RTC-Tools 2

A toolbox for Modeling Real-Time Control

Tutorial

Bernhard Becker, Maarten Smoorenburg and Jorn Baayen

Version: 1.0
Revision: 123

14 July 2016

RTC-Tools 2, Tutorial

Published and printed by:
Deltares
Boussinesqweg 1
2629 HV Delft
P.O. 177
2600 MH Delft
The Netherlands

telephone: +31 88 335 82 73
fax: +31 88 335 85 82
e-mail: info@deltares.nl
www: https://www.deltares.nl

Contact:

Bernhard Becker
telephone: +31 6 5241 6736
fax: +31 88 335 8582

Jorn Baayen
telephone: +31 6 1162 6941
fax: +31 88 335 8582

e-mail: rtc-tools@deltares.nl
www: http://oss.deltares.nl/web/rtc-tools

Copyright © 2016 Deltares
All rights reserved. No part of this document may be reproduced in any form by print, photo print, photo copy, microfilm or any other means, without written permission from the publisher: Deltares.

Contents

1 Preface **1**

2 Installation **3**

3 Introduction to Python **5**
3.1 Basic concepts 5
3.2 Exercise 6
3.2.1 Task 6
3.2.2 Solution 7

4 Modelica **9**
4.1 Introduction and basic concepts 9
4.2 Exercise 11
4.2.1 Task 11
4.2.2 Solution 12

5 A simple *RTC-Tools* model predictive control model example **15**
5.1 Introduction 15
5.2 The physical part: *Modelica* 15
5.3 The *Python* master script for the optimization problem 19
5.3.1 Introduction 19
5.3.2 Import of packages 21
5.3.3 Class declaration and initialization 21
5.3.4 The constructor 21
5.3.5 The objective function 22
5.3.6 Constraints and variable bounds 22
5.3.7 Run the optimization problem 22
5.4 Run the *Python* master script 23
5.5 Input data and results 24

6 References **27**

1 Preface

This tutorial explains how to build a model predictive control system in *RTC-Tools 2* and introduces the basic concepts of *RTC-Tools 2*. Basic knowledge on mathematical optimization in general and model predictive control in particular is required. Furthermore, basic experience with computer code, preferrably *Python*, is useful.

RTC-Tools 2 comprises the following third-party components:

- the *JModelica.org* compiler
- *CasADi*
- *IPOPT*

The *RTC-Tools 2* user specifies the equations within the *Modelica* sotware. The user can choose from pre-defined components (“models” in *Modelica* terms) for the modeling of

- reservoirs
- nodes and branches of an open channel flow network
- hydraulic structures like weirs and orifices
- pumps.

Modelica was chosen because users can specify components for their individual needs, like

- agricultural models
- population growth models
- unit outages to model maintenance actions in hydro power plants
- ...

Modelica works on is a declarative, equation-based language. This means that the user does not have to develop all the computer code for his/her mathematical model, but can specify the mathematical equations to be used.

The compiler, *JModelica.org*, compiles the *Modelica* equations to a symbolic representation. These symbols are differentiated with the help of the algorithmic differentiation package *CasADi* (**??**). The equations and their first- and second-order derivatives are provided to the optimizer. *RTC-Tools 2* is generally used with *IPOPT*, but may also be integrated with commercial packages such as *Gurobi*.

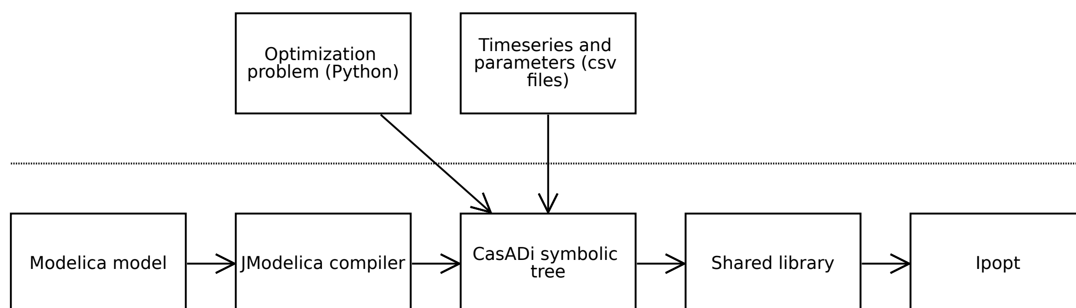

Figure 1.1: *Software components of RTC-Tools 2*

2 Installation

Run the file `RTCTools2Installer.exe`. This installs the following software on your computer:

- *JModelica.org*
- *Python 2.7*
- *CasADi*
- the *OpenModelica Connection Editor*
- the scientific plotting package *Veusz*
- the *Python* development environment *Spyder*

The installation tree will look like this:

- `c:\RTCTools2`
  - `mo`
  - `system`
    - `JModelica`
    - `OpenModelica`
    - `python27`
  - `tutorial`

3 Introduction to Python

3.1 Basic concepts

The hello world programme in *Python*:

```
1 print('Hello world!')
```

A simple for loop prints the word “Hooray” three times:

```
1 for i in range(3):
          print ('Hooray')
```

Note that *Python* does not require a type declaration for the variable `i`, and that indexing is zero-based. The range of 3 (i. e.: `range(3)`) returns a list of integers starting from 0:

```
[0,1,2]
```

Different to other programming languages, instead of brackets *Python* uses indentations. Statements close with a colon. Function arguments are in round brackets. Commands do not need to be closed with a semicolon.

Python is dynamically typed. This means, that types are implicit (more or less).

```
1 c = 1.4142                          # Float
  n = 3                               # Integer
3 b = True                            # Boolean
  s = 'Hello'                         # String
5 l = [1, 2, 3,]                      # List (of Integers)
  d = {'key1': 'Zebra', 'key2': 'Horse', 'key3': 'Donkey'}
7                                     # Dictionary (key-value
                                         pairs)
```

The search for `key1`

```
1 print(d['key1'])
```

returns ‘Zebra’.

It is important to be aware of the distinction between integer and float division:

```
1 a = 3
  b = 2
3 c = 2.0
  print(a / b)
```

This returns ’1’, the largest whole integer of the result, meaning the decimals are disregarded because the result of an operation of two integers is also an integer. However:

```
print(a / c)
```

returns ’1.5’ because *Python* recognizes the type mismatch and converts the integer variable ’a’ to a float before conducting the division operation. Note that in *Python 3* float division is the default behaviour.

If-statement example:

```
if x > 1:
        print ('x is greater than 1')
```

returns “x is greater than 1”.

