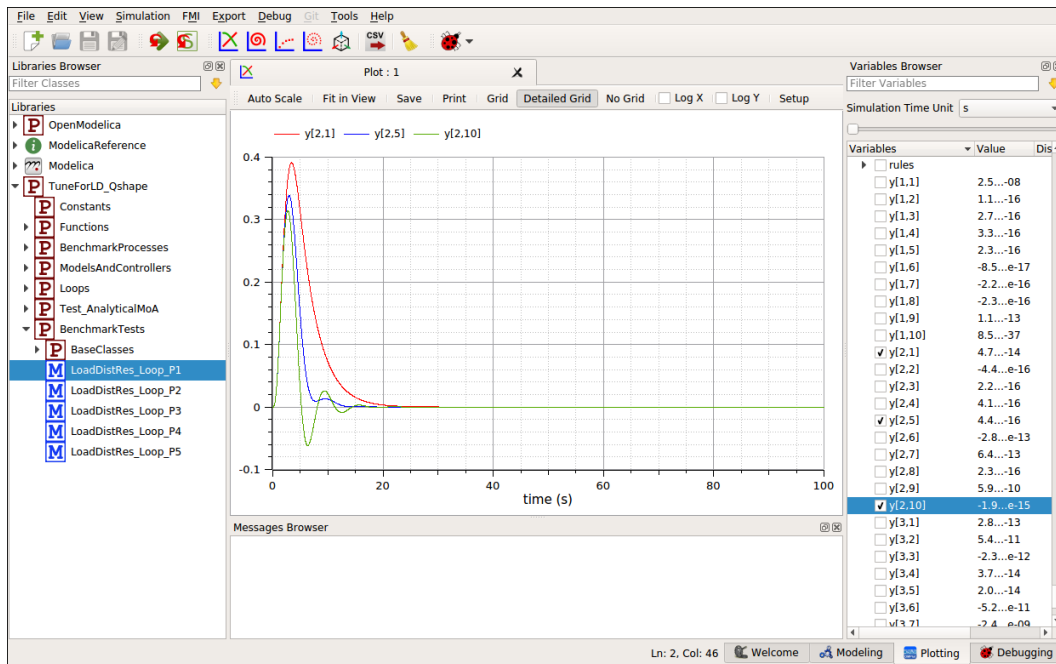


Figure 3 – the Plotting view with some results selected.


Running different simulations

The way to run additional tests by just changing parameters is quite straightforward. One first selects the process class to consider by

```
Bench_Loop = Loops.Loop_ISAPID_Bench_P<desired_class_No>,
pclass      = <desired_class_No>,
```

where the need for the apparent redundancy is too long to explain here. This done, one selects the process parameter values and the rules to employ as already shown, and finally decides the sampling time in the .csv result file produced to contain the simulated transients by

```
csvSampleTime = <desired_sampling_time_in_seconds>
```

One can also set additional simulation-related parameters by means of the dialog window obtained by clicking on the  icon (indicated by a green arrow in Figure 2) in the toolbar.

The dialog, easy to interpret for anyone familiar with simulation tools, is shown in Figure 4. For the purpose of the activity described herein it is advisable to just set the simulation stop time, and leave the other parameters to the proposed defaults.

For particularly critical cases from the numerical standpoint one can also intervene on the tolerances, and also select the solver to employ. Once again, full detail on all these aspects can be found in the OpenModelica documentation.

