RTC-Tools Hydraulic Structures Documentation

Release 2.0.0a12

Tjerk Vreeken, Klaudia Horvath, et al.

Sep 18, 2020

User Documentation

CHAPTER 1

Contents:

1.1 Getting Started

1.1.1 Installation

Installation of the RTC-Tools Hydraulic Structures library is as simple as:

```
# 1. Use pip and PyPI to install
pip install rtc-tools-hydraulic-structures
```

RTC-Tools Hydraulic Structures depends on RTC-Tools and its ChannelFlow library, which are automatically installed as dependencies.

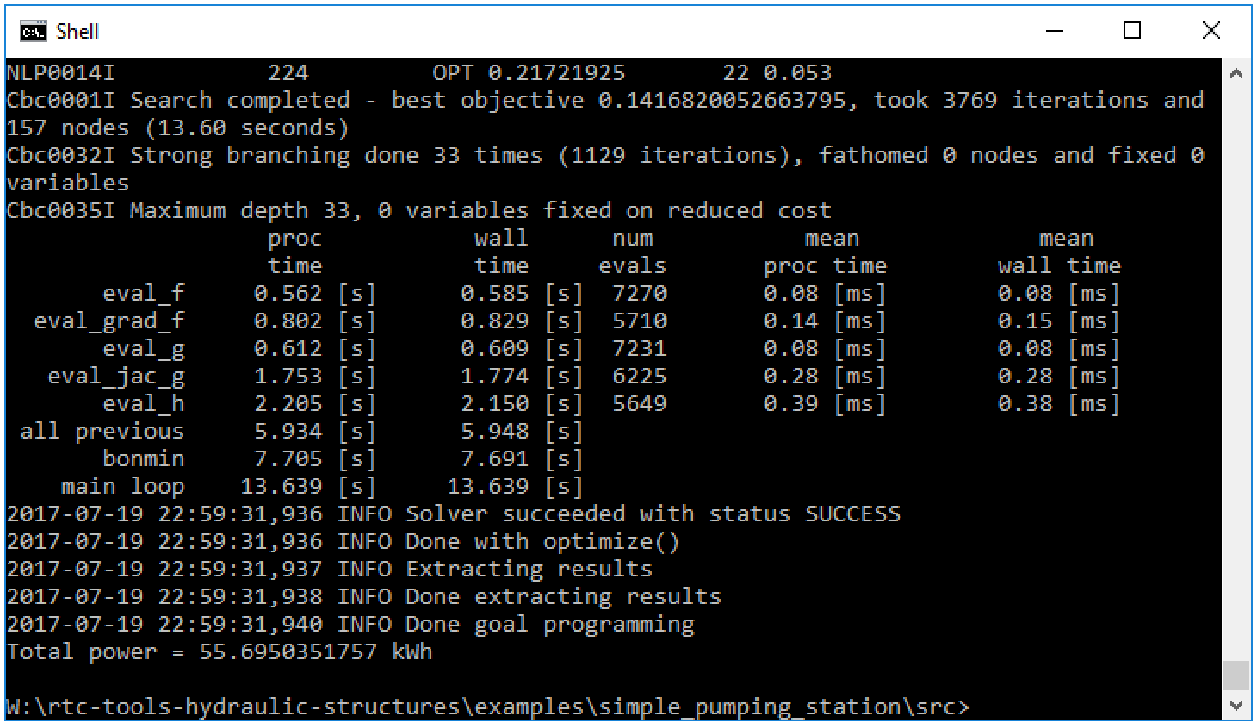
The Modelica library is installed in a hard to access location to make sure RTC-Tools can find it. If you want to load the library in an editor like OpenModelica, it is best to run *rtc-tools-copy-libraries*. See also the RTC-Tools documentation on this.

1.1.2 Running an example

To make sure that everything is set-up correctly, you can run one of the example cases. These do not come with the installation, and need to be downloaded separately:

```
# 1. Clone the repository
git clone https://gitlab.com/deltares/rtc-tools-hydraulic-structures.git

# 2. Change directory to the example folder
cd rtc-tools-hydraulic-structures/examples/pumping_station/basic/src

# 3. Run the example
python example.py
```

You will see the progress of RTC-Tools in your shell. If all is well, you should see something like the following output.



```
NLP0014I          224         OPT 0.21721925       22 0.053
Cbc0001I Search completed - best objective 0.1416820052663795, took 3769 iterations and
157 nodes (13.60 seconds)
Cbc0032I Strong branching done 33 times (1129 iterations), fathomed 0 nodes and fixed 0
variables
Cbc0035I Maximum depth 33, 0 variables fixed on reduced cost
                   proc           wall      num          mean             mean
                   time           time     evals       proc time        wall time
       eval_f     0.562 [s]      0.585 [s]  7270       0.08 [ms]        0.08 [ms]
  eval_grad_f     0.802 [s]      0.829 [s]  5710       0.14 [ms]        0.15 [ms]
       eval_g     0.612 [s]      0.609 [s]  7231       0.08 [ms]        0.08 [ms]
   eval_jac_g     1.753 [s]      1.774 [s]  6225       0.28 [ms]        0.28 [ms]
       eval_h     2.205 [s]      2.150 [s]  5649       0.39 [ms]        0.38 [ms]
 all previous     5.934 [s]      5.948 [s]
       bonmin     7.705 [s]      7.691 [s]
    main loop    13.639 [s]     13.639 [s]
2017-07-19 22:59:31,936 INFO Solver succeeded with status SUCCESS
2017-07-19 22:59:31,936 INFO Done with optimize()
2017-07-19 22:59:31,937 INFO Extracting results
2017-07-19 22:59:31,938 INFO Done extracting results
2017-07-19 22:59:31,940 INFO Done goal programming
Total power = 55.6950351757 kWh

W:\rtc-tools-hydraulic-structures\examples\simple_pumping_station\src>
```

1.1.3 Contribute

You can contribute to this code through Pull Request on GitLab. Please, make sure that your code is coming with unit tests to ensure full coverage and continuous integration in the API.

1.2 Support

Raise any issue on GitLab such that we can address your problem.

1.3 Python API

1.3.1 Pumping Station Mixin

class `rtctools_hydraulic_structures.pumping_station_mixin.`**`PumpingStationMixin`**(**args*, ***kwargs*)

Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds handling of PumpingStation objects in your model to your optimization problem.

Relevant parameters and variables are read from the model, and from this data a set of constraints and objectives are automatically generated to minimize cost.

If historical data regarding the status of the pumps is provided, this information is used to ensure that the minimum amount of time a pump must be on / off is respected.

`pumping_stations`() → List[rtctools_hydraulic_structures.pumping_station_mixin.PumpingStation]

User problem returns list of *`PumpingStation`* objects.

Returns A list of pumping stations.

pumpingstation_cache_hq_subproblem = True
Use pickle to cache the HQ subproblems that are solved.

pumpingstation_energy_price_symbol = 'energy_price'
Energy price symbol to use if no symbol specified per pumping station or per pump.

class `rtctools_hydraulic_structures.pumping_station_mixin.`**Pump**(*optimization_problem*, *symbol*, *energy_price_symbol*, *semi_continuous=None*, *status_history=None*)

Bases: `rtctools_hydraulic_structures.util._ObjectParameterWrapper`

Python Pump object as an interface to the `Pump` object in the model.

__init__(*optimization_problem*, *symbol*, *energy_price_symbol*, *semi_continuous=None*, *status_history=None*)
Initialize self. See help(type(self)) for accurate signature.

discharge()
Get the state corresponding to the pump discharge.

Returns *MX* expression of the pump discharge.

head()
Get the state corresponding to the pump head. This depends on the `head_option` that was specified by the user.

Returns *MX* expression of the pump head.

class `rtctools_hydraulic_structures.pumping_station_mixin.`**Resistance**(*optimization_problem*, *symbol*)

Bases: `rtctools_hydraulic_structures.util._ObjectParameterWrapper`

Python Resistance object as an interface to the `Resistance` object in the model.