The for-loop

```
for s in ['a', 'b', 'c', 'd']:
        print(s)
```

returns “a b c d”

```
for i in range(10):
        print(i)
```

returns “0 1 2 3 4 5 6 7 8 9”

The following example shows how to define a class:

```
class Example:
        def __init__(self, message):
                self.message = message
        def say(self):
                print(self.message)
```

`__init__` is the built-in name of the constructor method. The constructor is called on object creation.

`message` is an attribute of the object. The first argument to a method is the object itself.

`say` is a method for all instances of the class.

To create an object instance of our 'Example' class:

```
ex1 = Example('hello1')
```

A method of a class instance is called as follows:

```
ex1.say() # Output: hello1
```

For a single class multiple object instances may exist:

```
ex2 = Example('hello2')
ex2.say() # Output: hello2
```

3.2 Exercise

3.2.1 Task

1. Create a class `Creature`, with a variable `Color`.
2. Create a subclass of `Creature` called `Fish`.
3. Create a subclass of `Creature` called `Crab` with a variable `number_legs`
4. Create a class `Sea` with a list of creatures, and a method `Catch` with an argument `Color` returning a list of matching creatures.

5 Test your classes by creating a Sea with several Fish and Crabs of different colors. Call the `Catch` method with some colors, and output the number of matching creatures.

3.2.2 Solution

We here show how the (sub)classes can be defined in different files, and instances of these classes are called inside another file (all files must be located in the same folder). For reasons of code reusability, it is generally advisable to separate the files that define classes or other objects from those calling them (scripts). This solution also serves to illustrate also some commonly used pythonic idiomes not yet discussed.

1 Open the *Python* development environment *Spyder*.
2 Create a file `sea_creatures.py` and add the `Creature` class:

```
class Creature:
        def __init__(self, color):
                self.color = color
```

Note that file names are in lower case and use underscores by convention.
3 Add the `Fish` subclass below the `Creature` class:

```
class Fish(Creature):
        pass
```

4 Add the `Crab` subclass below this (note the added `number_legs` variable and how it is made a sublass of `Creature`):

```
class Crab(Creature):
        def __init__(self, color, number_legs):
                Creature.__init__(self, color)
                self.number_legs = number_legs
```

5 Make a file called `sea.py` and add the `Sea` class, with a method for counting the number of creatures with a specific color that are found in the sea; a collection of `Creature` objects, for example a `list` object. Also note how the counting is done.

```
class Sea:
        def __init__(self, creatures):
                self.creatures = creatures

        def catch(self, color):
                '''return the number of creatures with a
                    particular color with a group
                of creatures'''
                n = 0
                for creature in self.creatures:
                        if creature.color == color:
                                n += 1
                return n
```

6 Make a file called `main.py` and import the `SeaCreatures` and `Sea` modules:

```
from sea_creatures import Crab, Fish
from sea import Sea
```

Then define some crabs and fishes; note the different ways this is done here.

```
# Define some crabs, stating the attributes explicitly:
c1 = Crab(color='red', number_legs=8)
c2 = Crab(color='blue', number_legs=8)
```

```
4
# Define a group of fishes using list apprehension (a
    short loop), without
6 # stating the attributes explicitly
fishes = [Fish(c) for c in ['red', 'green', 'blue']]
```

Finally, define a Sea (list object) and print the number of creatures per color by running the script:

```
1 sea = Sea(fishes + [c1, c2])
for color in ['red', 'green', 'blue']:
3         print('There are {} {} creatures in the sea'.
              format(sea.catch(color), color))
```

4 Modelica

4.1 Introduction and basic concepts

Modelica is a mathematical modeling language. The basic element are equations.

The differential equation

$$\dot{x} = kx \tag{4.1}$$

or

$$\frac{\mathrm{d}x}{\mathrm{d}t} = kx \tag{4.2}$$

looks in a *Modelica* model as follows:

```
model Example //comment
        parameter Real k = -1.0;
        Real x(start = 1.0);
equation
        der(x) = k * x;
end Example;
```

A *Modelica* model consists of one or multiple equation(s) and the variable declarations. Basically, variables are treated as time variant, like the variable x in the current example. A time-invariant variable is called parameter and declared accordingly. The value of k does, unlike x, not change over time. Note that the equality sign ("=") is not a declaration like in programming languages, but declares an equality in a mathematical sense between the expressions left and right of the equality sign.

The simulation parameters given in Table 4.1 produces the simulation results plot in Figure 4.1. The start value for x that corresponds to the starting time has been set to 1.0 in the current model.

Modelica knows different data types, among others:

- Real
- Integer
- Boolean
- String

The relation to the environment is defined with the help of the following statements:

- `input` indicates that a variable is to be provided by the environment, i. e. another model or an external time series
- `output` says that this variable is published outside the model and can be used by other models.

Table 4.1: *Simulation parameters*

parameter	value
start time	0
stop time	10
interval	0.1

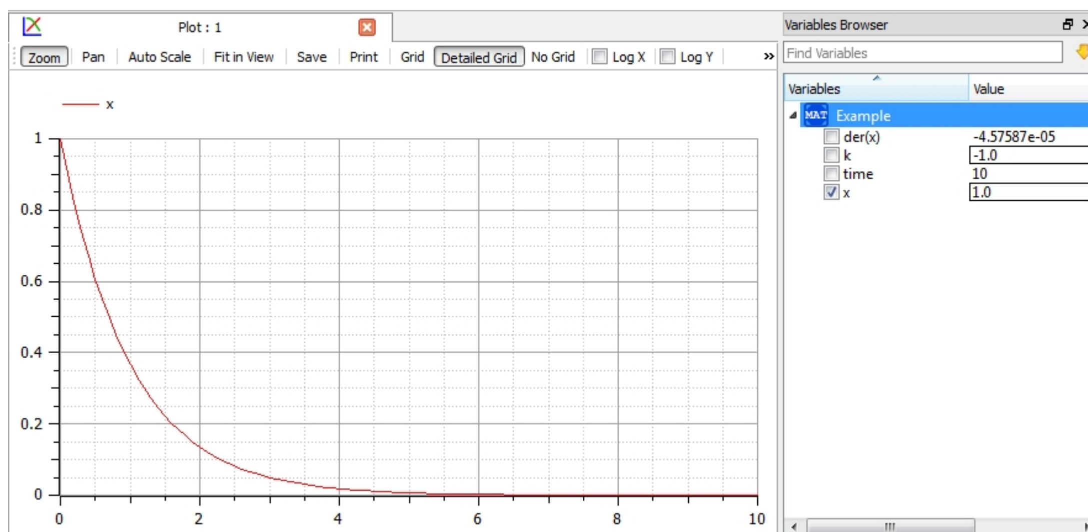

Figure 4.1: *Simulation result plot for the Modelica model of Equation 4.1 in the open Modelica editor OMEdit*

It is good modeling practice to use real-type variables annotated with a physical SI unit. The package `Modelica.SIunits` provides real types with SI units.