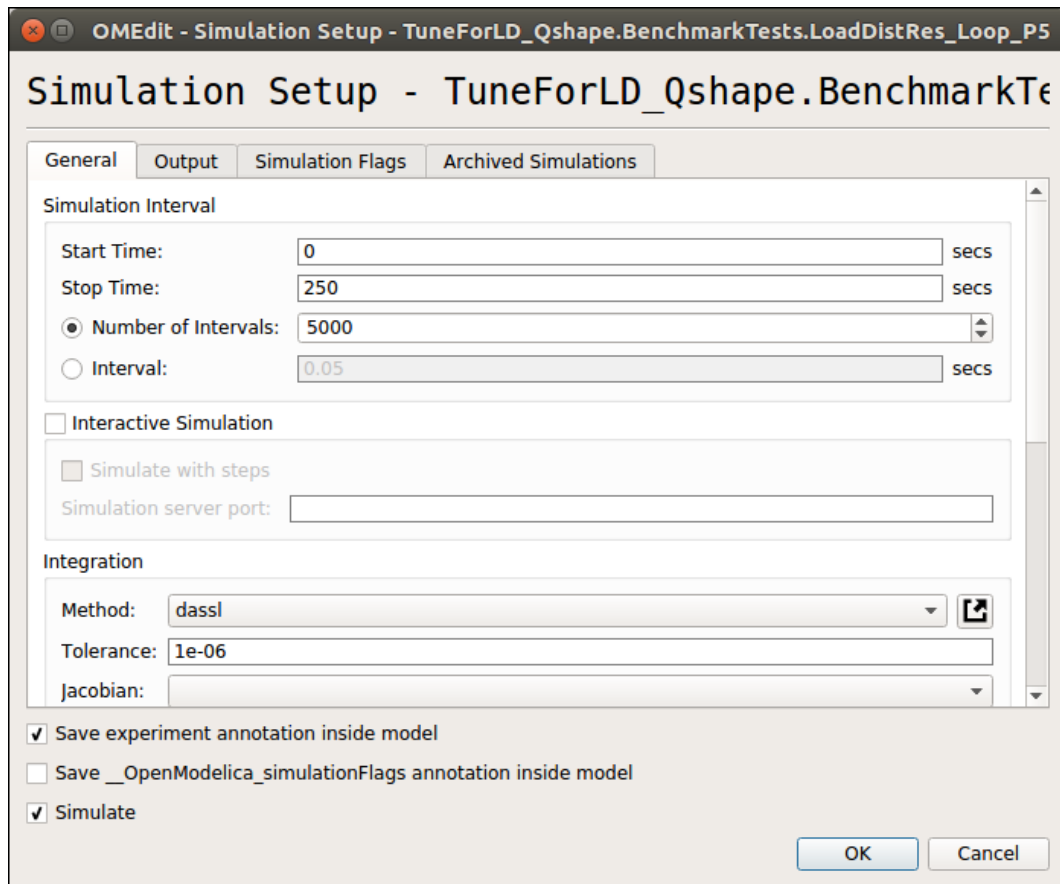


Figure 4 – the simulation setup dialog.

When a test is run, as anticipated, five .csv files are created with the names

class_C_param_P.csv

ISE_pclass_C.csv

IAE_Pclass_C.csv

ITAE_Pclass_C.csv

LRPI_Pclass_C.csv

to contain respectively the simulated transients, the ISE, IAE, ITAE and LRPI values (see the paper for the definition of these indices). Incidentally, this is how the data presented in the paper to which the package is companion were obtained.

The .csv files are put in the OMEdit working directory, that can be at different locations depending on the operating system, but is accessed directly by OMEdit by selecting Tools → Open Working Directory from the menu.

An overview of the package and possible extensions

We now give a brief an overview of the TuneForLD_Qshape.mo package, to provide the essential information for possible modifications and extensions.

Subpackage Constants

This defines some constants used in the rest of the package. At present it reads

```
package Constants
  constant Integer nClasses = 5  "No. of process classes";
  constant Integer nRules   = 10 "No. of tuning rules";
  constant Integer pmminRp  = 30 "default min pm for proposed rule";
end Constants;
```

and should be modified if for example a new class or a new rule were added.

Subpackage Functions

This contains some internal functions that we do not describe here, if not for mentioning one as an exception in the following section, the functions to analytically compute an FOPDT model with the method of areas, and those for the tuning rules.

If one wants to add a new model class, in addition to increase Constants.nClasses, he/she needs to add the corresponding function that parametrises the FOPDT model to the Functions.AnalyticalMoA package and name it MoA_Px, where x is the number of the new process class. The class must have one parameter, and the MoA_Px function must take this one Real parameter as input, and return the gain, time constant and delay of the FOPDT model (μ , T and D, respectively) as Real outputs.

Below the MoA_P1 function is reported as a template to follow. Notice that n is converted to Integer but taken as Real, because all the MoA_Px functions must have the same I/O structure to allow replaceability when assembling the compound models to run the tests.

```
function MoA_P1 "FOPDT approx for 1/(1+s)^n"
  input Real p;
  output Real mu, T, D;
protected
  Integer n;
algorithm
  n := integer(p);
  mu := 1;
  T := exp(1-n)*n^n /Functions.Internal.factorial(n-1);
  D := n - T;
end MoA_P1;
```

The Functions.AnalyticalMoA package also contains a comprehensive wrapper function, that selects the appropriate MoA_Px function to call when receiving the process class number and the parameter value as inputs.

If one adds a new process and analytical method of areas function, he/she has to also add that function to the wrapper, as briefly explained by the following listing.

```

function MoA_P_all
  "FOPDT approx for process of class c with parameter value p"
  input Integer c(min = 1, max = Constants.nClasses)
  "process class (1-Constants.nClasses)";
  input Real p "process param value";
  output Real mu, T, D;
algorithm
  if c == 1 then
    (mu,T,D) := Functions.AnalyticalMoA.MoA_P1(p);
  elseif c == 2 then
    ...
    "other cases"
    ...
  elseif c == <past values of nClasses> then
    (mu,T,D) := Functions.AnalyticalMoA.MoA_P_old_last(p);
  else
    (mu,T,D) := Functions.AnalyticalMoA.MoA_P_new_process(p);
  end if;
end MoA_P_all;

```

The Functions.TuningRules package, quite intuitively, contains the tuning rules. Names here are free, although maintaining uniformity is advisable. Functions must however take three Real inputs (mu, T and D) and return the parameters K, Ti, Td and N of a real PID in the ISA form (which is expected by the control loop models).