__init__(*optimization_problem*, *symbol*)
Initialize self. See help(type(self)) for accurate signature.

discharge()
Get the state corresponding to the discharge through the resistance.

Returns *MX* expression of the discharge.

head_loss()
Get the state corresponding to the head loss over the resistance.

Returns *MX* expression of the head loss.

class `rtctools_hydraulic_structures.pumping_station_mixin.`**`PumpingStation`**(*optimization_problem: rtc-tools.optimization.optimiz sym-bol: str*, *pump_symbols: List[str] = None*, *en-ergy_price_symbols: Union[str, List[str]] = None*, *semi_continuous: Union[str, List[str]] = None*, *sta-tus_history: Union[str, List[str]] = None*, ***kwargs*)

Bases: `rtctools_hydraulic_structures.util._ObjectParameterWrapper`

Python PumpingStation object as an interface to the *`PumpingStation`* object in the model.

`__init__`(*optimization_problem: rtctools.optimization.optimization_problem.OptimizationProblem*, *symbol: str*, *pump_symbols: List[str] = None*, *energy_price_symbols: Union[str, List[str]] = None*, *semi_continuous: Union[str, List[str]] = None*, *status_history: Union[str, List[str]] = None*, ***kwargs*)

Initialize the pumping station object.

Parameters

- **`optimization_problem`** – `OptimizationProblem` instance.
- **`symbol`** – Symbol name of the pumping station in the model.
- **`pump_symbols`** – Symbol names of the pumps in the pumping station.
- **`energy_price_symbols`** – String or list of names of the energy price's time series of the pumps in the pumping station.
- **`semi_continuous`** – String or list of names of the constant input indicating use of the semi-continuous approach
- **`status_history`** – String or list of names of the pump status history time series. If string, one can use e.g. "{pump}_status_hist" to map to "pumpingstation1.pump1_status_hist", with "pumpingstation1.pump1" the pump symbol.

`pumps`() → List[rtctools_hydraulic_structures.pumping_station_mixin.Pump]

Get a list of *`Pump`* objects that are part of this pumping station in the model.

Returns List of *`Pump`* objects.

resistances() → List[rtctools_hydraulic_structures.pumping_station_mixin.Resistance]
Get a list of *Resistance* objects that are part of this pumping station in the model.

Returns List of *Resistance* objects.

class rtctools_hydraulic_structures.pumping_station_mixin.**MinimizePumpCostGoal**(*use_dynamic_no bool = True, ex- clude_continuou bool = False, *args, **kwargs*)

Bases: rtctools_hydraulic_structures.pumping_station_mixin._MinimizePumpGoal

Goal that minimizes overall energy costs.

Loops over all pumps in all pumping stations, integrating all instantaneous pump operating costs (and any start-up and shut-down costs/energy) in the optimization window into one objective value.

Variables

- **function_nominal** – Nominal value of needed for scaling. Guessed automatically based on the power range of all pumps.
- **priority** – Priority of this goal. Default is sys.maxsize.

class rtctools_hydraulic_structures.pumping_station_mixin.**MinimizePumpEnergyGoal**(*use_dynamic_ bool = True, ex- clude_continu bool = False, *args, **kwargs*)

Bases: rtctools_hydraulic_structures.pumping_station_mixin._MinimizePumpGoal

Goal that minimizes overall energy consumption.

Loops over all pumps in all pumping stations, integrating all instantaneous pump powers (and any start-up and shut-down energy) in the optimization window into one objective value.

Variables

- **function_nominal** – Nominal value of needed for scaling. Guessed automatically based on the power range of all pumps.
- **priority** – Priority of this goal. Default is sys.maxsize.

rtctools_hydraulic_structures.pumping_station_mixin.**plot_operating_points**(*optimization_problem, out- put_folder=None, plot_expanded_working_*

Plot the working area of each pump with its operating points.

1.3.2 Weir Mixin

class `rtctools_hydraulic_structures.weir_mixin.`**`WeirMixin`**(**args*, ***kwargs*)

Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds handling of Weir objects in your model to your optimization problem.

`weirs``()` → List[rtctools_hydraulic_structures.weir_mixin.Weir]

User problem returns list of `Weir` objects.

Returns A list of weirs.

class `rtctools_hydraulic_structures.weir_mixin.`**`Weir`**(*optimization_problem*, *name*)

Bases: `rtctools_hydraulic_structures.util._ObjectParameterWrapper`

Python Weir object as an interface to the Weir object in the model.

In the optimization, the weir flow is implemented as constraints. It means that the optimization calculated a flow (not weir height!), that is forced by the constraints to be a physically possible weir flow.

`discharge``()`

Get the state corresponding to the weir discharge.

Returns *MX* expression of the weir discharge.

1.4 Modelica API

1.4.1 Pumping Station

The Modelica library `Deltares.HydraulicStructures.PumpingStation` is an extension to the `Deltares.ChannelFlow.Hydraulic` library, which is part of the ChannelFlow library. It consists of the following components:

Pump A pump model with a QHP (discharge, head, power) relationship, to be used for optimization of e.g. costs. It extends `Deltares.ChannelFlow.Hydraulic.Structures.Pump`

Resistance Quadratic resistance.

PumpingStation Encapsulating class for Pump and Resistance objects.

Note: `Pump` and `Resistance` objects should always be placed inside a `PumpingStation` object.

Pump

class **`Pump`** : Deltares::ChannelFlow::Hydraulic::Structures::Pump

Represents a single pump object. Because the power of the pump is seldom a linear function of Q and H, this class is wrapped by the Python API's `Pump` which turns the power equation specified by `power_coefficients` into a set of inequality constraints:

$$P \geq P_{min} \cdot S$$

$$P \leq P_{max} \cdot S$$

$$P \geq C_{1,1} + C_{1,2}Q + \ldots - P_{max} \cdot (1 - S)$$

where S is the pump status (off = 0, on = 1), and P_{min} and P_{max} are the minimum and maximum possible power when the pump is on.

With minimization of pumping costs (i.e. power), the optimization results will satisfy the first inequality constraint as if it was an equality constraint.

`Real power_coefficients[_, _, _]`

The power coefficients describe the relationship between the discharge, head and power. For example, one can consider a fit of the pump power of the general form:

$$P = C_{1,1} + C_{1,2}Q + C_{2,1}H + C_{2,2}QH + C_{1,3}Q^2 + \dots$$

The power coefficients matrix corresponds to the coefficients *C* in the equation above. To guarantee that optimization finds a good and stable solution, we require the coefficients of this polynomial to be chosen such that the polynomial is convex over the entire domain.

When several power coefficients matrices are given, the pump power will be greater than or equal to all resulting equations. In other words, the pump power will take the maximum value of all equations. The specification of multiple power coefficients can, for example, be used to define a fully linear power relation that can be handled by MILP solvers.

Note: Strictly speaking it would only have to be convex over the (automatically) extended working area domain, the size of which is not always known before run-time.

`Real working_area[_, _, _]`

The working area array describes the polynomials bounding the convex set of allowed Q-H coordinates. These polynomials typically arise from one of the following properties:

- Q-H curve at minimum pump speed
- Q-H curve at maximum pump speed
- Minimum required efficiency (e.g. 50%)
- Minimum and/or maximum input power constraint
- Cavitation constraints
- NPSH constraints

The first coordinate of the array is the polynomial number. For example, `working_area[1, :, :]` would describe the first working area polynomial. The order of Q and H coefficients is the same as in `power_coefficients`.

Real **`working_area_direction`**[_]

The polynomials in `working_area` describe the polynomials, but do not yet indicate what side of this polynomial the Q-H combination has to be on. So for each of the polynomials in the working area we have to specify whether the expression should evaluate to a positive expression (=*1*), or a negative expression (=*-1*).

Note: It may become unnecessary to specify this in the future, if it is possible to figure out a way to determine this automatically based on the polynomials and their crossing points.