Models must be balanced, i. e. the number of equations must be equal to the number of non-input, non-constant variables. Only balanced models can be solved numerically.

Models can inherit from other models with the `extends` statement:

```
model MoreComplexExample
  extends Example;
  Real z;
equation
  z^2 = x;
end MoreComplexExample;
```

Models that are not complete (i. e. not balanced), but are meant to be inherited by other models, are referred to as *partial* models.

Nesting of models is possible by declaring the child model as a variable in the parent model:

```
model ChildModel // child model
  parameter Real k = 1.0;
  Real x(start = 0.0);
  input Real u; // needs an input for u from another model
equation
  der(x) = k * x + u;
end ChildModel;

model ParentModel // parent model
  ChildModel s; // declaration of s refers to child model
equation
  s.u = sin(time); // an equation for the input variable u
      in the child model
end ParentModel;
```

A connector is an interface for a model to handle the exchange of variables.

```
connector HQPort "Connector with potential water level (H)
    and flow discharge (Q)"
  Modelica.SIunits.Position H "Level above datum";
  flow Modelica.SIunits.VolumeFlowRate Q "Volume flow (
     positive inwards)";
end HQPort;
```

In this example, two variables `H` and `Q` are defined. The connector makes sure that variables that represent a potential H are equal (by default all variables are interpreted as potential) and that the sum of flux variables indicated as `flow` Q sum up to zero:

$$H_i = H_j \tag{4.3}$$

$$\sum Q_i = 0 \tag{4.4}$$

Consequently, for flow variables a sign convention is required. For *RTC-Tools 2* models this convention is

- inflow towards an element (i. e. a *Modelica* model) is positive
- outflow from an element is negative.

Two models are connected with the help of the `connect` statement. Two models `model1` and `model2` based on the partial model `HQTwoPort`, which has been designed for models that are to be connected on two sides

```
partial model HQTwoPort "Partial model of two port"
  HQPort HQUp;
  HQPort HQDown;
end HQTwoPort;
```

are connected with the `connect` statement as follows:

```
connect (model1.HQUp, model2.HQDown);
```

4.2 Exercise

4.2.1 Task

To get familiar with *Modelica*, we build a simple population model for a biological equilibrium of wolves and sheep.

1. Create a partial model “Population” with state variable x (i. e. the sheep population), input variable u for a foreign population, a growth parameter r and a probability p for the meeting of wolves and sheep.
2. Create a model “SheepPopulation” that extends “Population” with the equation

$$\frac{\mathrm{d}x}{\mathrm{d}t} = rx - pxu \tag{4.5}$$

   Set the growth parameter to $r = 0.04$ and the initial population to 100.
3. Create a model “WolfPopulation” by extending “Population” with the equation

$$\frac{\mathrm{d}x}{\mathrm{d}t} = rpxu - dx \tag{4.6}$$

   where d denotes a mortality rate of 0.2. the growth rate is $r = 0.8$ and the start population is 30.

Table 4.2: *Simulation parameters for the wolf sheep population model. One time step represents one year.*

parameter	value
start time	0
stop time	250
interval	1

4 Connect the two population models in a master model “World”: the wolf population is the input for the sheep population model, and vice versa. The probability of meeting is $p = 0.0003$.

5 Simulate the model for a time span of 250 years. Plot both the sheep and the wolf population in a single graph.

4.2.2 Solution

1 Open the *Modelica* editor *OMEdit*. If the *RTC-Tools 2* has been installed correctly, you can open *OMEdit* via the *Windows* start menu.

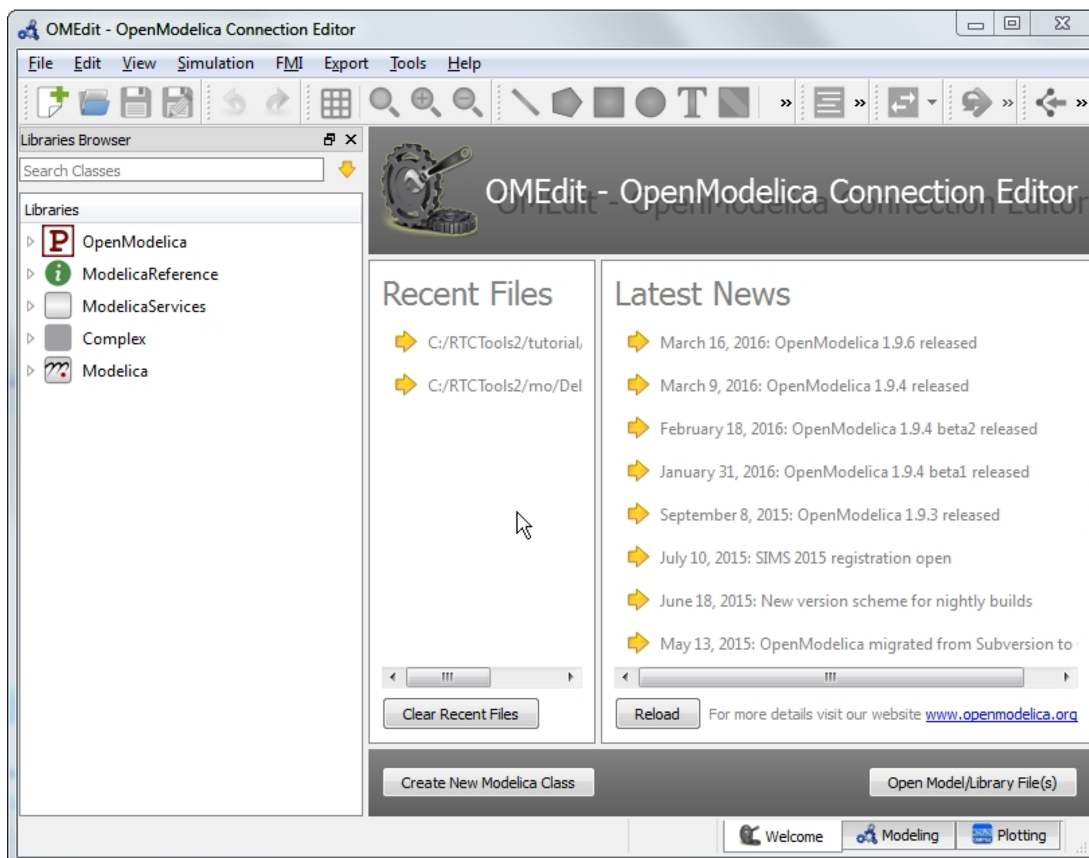

Figure 4.2: *OMEdit start screen*

Create a new *Modelica* class via `File→New Modelica Class` and name it “Population”. Code the population model (task 1) as follows:

```
1 partial model Population
    // Population
3   Real x;
    // Foreign population
```

```
5   input Real u;
    // Growth factor
7   parameter Real r;
    // Wolf-sheep meeting probability factor
9   parameter Real p;
  end Population;
```