When a rule does not return an ISA PID, a parameter conversion function exists in the Functions.Internal package. As an example, the AMIGO rule corresponds to the listing below.

```

function PID_AMIGO "AMIGO, ISA PID"
  input Real mu, T, D;
  output Real K, Ti, Td, N;
algorithm
  K := 1/mu*(0.2+0.45*T/D);
  Ti := (0.4*D+0.8*T)/(D+0.1*T)*D;
  Td := 0.5*D*T/(0.3*D+T);
  N := 20;
end PID_AMIGO;

```

As an “advanced” topic, if a rule has a tuning parameter, the way to go is to declare it optional in the function, setting a default in the Constants package. The reader willing to do this, surely has enough Modelica knowledge to figure out the way by looking at how the tuning rule proposed in the paper is implemented. We do not delve here into further details for brevity and readability.

Back to the main topic, to add a new rule, besides increasing Constants.nRules, one has to create the new tuning function (like PID_AMIGO above, and maintaining the I/O structure) and add it to the comprehensive wrapper ISAPID_pclass_ppar_trule, analogous to the previously described MoA_P_All; the main lines of ISAPID_pclass_ppar_trule are reported in the following, to explain what needs to be done.

```

function ISAPID_pclass_ppar_trule
    "tune ISA PID for process class c and parameter value p,
    with rule r"
    input Integer c(min = 1, max = Constants.nClasses)
        "process class (1-Constants.nClasses)";
    input Real p "process class parameter";
    input Integer r(min = 1, max = Constants.nRules)
        "tuning rule (1-Constants.nRules)";
    output Real K, Ti, Td, N;
protected
    Real mu, T, D;
algorithm
    (mu,T,D) := Functions.AnalyticalMoA.MoA_P_all(c, p);
    if r == 1 then
        (K,Ti,Td,N) := Functions.TuningRules.PID_KS88IAEr(mu,T,D);
    elseif r == 2 then
        ...
    elseif r == <past value of nRules> then
        (K,Ti,Td,N) := Functions.TuningRules.LAST_OLD_RULE(mu,T,D);
    else
        (K,Ti,Td,N) := Functions.TuningRules.YOUR_NEW_RULE(mu,T,D);
    end if;
end ISAPID_pclass_ppar_trule;

```

Subpackage BenchmarkProcesses

This just contains the processes, named Bench_Px, and described as continuous-time systems with input u and output y; in Modelica the time derivative operator is der(). Model Bench_P2 is shown below as example: here extensions are straightforward.

```

model Bench_P2
    "1/((1+s)(1+ps)(1+p^2s)(1+p^3s), process class 2 in
    Astrom & Hagglund, 2000"
    parameter Real p = 1;
    input Real u;
    output Real y;
protected
    Real[4] x(each start = 0, each fixed = true);
equation
    der(x[1]) + x[1] = u;
    for i in 2:4 loop
        p^(i-1)*der(x[i])+x[i] = x[i-1];
    end for;
    y = x[4];
end Bench_P2;

```


Subpackage ModelsAndControllers

This subpackage contains the FOPDT model and the ISA PPID controller. No comments are needed in this document.

Subpackage Loops

This package contains one model per class, obtained by extending the partial model in the enclosed subpackage BaseClasses. The model for each class is composed of the corresponding process and an ISA PID controller, forming the loop, with inputs r and d (set point and load disturbance) and poutputs y and u (controlled variable and control signal). For example, the loop model for class P_1 reads

```
model Loop_ISAPID_Bench_P1
  extends BaseClasses.Loop_ISAPID_Bench_P(
    redeclare model Bench_P = BenchmarkProcesses.Bench_P1
  );
end Loop_ISAPID_Bench_P1;
```

Extending to additional classes just requires adding another such model and adapting the redeclare statement accordingly.

Subpackage Test_AnalyticalMoA

This package contains models used to test the analytical application of the method of areas. The results of these models are not included in the paper, but the package was left to allow the interested user to test the method, possibly also applied to the additional process classes he/she would like to introduce as extensions.

Subpackage BenchmarkTests

This is structured in the same way as Loops, i.e., with a base class extended by means of redeclare statement. The model for class P_1 is reported below as an example, and the way to add other classes/rules stems directly.

```
model LoadDistRes_Loop_P1
  extends BaseClasses.LoadDistRes_Loop_P(
    redeclare model Bench_Loop = Loops.Loop_ISAPID_Bench_P1,
    pclass          = 1,
    ppars           = {2,3,4,6,8,10},
    csvSampleTime   = 0.25,
    rules           = 1:Constants.nRules
  );
end LoadDistRes_Loop_P1;
```