Integer **`head_option`** = 0

What head to use for the pump head. This can be one of the following three options:

-1 The upstream head

0 The differential head (i.e. downstream head minus upstream head)

1 The downstream head.

Modelica::SIunits::Duration **minimum_on** = 0.0 * 3600

The minimum amount of time in seconds a pump needs to be on before allowed to switch off again. This applies to all pumps in this pumping station.

Note: Only multiples of the (equidistant) time step are allowed.

Modelica::SIunits::Duration **minimum_off** = 0.0 * 3600

The minimum amount of time in seconds a pump needs to be off before allowed to switch on again. This applies to all pumps in this pumping station.

Note: Only multiples of the (equidistant) time step are allowed.

Modelica::SIunits::Energy **start_up_energy** = 0.0

The energy needed to start a pump. This will be multiplied with the energy price associated to each pump to calculate the costs.

Real **start_up_cost** = 0.0

Costs in e.g. EUR or kg CO2 associated with a pump start up. Many pump switches could for example mean the pump life is shortened, or that more maintenance is needed. These could then be expressed in monetary value, and associated with pump start up.

Important: Make sure that the units of this value are of the same units as *start_up_energy* times the energy price.

Modelica::SIunits::Energy **shut_down_energy** = 0.0

Energy needed to shut down a pump. See equivalent parameter for pump start *start_up_energy* for more information.

Real **shut_down_cost** = 0.0

Cost associated with a pump shutdown. See equivalent parameter for pump start *start_up_cost* for more information.

Resistance

class Resistance : Deltares::ChannelFlow::Internal::HQTwoPort

Represents a single quadratic resistance object relating the head loss to the discharge:

$$dH = C \cdot Q^2$$

Because a non-linear equality constraint is not allowed in convex optimization, this class is wrapped by the Python API's *Resistance* which turns it into two inequality constraints:

$$dH \geq C \cdot Q^2$$

$$dH \leq \alpha \cdot Q$$

where the second constraint makes sure that dH is zero when Q is zero. The parameter α is automatically chosen such that all possible values of Q and their accompanying dH are included.

With minimization of pumping costs (i.e. power), the optimization results will satisfy the first inequality constraint as if it was an equality constraint, provided the power relationship of every pump is monotonically increasing with H.

Note: Only positive flow is allowed (read: enforced).

Real **C** = 0.0

PumpingStation

class PumpingStation : Deltares::ChannelFlow::Internal::HQTwoPort

Represents a pumping station object, containing one or more *Pump* or *Resistance* objects.

Integer **n_pumps**

The number of pumps contained in the pumping station. This is necessary to enforce the right size of e.g. the `pump_switching_matrix`.

Integer pump_switching_matrix[n_pumps, n_pumps] = -999

Together with `pump_switching_constraints` describes which pumps are allowed to be on at the same time. The default value of -999 will make Python fill it with the default matrix. This default matrix implies that the second pump can only be on when the first pump is on, that the third pump can only be on when the second pump is on, etc.

In matrix multiplication form

$$b[:,1] \leq A \cdot x \leq b[:,2]$$

with A the `pump_switching_matrix`, b the `pump_switching_constraints`, and x the vector of pump statuses:

$$x = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \end{bmatrix}$$

where S_1 is the status of pump 1 (on = *1*, off = *0*).

So the default matrix, where a pump being on requires all lower numbered pumps to be on as well, can be expressed as follows:

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 1 & -2 \end{bmatrix}$$

with `pump_switching_constraints` equal to:

$$b = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}$$

To allow all pumps to switch independently from each other, it is sufficient to set the coefficient matrix to all zeros (e.g. `pump_switching_matrix = fill(0, n_pumps, n_pumps)`). For rows in the matrix not containing any non- zero values, the accompanying constraints are not applied.

Note: Only square matrices allowed, i.e. a single constraint per pump.

Integer pump_switching_constraints[n_pumps, 2]

See discussion in `pump_switching_matrix`.

1.4.2 Weir

class Weir : Deltares::ChannelFlow::Internal::HQTwoPort

Represents a general movable-crest weir object described by the conventional weir equation (see e.g. Swamee, Prabhata K. "Generalized rectangular weir equations." Journal of Hydraulic Engineering 114.8 (1988): 945-949.):

$$Q = \frac{2}{3} C B \sqrt{2g} \left(H - H_w\right)^{1.5}$$

where Q is the discharge of the weir, C is the weir discharge coefficient (very well approximated by 0.61), B is the width of the weir, g is the acceleration of gravity, H is the water level, and H_w is the level of the movable weir crest. The equation assumes critical flow over the weir crest.

Modelica::SIunits::Length **width**

The physical width of the weir.

Modelica::SIunits::VolumeFlowRate **q_min**

The minimal possible discharge on this weir. It can be known from the physical characteristics of the system. The linear approximation works the best if this is set as tight as possible. It is allowed to set it to zero.

Modelica::SIunits::VolumeFlowRate **q_max**

The maximum physically possible flow over the weir. It should be set as tight as possible

Modelica::SIunits::Length **hw_min**

The minimal possible crest elevation.

Modelica::SIunits::Length **hw_max**

The maximum possible crest elevation.

Real **weir_coef** = 0.61

The discharge coefficient of the weir. Typically the default value of 0.61.

1.5 Examples

1.5.1 Pumping Station

Basic Pumping Station

Note: This example focuses on how to implement optimization for pumping stations in RTC-Tools using the Hydraulic Structures library. It assumes basic exposure to RTC-Tools. If you are a first-time user of RTC-Tools, please refer to the RTC-Tools documentation.

The purpose of this example is to understand the technical setup of a model with the Hydraulic Structures Pumping Station object, how to run the model, and how to interpret the results.

The scenario is the following: A pumping station with a single pump is trying to keep an upstream polder in an allowable water level range. Downstream of the pumping station is a sea with a (large) tidal range, but the sea level never drops below the polder level. Any pump must be on or off for at least a predetermined amount of time. The price on the energy market fluctuates, and the goal of the operator is to keep the polder water level in the allowable range while minimizing the pumping costs.

The folder `examples/pumping_station/basic` contains the complete RTC-Tools optimization problem.

The Model

For this example, the model represents a typical setup for a polder pumping station in lowland areas. The inflow from precipitation and seepage is modeled as a discharge (left side), with the total surface area / volume of storage in the polder modeled as a linear storage. The downstream water level is assumed to not be (directly) influenced by the pumping station, and therefore modeled as a boundary condition.