2 Create a new *Modelica* class for the sheep population model:

```
  model SheepPopulation
2   extends Population(r = 0.04, x(start = 100)); // set
       initial population of sheep and growth parameter
  equation
4   der(x) = r * x - p * x * u;
  end SheepPopulation;
```

3 Now prepare the wolf population model:

```
1 model WolfPopulation
    extends Population(r = 0.8, x(start = 30));
3   // mortality rate
    parameter Real d = 0.2;
5 equation
    der(x) = r * p * x * u - d * x;
7 end WolfPopulation;
```

4 The “world model” that connects the population of wolves and sheep looks as follows:

```
1 model World
    parameter Real p = 0.0003;
3   WolfPopulation wpl(p = p);
    SheepPopulation spl(p = p);
5 equation
    wpl.x = spl.u;
7   spl.x = wpl.u;
  end World;
```

Here we declare the probability that wolves and sheep meet and define a value. This parameter is the same in both populations. In the equation section the two population models are connected via the equality statement and the corresponding variables: the state (i. e. the state) of the first model is the input of the second model and vice versa.

5 Before running the model, the simulation parameters listed in Table 4.2 must be entered in the Simulation Setup (`Simulation→Simulation Setup`, in the `Simulation Interval` tab). After saving and checking the model, the simulation can be started. The simulation results are presented in Figure 4.3; Note how the (state) variable x is selected in the `Variable Browser` pane for plotting both the sheep and wolf populations.

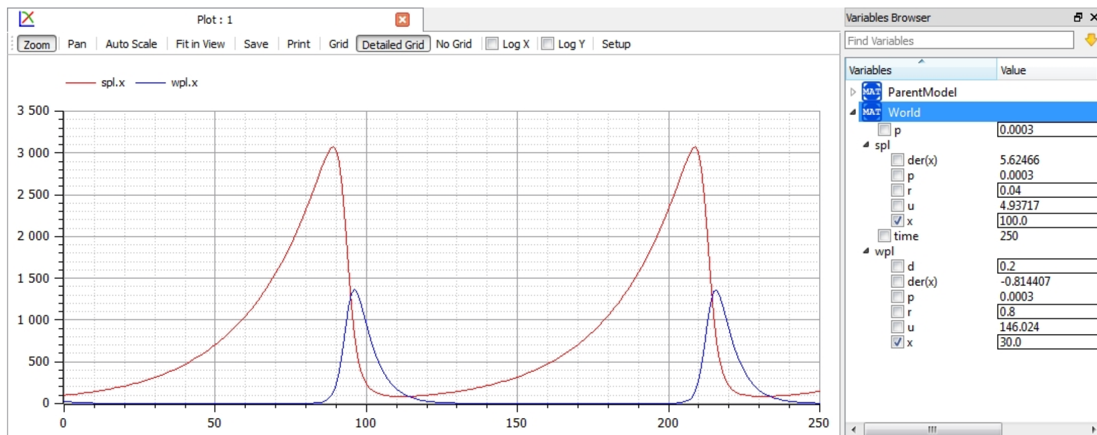

Figure 4.3: *Sheep and wolf population over time, OMEditor*

5 A simple *RTC-Tools* model predictive control model example

5.1 Introduction

The folder `<installation directory>\RTCTools2\tutorial` contains a complete *RTC-Tools 2* model. The purpose of this chapter is to understand the technical setup of an *RTC-Tools 2* model with the help of this example, to run the model, and to understand the results.

The directory has the following structure:

- `input` with the model input data. These are several files in comma separated value format (`csv`).
- `model` contains the *Modelica* model. The *Modelica* model contains the physics of the *RTC-Tools 2* model.
- In the `output` folder the simulation output is saved in the file `timeseries_export.csv`.
- `src` contains a *Python* file. This file contains the configuration of the model and can be used to run the model.

5.2 The physical part: *Modelica*

Load the Deltares library `<installation directory>\RTCTools2\mo\Deltares\package.mo` via the menu `File→Open Model/Library File(s)`.

The Deltares library contains a package with pre-defined water related models. Figure 5.1 shows the *OMEdit* with the Deltares library loaded. The lookup table reservoir model has been selected in the `text view` mode.

Load the tutorial model `<installation directory>\RTCTools2\tutorial\model\Tutorial.mo` and open it in diagram view. The `Libraries Browser` and the mouse-over feature help to identify the pre-defined models from the Deltares library (Figure 5.2).

The model “tutorial” represents a simple water system with the following elements:

- a canal segment, modelled as storage element (`Deltares.Flow.OpenChannel.Storage.LookupTable`) with the waterlevel–storage relation given in Table 5.1. This relation is specified as input file in the folder `input`.
- a discharge boundary condition on the right side
- a water level boundary condition on the left side
- two hydraulic structures, both modeled as `Deltares.Flow.OpenChannel.Structures.Pump`:
  - a pump
  - an orifice

The model represents a typical setup for the dewatering of lowland areas. Water is routed from the hinterland (modeled as discharge boundary condition, right side) through a canal (modeled as storage element) towards the sea (modeled as water level boundary condition on the left side). Keeping the lowland area dry requires that enough water is discharged to the sea. If the sea water level is lower than the water level in the canal, the water can be discharged to the sea via gradient flow through the orifice (or a weir). If the sea water level is higher than in the canal, water must be pumped.

To discharge water via gradient flow is cheap, pumping costs money. The control task is to keep the water level in the canal below a given flood warning level at minimum costs. The expected result is that the model computes a control pattern that makes use of gradient flow

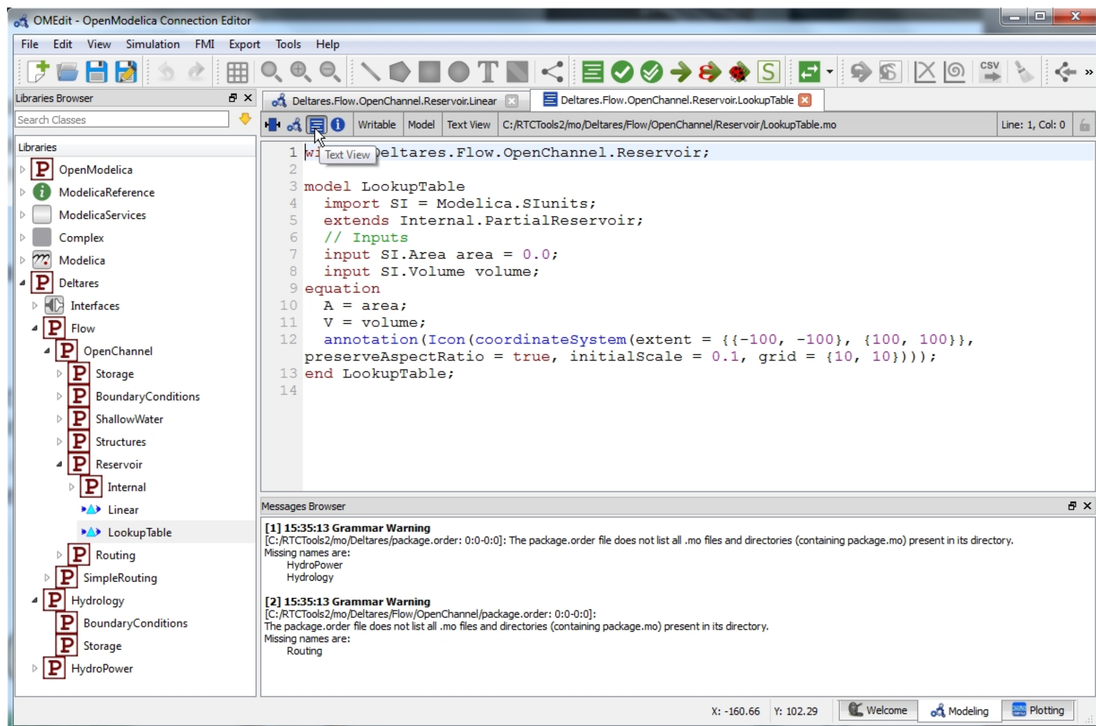

Figure 5.1: *Deltares library loaded, the lookup table reservoir model selected in text view*

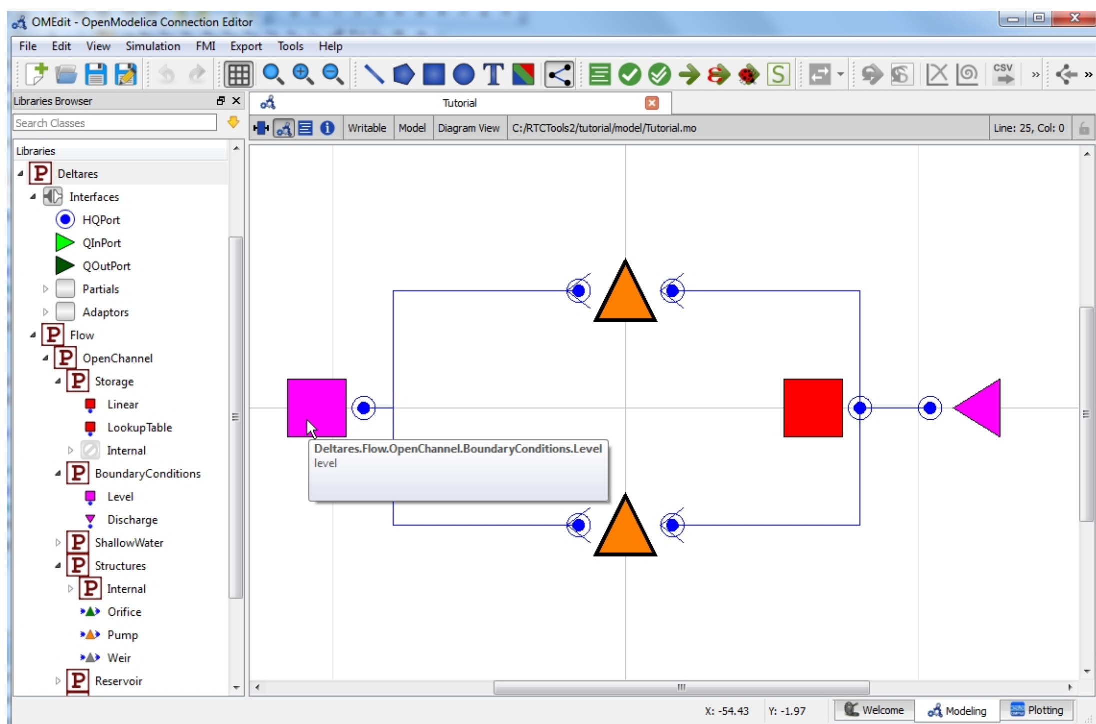

Figure 5.2: *Model “tutorial” in OMEdit diagram view.* `Libraries Browser` *is open and mouse-over is active on a* `Boundary Conditions` *object.*

Table 5.1: *Waterlevel-storage relation*

Volume	Water level
0	-0.75
0	-0.5
0	-0.25
0	0
100000	0.25
500000	0.5
1000000	0.75

Figure 5.3: *Sluice and pump complex "Eefde" in the Twenthe Canal near the IJssel river, the Netherlands (source: https://nl.wikipedia.org/wiki/Sluis_Eefde)*

whenever possible and activates the pump only when necessary.

In text mode the *Modelica* model looks as follows (annotation statements have been cleaned up):

Listing 5.1: *Modelica model* `Tutorial.mo`

```
model Tutorial
  Deltares.Flow.OpenChannel.Storage.LookupTable storage;
  Deltares.Flow.OpenChannel.BoundaryConditions.Discharge
     discharge;
  Deltares.Flow.OpenChannel.BoundaryConditions.Level level;
  Deltares.Flow.OpenChannel.Structures.Pump pump;
  Deltares.Flow.OpenChannel.Structures.Pump orifice;
  input Modelica.SIunits.Volume V_storage;
  input Modelica.SIunits.VolumeFlowRate Q_in;
  input Modelica.SIunits.Position H_sea;
  input Modelica.SIunits.VolumeFlowRate Q_pump;
  input Modelica.SIunits.VolumeFlowRate Q_orifice;
equation
  connect(orifice.HQDown, level.HQ);
  connect(storage.HQ, orifice.HQUp);
  connect(storage.HQ, pump.HQUp);
  connect(discharge.HQ, storage.HQ);
  connect(pump.HQDown, level.HQ);
  storage.volume = V_storage;
  discharge.Q = Q_in;
  level.H = H_sea;
  pump.Q = Q_pump;
  orifice.Q = Q_orifice;
end Tutorial;
```

The five water system elements

- storage with lookup table (i. e., a level-storage relation)
- discharge boundary condition
- water level boundary condition
- pump
- orifice

appear under the `model` statement. The input variables

- `V_storage`
- `Q_in`
- `H_sea`
- `Q_pump`
- `Q_orifice`

are assigned to the water system elements.