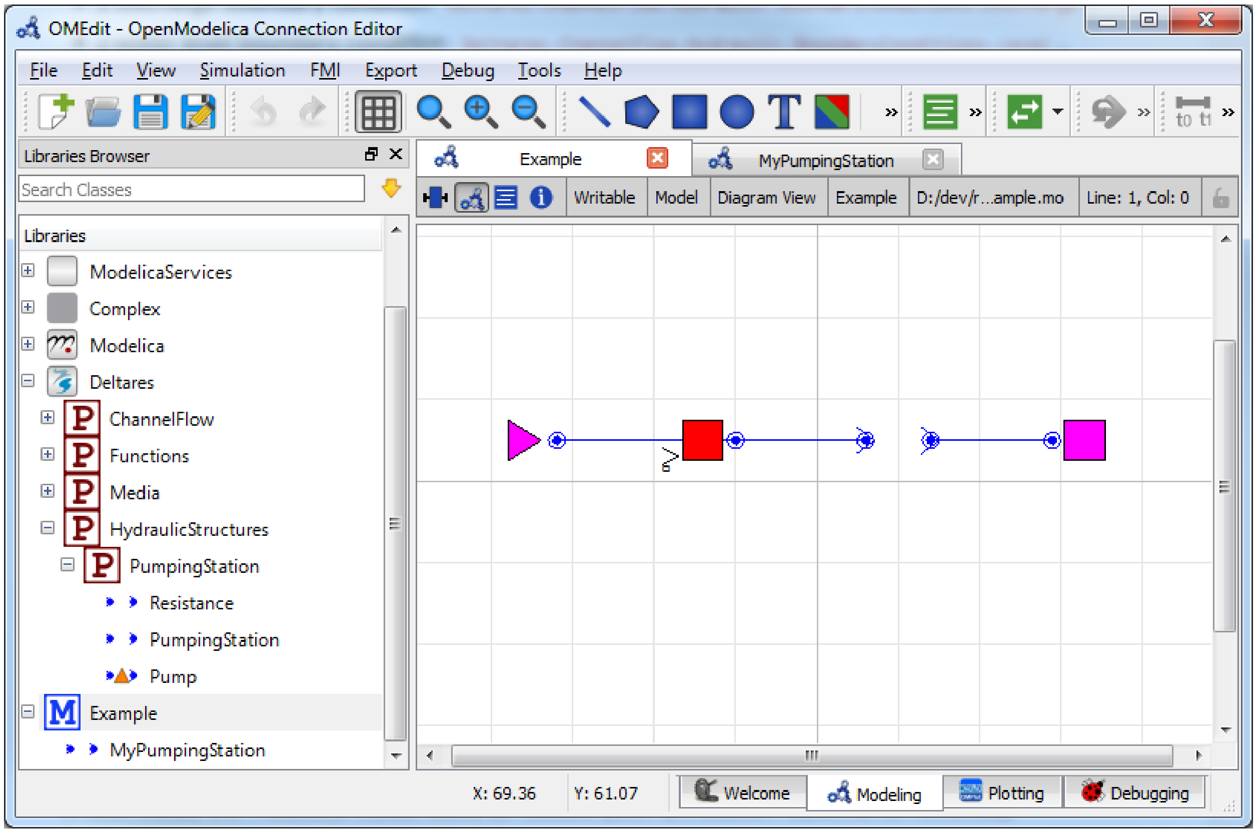
Operating the pumps to discharge the water in the polder consumes power, which varies based on the head difference and total flow. In general, the lower the head difference or discharge, the lower the power needed to pump water.

The expected result is therefore that the model computes a control pattern that makes use of these tidal and energy fluctuations, pumping water when the sea water level is low and/or energy is cheap. It is also expected that as little water as necessary is pumped, i.e. the storage available in the polder is used to its fullest. Concretely speaking this means that the water level at the last time step will be very close (or equal) to the maximum water level.

The model can be viewed and edited using the OpenModelica Connection Editor program. First load the Deltares library into OpenModelica Connection Editor, and then load the example model, located at `examples/`

`pumping_station/basic/model/Example.mo`. The model `Example.mo` represents a simple water system with the following elements:

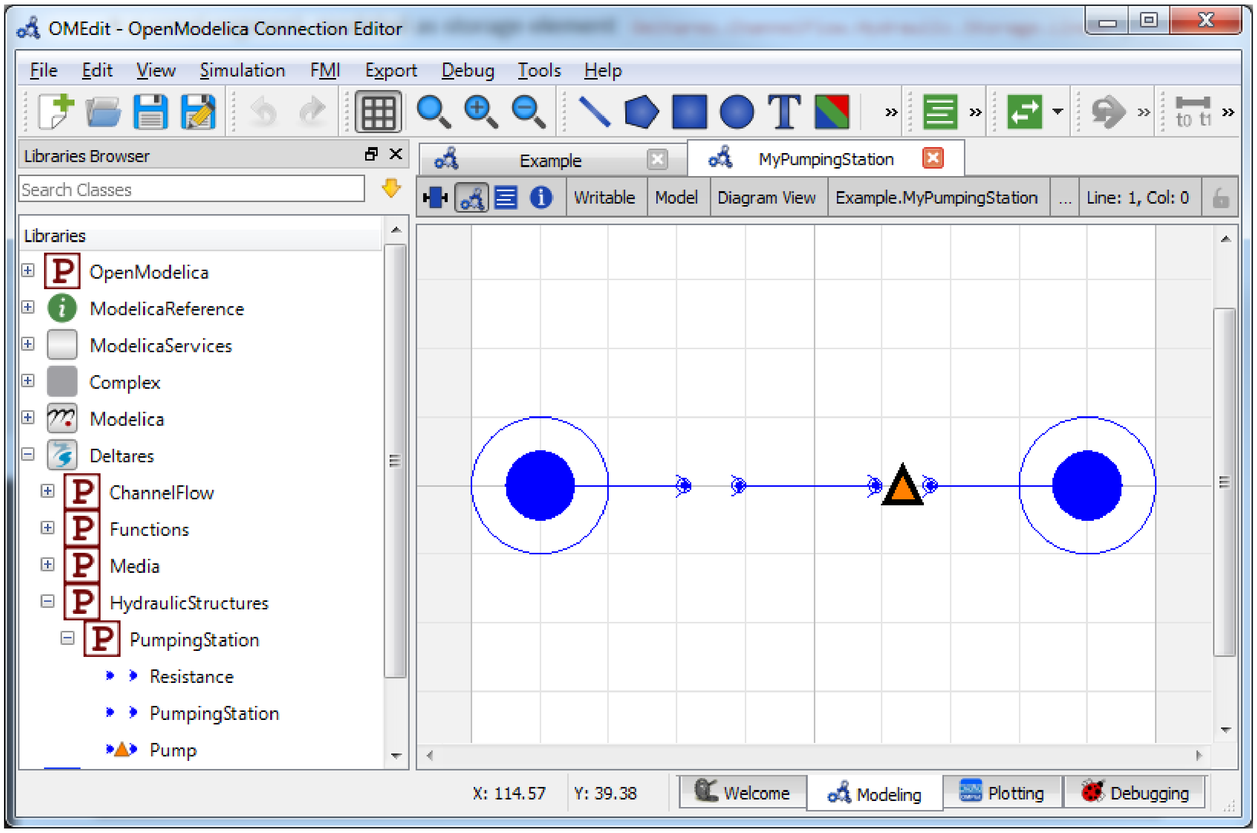
- the polder canals, modeled as storage element `Deltares.ChannelFlow.Hydraulic.Storage.Linear`,
- a discharge boundary condition `Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge`,
- a water level boundary condition `Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level`,
- a pumping station `MyPumpingStation` extending `Deltares.HydraulicStructures.PumpingStation.PumpingStation`



Note it is a nested model. In other words, we have defined our own `MyPumpingStation` model, which is in itself part of the `Example` model. You can add classes (e.g. models) to an existing model in the OpenModelica Editor by right clicking your current model (e.g. `Example`) –> `New Modelica Class`. Make sure to extend the `Deltares.HydraulicStructures.PumpingStation.PumpingStation` class.