The `equation` part connects these five elements with the help of connections.

Note that `Pump` extends the partial model `HQTwoPort` which inherits from the connector `HQPort`. With `HQTwoPort`, `Pump` can be connected on two sides. `level` represents a model boundary condition (model is meant in a hydraulical sense here), so it can be connected to one other

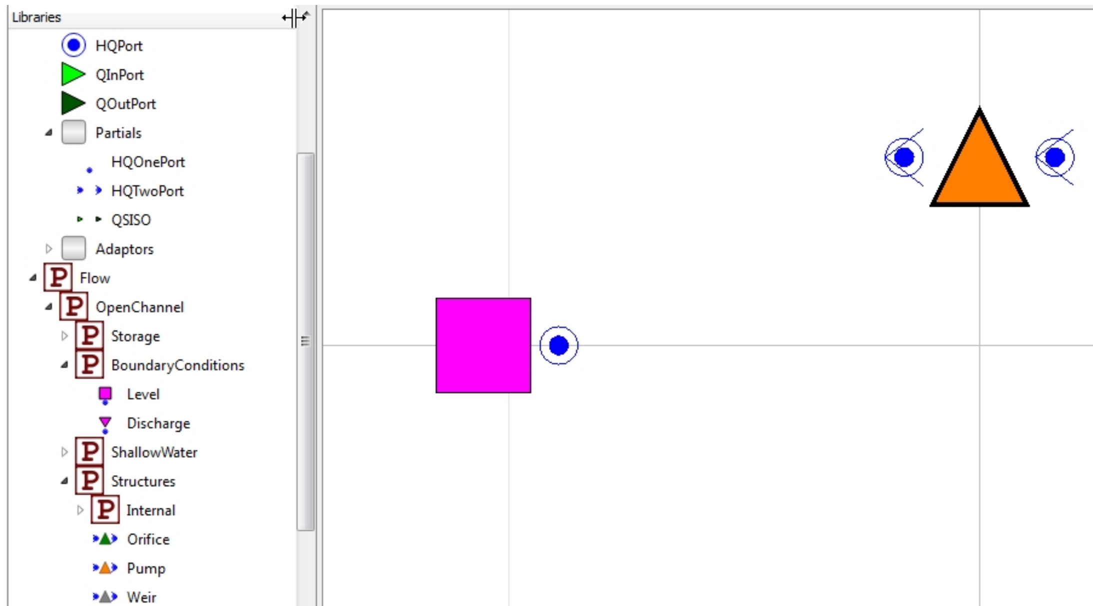

Figure 5.4: *Map symbols are shown for interfaces, here shown for the level boundary condition model and the pump model from the "tutorial" example, connections are hidden.*

element only. It extends the `HQOnePort` which again inherits from the connector `HQPort`. For the interfaces `HQTwoPort` and `HQOnePort` map symbols are shown in the diagram view in *OMEdit*. These map symbols are loosely connected to the map symbols of the feature they belong to (Fig. 5.4).

5.3 The *Python* master script for the optimization problem

5.3.1 Introduction

Listing 5.2: *Python master script* `tutorial.py`

```
1 from rtctools.optimization.
      hybrid_shooting_optimization_problem import
      HybridShootingOptimizationProblem
  from rtctools.optimization.modelica_mixin import
      ModelicaMixin
3 from rtctools.optimization.csv_mixin import CSVMixin
  from rtctools.optimization.csv_lookup_table_mixin import
      CSVLookupTableMixin
5 from rtctools.util import run_optimization_problem

7 class Tutorial(CSVLookupTableMixin, CSVMixin, ModelicaMixin
      , HybridShootingOptimizationProblem):
      def __init__(self, model_folder, input_folder,
          output_folder):
9           # Call constructors
            CSVLookupTableMixin.__init__(self,
11                                         input_folder=
                                               input_folder)
            CSVMixin.__init__(self,
13                               input_folder=input_folder,
                                output_folder=output_folder)
```

```
15          ModelicaMixin.__init__(self,
                                   model_name='Tutorial',
17                                 model_folder=model_folder,
                                   control_inputs=['Q_pump', '
                                       Q_orifice'],
19                                 lookup_tables=['V_storage'])
            HybridShootingOptimizationProblem.__init__(self)
21
        def objective(self):
23          # Minimize water pumped
            return self.integral('Q_pump')
25
        def constraints(self):
27          constraints = []
            # Pump uphill only
29          for q, h_up, h_down in zip(self.states_in('Q_pump')
                , self.states_in('storage.HQ.H'), self.states_in
                ('level.H')):
                constraints.append((q * (h_down - h_up), 0.0, 1
                    e10))
31          # Release through orifice downhill only
            for q, h_up, h_down in zip(self.states_in('
                Q_orifice'), self.states_in('storage.HQ.H'),
                self.states_in('level.H')):
33              constraints.append((q * (h_up - h_down), 0.0, 1
                    e10))
            return constraints
35
        def bounds(self):
37          # Bound variables
            return {'Q_pump': (0.0, 10.0),
39                  'Q_orifice': (0.0, 10.0),
                    'storage.HQ.H': (0.0, 0.5)}
41
    # Run
43  run_optimization_problem(Tutorial, base_folder='..')
```

Load the script `<installation directory>\RTCTools2\tutorial\src\tutorial.py` into a *Python* editor. This *Python* script connects the components of *RTC-Tools 2* (Chapter 1, Figure 1.1). We briefly introduce the most important parts of the script, here displayed in Listing 5.2.

The basic idea is that the *RTC-Tools 2* modeler modifies this script according to his individual needs. This is more configuration work in *Python* than programming in *Python*.