If we navigate into our nested MyPumpingStation model, we have the following elements:

- our single pump `Deltares.HydraulicStructures.PumpingStation.Pump`,
- a resistance `Deltares.HydraulicStructures.PumpingStation.Resistance`,



In text mode, the Modelica model looks as follows (with annotation statements removed):

```
model Example

  model MyPumpingStation
    extends Deltares.HydraulicStructures.PumpingStation.PumpingStation(
      n_pumps=1
    );

    Deltares.HydraulicStructures.PumpingStation.Pump pump1(
      power_coefficients = {{{-39095.7484857 ,  66783.26438029},
                             {  7281.58399033,       0.0}},
                            {{-27859.72396609,  52991.89678362},
                             {  6547.78028277,       0.0}},
                            {{-23247.11759041,  47371.07046571},
                             {  6106.48287875,       0.0}},
                            {{-33451.06236192,  59110.131313   },
                             {  7034.0025483,       0.0}},
                            {{-39861.93265223,  67925.88638803},
                             {  7281.81109575,       0.0}},
                            {{-18905.97159263,  40725.7120928 },
                             {  5785.42670305,       0.0}},
                            {{-69080.26073112,  97844.92974535},
                             {  8776.27578528,       0.0}},
                            {{-27357.47031974,  52938.41395038},
                             {  6406.26137405,       0.0}},
                            {{-49670.43493511,  78653.65042992},
                             {  7840.68764271,       0.0}},
                            {{ -8104.87351317,  21084.47678188},
```

(continues on next page)

(continued from previous page)

```
                               {  4586.35220572,       0.0}},
                              {{-35797.98792283,  54481.08143225},
                               {  7595.20055527,       0.0}},
                              {{-60204.93398172,  89038.39027083},
                               {  8408.00958411,       0.0}},
                              {{-22131.50613895,  39713.73557364},
                               {  6465.17137718,       0.0}},
                              {{-14217.62875014,  34100.99360442},
                               {  5201.88819881,       0.0}},
                              {{-65348.50815507,  94475.38978919},
                               {  8575.77174818,       0.0}},
                              {{-39565.60866869,  63810.57380161},
                               {  7597.68369003,       0.0}},
                              {{-15114.73306987,  33754.82613809},
                               {  5508.58396756,       0.0}},
                              {{-51205.68182194,  79946.23472306},
                               {  7963.3995412,       0.0}},
                              {{ -6701.19277067,  20672.59101587},
                               {  4071.93143896,       0.0}},
                              {{-30637.27553712,  53847.70016923},
                               {  6976.92174728,       0.0}},
                              {{-23174.20780931,  45291.10739497},
                               {  6315.4012685,       0.0}},
                              {{-16838.11645224,  33397.62607586},
                               {  5890.34692945,       0.0}},
                              {{-48684.40184549,  73130.85845968},
                               {  8148.05117355,       0.0}},
                              {{ -5585.5596654 ,  17917.6377768 },
                               {  3903.50760056,       0.0}},
                              {{-11530.80023073,  25643.30751799},
                               {  5209.61580979,       0.0}},
                              {{-54050.93126723,  81474.03659672},
                               {  8244.19508265,       0.0}},
                              {{-17059.63308541,  38705.96069607},
                               {  5468.15043893,       0.0}},
                              {{-42990.53693314,  61384.80799185},
                               {  8077.30306983,       0.0}},
                              {{-28740.24310978,  47105.49859182},
                               {  7058.4428374,       0.0}},
                              {{ -9433.46984887,  26148.5409928 },
                               {  4488.53180172,       0.0}},
                              {{-54478.410763  ,  83635.76502236},
                               {  8076.34522102,       0.0}},
                              {{-34733.3023365 ,  61966.5148778 },
                               {  6963.68444901,       0.0}},
                              {{ -4371.73397616,  13785.26261371},
                               {  3759.68617973,       0.0}},
                              {{-44469.5263585 ,  71683.6367841 },
                               {  7713.90758475,       0.0}},
                              {{-10835.36867811,  27900.70743971},
                               {  4844.53856386,       0.0}}},

        working_area = {{{ -5.326999,  54.050758},
                         { -1.0,        0.0}},
                        {{-0.75242261,  2.79784166},
                         {-1.0,         0.0}},
                        {{-0.50567904,  2.3278547},
```

(continues on next page)

(continued from previous page)

```
                         {-1.0,          0.0}},
                        {{-1.31944138,  3.6515112},
                         {-1.0,          0.0}},
                        {{-1.93409137,  4.35798345},
                         {-1.0,          0.0}},
                        {{-1.04947144,  3.27458583},
                         {-1.0,          0.0}},
                        {{ 2.577975,   -5.203480},
                         {-1.0,          0.0}},
                        {{18.23055367, -15.4218745},
                         {-1.0,           0.0}},
                        {{16.79364838, -13.50826804},
                          {-1.0,          0.0}},
                        {{14.2340237,  -8.67903153},
                          {-1.0,          0.0}},
                        {{19.76140152, -17.16334892},
                          {-1.0,          0.0}},
                        {{15.40792185, -11.24444311},
                          {-1.0,          0.0}}},

      speed_coefficients = {{88.3224,  281.642,  143.011},
                             {58.8183,  -24.9845,   0.0},
                             {-1.45483,   0.0,      0.0}},

      working_area_direction = {1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1},

      minimum_on=3.0*3600
    );
    Deltares.HydraulicStructures.PumpingStation.Resistance resistance1(C=0.0);
  equation
    connect(HQUp, resistance1.HQUp);
    connect(resistance1.HQDown, pump1.HQUp);
    connect(pump1.HQDown, HQDown);
  end MyPumpingStation;

 // Elements in model flow chart
  Deltares.ChannelFlow.Hydraulic.Storage.Linear storage(
    A = 149000,
    H_b = -1.0,
    HQ.H(min = -0.7, max = 0.2),
    V(nominal = 1E5)
  );
  Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level sea;
  Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge inflow;
  MyPumpingStation pumpingstation1;

  // Input variables
  input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
  input Modelica.SIunits.Position H_ext(fixed=true);

  // Energy price is typically of units EUR/kWh (when optimizing for energy
  // usage), but one can also choose for e.g. ton CO2/kWh to get the lowest
  // CO2 output.
  input Real energy_price(fixed=true);

  // NOTE: Because we cannot flag each pump's .Q as "input", we need an extra
  // variable to do this. Format is expected to be the fully specified name,
```

(continues on next page)

(continued from previous page)

```
  // with all dots replaced with underscores.
  input Real pumpingstation1_pump1_Q;
  // TODO: Move bounds to the mixin.
  input Real pumpingstation1_resistance1_dH(min=0.0, max=10.0);

  // Output variables
  output Modelica.SIunits.Position storage_level;
  output Modelica.SIunits.Position sea_level;
equation
  connect(pumpingstation1.HQUp, storage.HQ);
  connect(pumpingstation1.HQDown, sea.HQ);
  connect(inflow.HQ, storage.HQ);
  // Mapping of variables
  inflow.Q = Q_in;
  sea.H = H_ext;
  pumpingstation1.pump1.Q = pumpingstation1_pump1_Q;
  pumpingstation1.resistance1.dH = pumpingstation1_resistance1_dH;
  storage_level = storage.HQ.H;
  sea_level = H_ext;
end Example;
```

The attributes of `pump1` are explained in detail in *Pump*.

In addition to the elements, two input variables `pumpingstation1_pump1_Q` and `pumpingstation1_resistance1_dH` are also defined, with a set of equations matching them to their dot-equivalent (e.g. `pumpingstation1.pump1.Q`).

Important: Because nested `input` symbols cannot be detected, it is necessary for the user to manually map this symbol to an equivalent one with dots replaced with underscores.

The Optimization Problem

The python script consists of the following blocks:

- Import of packages
- Definition of water level goal
- Definition of the optimization problem class
  - Constructor
  - Passing a list of pumping stations
  - Additional configuration of the solver
- A run statement

Importing Packages

For this example, the import block is as follows:

```
import os
from datetime import timedelta
```

(continues on next page)

(continued from previous page)

```python

from rtctools.optimization.collocated_integrated_optimization_problem import␣
↪CollocatedIntegratedOptimizationProblem
from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin,␣
↪StateGoal
from rtctools.optimization.linearized_order_goal_programming_mixin import␣
↪LinearizedOrderGoalProgrammingMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.optimization.pi_mixin import PIMixin
from rtctools.util import run_optimization_problem

```

Note that we are importing both `PumpingStationMixin` and `PumpingStation` from `rtctools_hydraulic_structures.pumping_station_mixin`.

Water Level Goal

Next we define our water level range goal. It reads the desired upper and lower water levels from the optimization problem class. For more information about how this goal maps to an objective and constraints, we refer to the documentation of `StateGoal`.

```python
class WaterLevelRangeGoal(StateGoal):
    """
    Goal that tries to keep the water level minum and maximum water level,
    the values of which are read from the optimization problem.
    """

    state = 'storage.HQ.H'

    priority = 1

    def __init__(self, optimization_problem):
        self.target_min = optimization_problem.wl_min
        self.target_max = optimization_problem.wl_max

        _range = self.target_max - self.target_min
        self.function_range = (self.target_min - _range, self.target_max + _range)

        super().__init__(optimization_problem)
```

Optimization Problem

Then we construct the optimization problem class by declaring it and inheriting the desired parent classes.

```python
class Example(PumpingStationMixin,
              LinearizedOrderGoalProgrammingMixin, GoalProgrammingMixin,
              PIMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
```

Now we define our pumping station objects, and store them in a local instance variable. We refer to this instance variable from the abstract method `pumping_stations()` we have to override.

```python

        # Here we define a list of pumping stations, each consisting of a list
```

(continues on next page)

(continued from previous page)

```
        # of pumps. In our case, there is only one pumping station containing
        # a single pump.
        self.__pumping_stations = [PumpingStation(self, 'pumpingstation1',
                                                  pump_symbols=['pumpingstation1.pump1
↪'],
                                                  energy_price_symbols='energy_price
↪')]
```

```
    def pumping_stations(self):
        # This is the method that we must implement. It has to return a list of
        # PumpingStation objects, which we already initialized in the __init__
        # function. So here we just return that list.
        return self.__pumping_stations
```

Then we append our water level range goal to the list of path goals from our parents classes:

```
    def path_goals(self):
        goals = super().path_goals()
        goals.append(WaterLevelRangeGoal(self))
        return goals
```

We also add a minimization goal for the costs, which has a default priority of `sys.maxsize`. This goal will automatically integrate the product of power and energy price for all pumps in all pumping stations,

```
    def goals(self):
        goals = super().goals()
        goals.append(MinimizePumpCostGoal(self))
        return goals
```

Finally, we want to apply some additional configuration, reducing the amount of information the solver outputs:

```
    def solver_options(self):
        options = super().solver_options()
        options['solver'] = 'cbc'
        options['casadi_solver'] = 'qpsol'
        return options
```

Run the Optimization Problem

To make our script run, at the bottom of our file we just have to call the `run_optimization_problem()` method we imported on the optimization problem class we just created.

```
run_optimization_problem(Example, base_folder='..')
```

The Whole Script

All together, the whole example script is as follows:

```
import os
from datetime import timedelta

from rtctools.optimization.collocated_integrated_optimization_problem import␣
↪CollocatedIntegratedOptimizationProblem
```

(continues on next page)

(continued from previous page)

```
from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin,␣
↪StateGoal
from rtctools.optimization.linearized_order_goal_programming_mixin import␣
↪LinearizedOrderGoalProgrammingMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.optimization.pi_mixin import PIMixin
from rtctools.util import run_optimization_problem

from rtctools_hydraulic_structures.pumping_station_mixin import \
    MinimizePumpCostGoal, PumpingStation, PumpingStationMixin, plot_operating_points


class WaterLevelRangeGoal(StateGoal):
    """
    Goal that tries to keep the water level minum and maximum water level,
    the values of which are read from the optimization problem.
    """

    state = 'storage.HQ.H'

    priority = 1

    def __init__(self, optimization_problem):
        self.target_min = optimization_problem.wl_min
        self.target_max = optimization_problem.wl_max

        _range = self.target_max - self.target_min
        self.function_range = (self.target_min - _range, self.target_max + _range)

        super().__init__(optimization_problem)


class Example(PumpingStationMixin,
              LinearizedOrderGoalProgrammingMixin, GoalProgrammingMixin,
              PIMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
    """
    An example showing the basic usage of the PumpingStationMixin. It consists of two␣
↪goals:
    1. Keep water level in the acceptable range.
    2. Minimize power usage for doing so.
    """

    # Set the target minimum and maximum water levels.
    wl_min, wl_max = (-0.5, 0)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.__output_folder = kwargs['output_folder']  # So we can write our␣
↪pictures to it

        # Here we define a list of pumping stations, each consisting of a list
        # of pumps. In our case, there is only one pumping station containing
        # a single pump.
        self.__pumping_stations = [PumpingStation(self, 'pumpingstation1',
                                                  pump_symbols=['pumpingstation1.pump1
↪'],
```

(continues on next page)

(continued from previous page)

```
                                                  energy_price_symbols='energy_price
↪')]

    def pumping_stations(self):
        # This is the method that we must implement. It has to return a list of
        # PumpingStation objects, which we already initialized in the __init__
        # function. So here we just return that list.
        return self.__pumping_stations

    def goals(self):
        goals = super().goals()
        goals.append(MinimizePumpCostGoal(self))
        return goals

    def path_goals(self):
        goals = super().path_goals()
        goals.append(WaterLevelRangeGoal(self))
        return goals

    def solver_options(self):
        options = super().solver_options()
        options['solver'] = 'cbc'
        options['casadi_solver'] = 'qpsol'
        return options

    def post(self):
        super().post()

        results = self.extract_results()

        # TODO: Currently we use hardcoded references to pump1. It would be
        # prettier if we could generalize this so we can handle an arbitrary
        # number of pumps. It would also be prettier to replace hardcoded
        # references to e.g. pumpingstation1.pump1__power with something like
        # pumpingstation1.pump.power(), if at all possible.

        # Calculate the total amount of energy used. Note that QHP fit was
        # made to power in W, and that our timestep is 1 hour.
        powers = results['pumpingstation1.pump1__power'][1:]
        total_power = sum(powers)/1000
        print("Total power = {} kWh".format(total_power))

        # Calculate total energy cost
        price = self.get_timeseries('energy_price').values[1:]
        cost = sum(powers*price)/1000
        print("Total cost = {} Euro".format(cost))

        # Make plots
        import matplotlib.dates as mdates
        import matplotlib.pyplot as plt
        import numpy as np

        plt.style.use('ggplot')

        def unite_legends(axes):
            handles, labels = [], []
            for ax in axes:
```

(continues on next page)

(continued from previous page)

```python
                tmp = ax.get_legend_handles_labels()
                handles.extend(tmp[0])
                labels.extend(tmp[1])
            return handles, labels

        # Discard history from plotting.
        i_start = list(self.get_timeseries('energy_price', 0).times).index(self.initial_time)

        # Plot #1: Data over time. X-axis is always time.
        f, axarr = plt.subplots(5, sharex=True)
        times = [self.io.reference_datetime + timedelta(seconds=s) for s in self.times()]

        axarr[0].set_ylabel('Water level\n[m]')
        axarr[0].plot(times, results['storage_level'], label='Polder',
                      linewidth=2, color='b')
        axarr[0].plot(times, self.wl_max * np.ones_like(times), label='Polder Max',
                      linewidth=2, color='r', linestyle='--')
        axarr[0].plot(times, self.wl_min * np.ones_like(times), label='Polder Min',
                      linewidth=2, color='g', linestyle='--')
        ymin, ymax = axarr[0].get_ylim()
        axarr[0].set_ylim(ymin - 0.1, ymax + 0.1)

        axarr[1].set_ylabel('Water level\n[m]')
        axarr[1].plot(times, self.get_timeseries('H_ext', 0).values[i_start:], label='Sea',
                      linewidth=2, color='b')
        ymin, ymax = axarr[1].get_ylim()
        axarr[1].set_ylim(ymin - 0.5, ymax + 0.5)

        axarr[2].set_ylabel('Energy price\n[EUR/kWh]')
        axarr[2].step(times, self.get_timeseries('energy_price', 0).values[i_start:], label='Energy price',
                      linewidth=2, color='b')
        ymin, ymax = axarr[2].get_ylim()
        axarr[2].set_ylim(-0.1, ymax + 0.1)

        axarr[3].set_ylabel('Discharge\n[$\\mathdefault{m^3\\!/s}$]')
        axarr[3].step(times, results['pumpingstation1.pump1.Q'], label='Pump',
                      linewidth=2, color='b')
        axarr[3].plot(times, self.get_timeseries('Q_in', 0).values[i_start:], label='Inflow',
                      linewidth=2, color='g')
        ymin, ymax = axarr[3].get_ylim()

        axarr[3].set_ylim(-0.05 * (ymax - ymin), ymax * 1.1)
        axarr[3].xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))

        axarr[4].set_ylabel('Pump speed\n[$\\mathdefault{min^{-1}}$]')
        axarr[4].step(times, results['pumpingstation1.pump1_speed'], label='Speed',
                      linewidth=2, color='b')
        ymin, ymax = axarr[4].get_ylim()
        axarr[4].set_ylim(-0.05 * (ymax - ymin), ymax * 1.1)
        axarr[4].xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))

        for ax in axarr:
```