The script consists of the following blocks:

1 import of *Python* packages
2 definition of the class

- ◇ constructor
    - definition of the optimization problem:
    - objective function
    - definition of constraints

    - definition of varieable bounds

3 a run statement

5.3.2 Import of packages

The first block imports the necessary *RTC-Tools 2* classes:

- `HybridShoothingOptimizationProblem` time-discretizes the continuous physical model (in the current case represented with *Modelica*) and prepares the optimization problem for the optimizer.
- `ModelicaMixin` reads the model from the *Modelica* input file (`*.mo`) and lets *JModellica* put out the model to *Python*.
- `CSV(LookupTable)Mixin` is the interface to CSV files for input and output in time series, initial states, and lookup tables.
- `run_optimization_problem` runs the optimization problem (the simulation).

These classes are available as pre-compiled *Python* bytecode. The files are located in the folder `<installation directoru>\system\python27\Lib\site-packages\rtctools\optimization*.pyd`.

5.3.3 Class declaration and initialization

After the import block follows the class definition. In the current example, the class name is “Tutorial”, but it can be a different name, too. The class arguments refer to the classes being inherited from (see above). The class has the following methods:

- the constructor `__init__`
- the method `objective` for the objective function of the optimization problem
- the method `constraints` and
- the method `bounds` for the variable bounds.

5.3.4 The constructor

The constructor initializes the imported classes with the following arguments:

- `model folder`
- `input folder`
- `output folder`

The arguments `input folder` and `output folder` are used to feed the first two constructor methods `CSVLookupTableMixin` and `CSVMixin` with the information where to find the input files and the output files.

The constructor method `ModelicaMixin` connects to the *Modelica* model from Section 5.2:

```
ModelicaMixin.__init__(self,
                        model_name='Tutorial',
                        model_folder=model_folder,
                        control_inputs=['Q_pump', '
                           Q_orifice'],
                        lookup_tables=['V_storage'])
```

Variables from the *Modelica* model can be used for computations within the class “Tutorial”.

The model_name 'Tutorial' refers to the name of the *Modelica* model to use as root model for the simulation. model_folder refers to the folder where the *Modelica* model can be found.

The control_inputs are Q_pump and Q_orifice. These two variables show up in the *Modelica* model 'Tutorial' as two of five input variables (see Listing 5.1). Note that *Modelica* does not distinguish between control variables and other variables, so the user is responsible for keeping track of variables how are used.

Lookup tables from CSV files are transferred to *Modelica* also via the master script and the ModelicaMixin method. Here the lookup table data from input file V_storage.csv is fed towards input variable V_storage.

5.3.5 The objective function

The method objective defines the objective function, also referred to as cost function. Within the objective function targets and soft constraints are specified. The optimization aims to minimize this objective function. In the current example the objective function has one term. The goal is to minimize the total pump volume over all time steps. integral is a method from the *RTC-Tools 2 Python* libraries and accumulates values of a variable – in the current case the total discharge Q_pump – over the simulation time. Q_pump is provided by *Modelica* under the given input data and control variables.

5.3.6 Constraints and variable bounds

Most optimization problems are subject to mathematical constraints. In *RTC-Tools 2* the constraints of the optimization problem are specified in the methods constraints and bounds.

For the current case there are two constraints defined in the method constraints. The first constraint is that pump flow must only allow upstream discharge, i. e.: pump discharge times the head difference at the pump (q * (h_down - h_up)) must be larger than zero. The second constraint is that discharge through the orifice discharge is allowed only in downstream direction. The constraints.append method is part of the *RTC-Tools 2* library. Note that the constraints use the variables 'Q_pump', 'Q_orifice', 'storage.HQ.H' and 'level.H'. These four variables are part of the *Modelica* model "Tutorial".

While constraints are mathematical expressions, bounds define a feasible range for variables. Variable bounds are specified in the method bounds. Often these variable bounds specify physical limits like the minimum and maximum water level of a reservoir, pump capacities or river bed levels. In the current example the pump capacity and the discharge capacity of the orifice have been set to a range between 0 and 10, and the water level in the canal segment must be between 0 and 0.5. Note that the range for the water level does not cover the whole range of the waterlevel-storage relation given in Table 5.1.

Note that constraints limit the solution space. If no optimal solution can be found within the solution space, the optimizer returns an infeasible solution. This means, that hard constraints can not be violated. Constraints specified in the constraints and bounds method are also referred to as hard constraints.

5.3.7 Run the optimization problem

The last command executes the optimization with the help of the command run_optimization_problem. This command is part of an import library, and an object instance of the "Tutorial" class is created within this command.

5.4 Run the *Python* master script

To run the *Python* master script search for `Tutorial` in the *Windows* start menu and run it. The command behind this start menu entry is `<installationdirectory>\RTCTools2\system\JModelica\RTC2_Python.batC:\RTCTools2\tutorial\src\tutorial.py`

You can check this in the properties dialogue (right mouse button → "Properties". This shortcut ensures two things: 1), that the *Python* distribution that was installed as part of the *RTC-Tools 2* package is used (i. e., `<installationdirectory>\RTCTools2\system\python27\python.exe`), and 2), that this *Python* distribution is used within a properly set-up environment (as defined in `<installationdirectory>\RTCTools2\system\JModelica\RTC2_setenv.bat`). A terminal window opens, and the first lines should look like

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

NOTE: You are using Ipopt by default with the MUMPS linear solver.
      Other linear solvers might be more efficient (see Ipopt documentation).


This is Ipopt version 3.10.3, running with linear solver mumps.

Number of nonzeros in equality constraint Jacobian...:      359
Number of nonzeros in inequality constraint Jacobian.:      120
Number of nonzeros in Lagrangian Hessian.............:      100

Total number of variables............................:      200
                     variables with only lower bounds:        0
                variables with lower and upper bounds:       60
                     variables with only upper bounds:        0
Total number of equality constraints.................:      160
Total number of inequality constraints...............:       42
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:       42
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls

   0 7.0199930e+002 3.02e+002 3.30e+001  -1.0 0.00e+000    -  0.00e+000 0.00e+000   0
```

The last lines should look like the following:

```
  64 7.9999765e+003 1.69e-009 7.28e-012  -8.6 2.52e-005    -  1.00e+000 1.00e+000h  1

Number of Iterations....: 64

                                   (scaled)                 (unscaled)
Objective...............:  2.2222156933433118e+002   7.9999764960359225e+003
Dual infeasibility......:  7.2759576141834259e-012   2.6193447411060333e-010
```

```
Constraint violation....:   1.6880221664905547e-013   1.6880221664905548e-009
Complementarity.........:   2.5084101706846311e-009   9.0302766144646727e-008
Overall NLP error.......:   8.2934654460433754e-010   9.0302766144646727e-008