(continues on next page)

(continued from previous page)

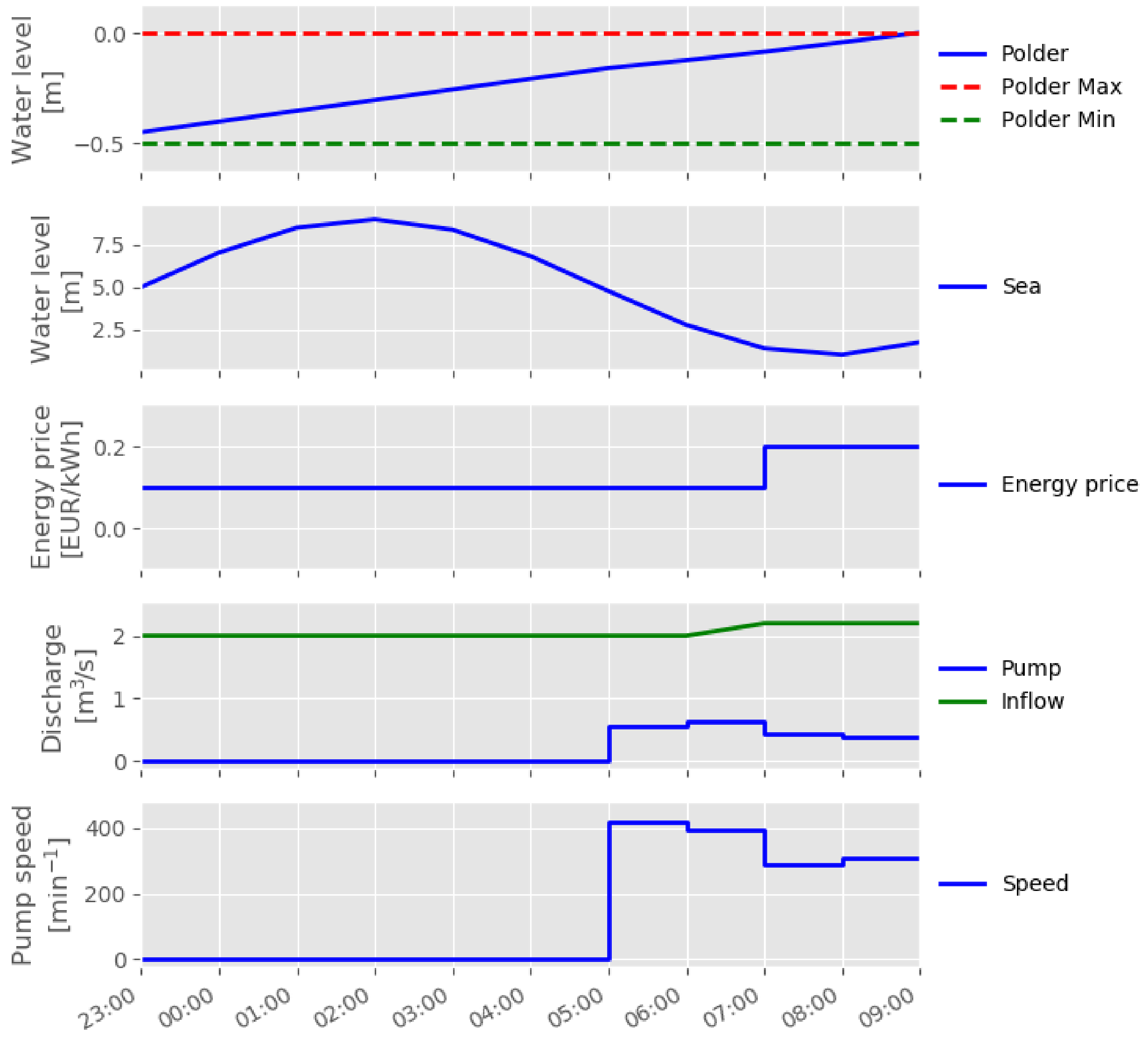
```
            ax.set_xlim(min(times), max(times))

        f.autofmt_xdate()

        # Shrink each axis by 20% and put a legend to the right of the axis
        for i in range(len(axarr)):
            box = axarr[i].get_position()
            axarr[i].set_position([box.x0, box.y0, box.width * 0.8, box.height])
            axarr[i].legend(loc='center left', bbox_to_anchor=(1, 0.5), frameon=False)

        # Output Plot
        f.set_size_inches(8, 9)
        plt.savefig(os.path.join(self.__output_folder, 'overall_results.png'), bbox_
↪inches='tight', pad_inches=0.1)

        # Plot the working area with the operating points of the pump.
        plot_operating_points(self, self.__output_folder)


# Run
run_optimization_problem(Example, base_folder='..')
```

Results

The results from the run are found in `output/timeseries_export.xml`. Any PI-XML-reading software can import it.

The `post()` method in our `Example` class also generates some pictures to help understand what is going on.

First we have an overview of the relevant boundary conditions and control variables.