Number of objective function evaluations             = 77
Number of objective gradient evaluations             = 65
Number of equality constraint evaluations            = 77
Number of inequality constraint evaluations          = 77
Number of equality constraint Jacobian evaluations   = 65
Number of inequality constraint Jacobian evaluations = 65
Number of Lagrangian Hessian evaluations             = 64
Total CPU secs in IPOPT (w/o function evaluations)   =      0.179
Total CPU secs in NLP function evaluations           =      0.031

EXIT: Optimal Solution Found.
time spent in eval_f: 0.004 s. (77 calls, 0.0519481 ms. average)
time spent in eval_grad_f: 0.008 s. (66 calls, 0.121212 ms. average)
time spent in eval_g: 0.001 s. (77 calls, 0.012987 ms. average)
time spent in eval_jac_g: 0.006 s. (67 calls, 0.0895522 ms. average)
time spent in eval_h: 0.01 s. (65 calls, 0.153846 ms. average)
time spent in main loop: 0.216 s.
time spent in callback function: 0 s.
time spent in callback preparation: 0 s.
2016-06-14 10:59:38,548 INFO Solver succeeded with status Solve_Succeeded
2016-06-14 10:59:38,548 INFO Extracting results
2016-06-14 10:59:38,552 INFO Done extracting results
2016-06-14 10:59:38,558 INFO Done with optimize()
>>>
```

The message “Optimal Solution Found” is a log message from the optimizer *IPOPT* that indicates that the simulation has terminated successfully. The objective function value has developed within 64 iteration steps from $7.0199930\text{E}+002$ to $7.9999765\text{E}+003$. In our case the objective function value is the total pumped volume. The fact that the objective function is larger than zero means that the optimizer was not able to operate the water system without pumping. For more information about the *IPOPT* convergence info see **?**, Appendix F.

5.5 Input data and results

Figure 5.5 shows simulation results from the file `<installation directory>\RTCTools2\tutorial\output\timeseries_export.csv`. This file can be opened with *Microsoft Excel* or *Veusz*.

In the top diagram the sea level (downstream boundary condition, `level.H`) and the water level in the canal segment (which is the water level that corresponds to the discharge boundary condition of 5 m^3/s `discharge.HQ.H`) are shown. The bottom diagram shows the pump discharge and the discharge of the orifice. At times where the water level in the canal segment is higher than the sea level the water is discharged via the orifice. A significant pump usage is not applied before the water level in the canal reaches the maximum value (hard constraint) at 2013-05-20 12:00 hours. A very small pump discharge is applied before this point in time, but the optimizer chooses to have the bulk of pumping activity when the canal water level is high and the sea level is low, because the head difference is small then, which leads to a small objective function value.

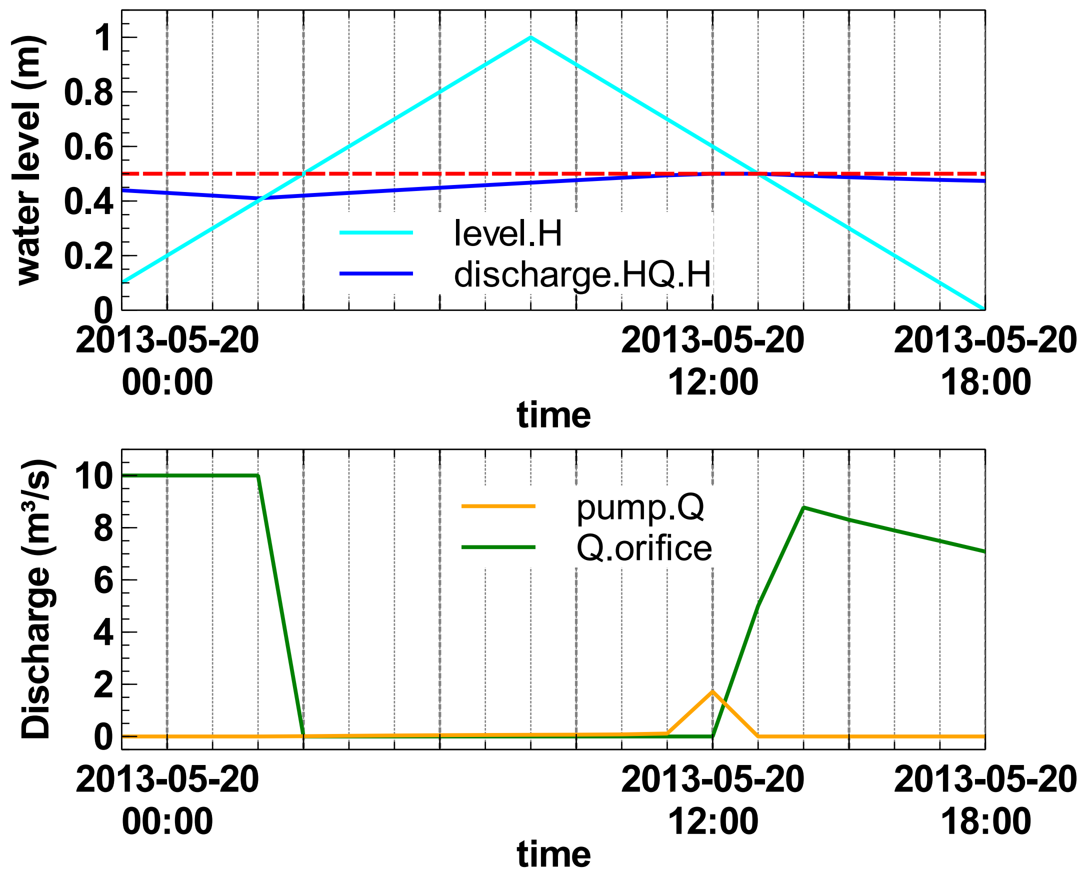

Figure 5.5: *Water levels and discharges over time*

6 References

Andersson, J., 2013. *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium.

Andersson, J., J. Åkesson and M. Diehl. "CasADi: A Symbolic Package for Automatic Differentiation and Optimal Control." In S. Forth, P. Hovland, E. Phipps, J. Utke and A. Walther, eds., *Recent Advances in Algorithmic Differentiation*, vol. 87, pages 297–307. Springer Berlin Heidelberg. URL http://link.springer.com/10.1007/978-3-642-30023-3_27.

Schwanenberg, D. and B. Becker, 2016. *Software tools for modelling Real-time control / RTC-Tools*. Technical Reference Manual, Deltares, Delft.