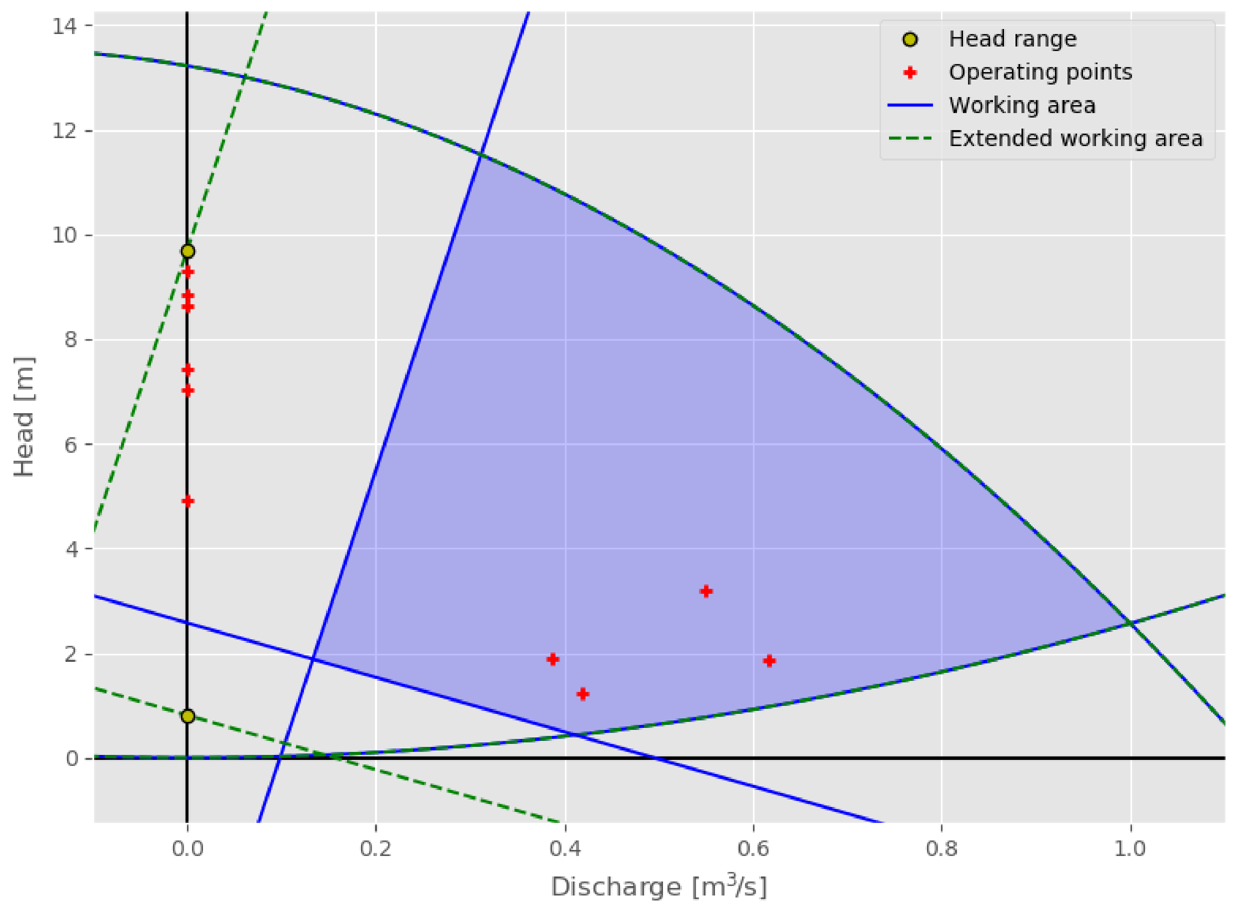


As expressed in the introduction of this example problem, we indeed see that the available buffer in the polder is used to its fullest. The water level at the final time step is (almost) equal to the maximum water level.

In this example, no historical data regarding the pump history is provided. Thus there is no prescribed behavior for the pump status on the initial time steps.

Furthermore, we see that the pump only discharges water when the water level is low. It is interesting to see that the optimal solution for costs means pumping at the lowest water level, even though the energy price is twice as high.

Using `plot_operating_points()` it is possible to generate a Q-H plot of the pump's working area and operating points, such as shown below. Here we see that the pumping at lower heads occurs with relatively large discharges, meaning that the pump is sometimes operating close to the edges of its working area.



Two Pumps

Note: This example focuses on how to put multiple pumps in a hydraulic model, and assumes basic exposure to RTC-Tools and the *`PumpingStationMixin`*. To start with basics of pump modeling, see *Basic Pumping Station*.

The purpose of this example is to understand the technical setup of a model with multiple pumps.

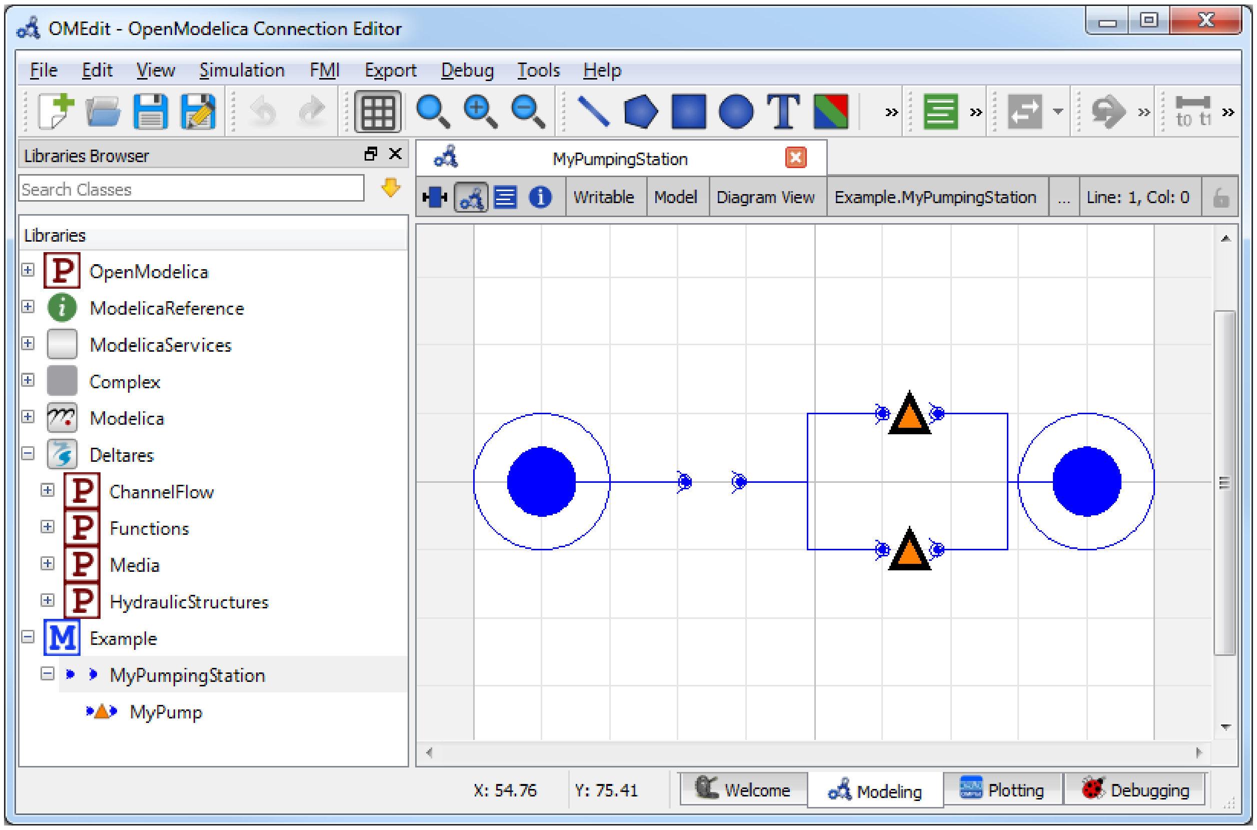
The scenario of this example is similar to that of *Basic Pumping Station* with the following exceptions:

- there are two available pumps instead of one;
- each pump has its associated price of energy;
- from historical data we know that `pump1` has been on during the two previous hours.

The folder `examples/pumping_station/two_pumps` contains the complete RTC- Tools optimization problem. The discussion below will focus on the differences from the *Basic Pumping Station*.

The Model

The pumping station object `MyPumpingStation` looks as follows in its diagram representation in OpenModelica:



When modeling multiple pumps of the same type, it makes sense to define a model, which can then be instantiated into multiple objects. In the file `Example.mo` this can be seen in the submodel `MyPump` of `MyPumpingStation`:

```
    model MyPump
      extends Deltares.HydraulicStructures.PumpingStation.Pump(
        power_coefficients = {{{-39095.7484857 ,  66783.26438029},
                               {  7281.58399033,        0.0}},
                              {{-27859.72396609,  52991.89678362},
                               {  6547.78028277,        0.0}},
                              {{-23247.11759041,  47371.07046571},
                               {  6106.48287875,        0.0}},
                              {{-33451.06236192,  59110.131313  },
                               {  7034.0025483,       0.0}},
                              {{-39861.93265223,  67925.88638803},
                               {  7281.81109575,        0.0}},
                              {{-18905.97159263,  40725.7120928 },
                               {  5785.42670305,        0.0}},
                              {{-69080.26073112,  97844.92974535},
                               {  8776.27578528,        0.0}},
                              {{-27357.47031974,  52938.41395038},
```

(continues on next page)

(continued from previous page)

```
                                              {   6406.26137405,        0.0}},
                                             {{-49670.43493511,   78653.65042992},
```

The data of this pump is exactly equal to that used in *Basic Pumping Station*, but is not instantiated yet. To instantiate two pumps using this data, we define two components `pump1` and `pump2`:

```
                                              {   8408.00958411,        0.0}},
                                             {{-22131.50613895,   39713.73557364},
```

Lastly, it is important not to forget to set the right number of pumps on the pumping station object:

```
model MyPumpingStation
  extends Deltares.HydraulicStructures.PumpingStation.PumpingStation(
    n_pumps=2
  );
```

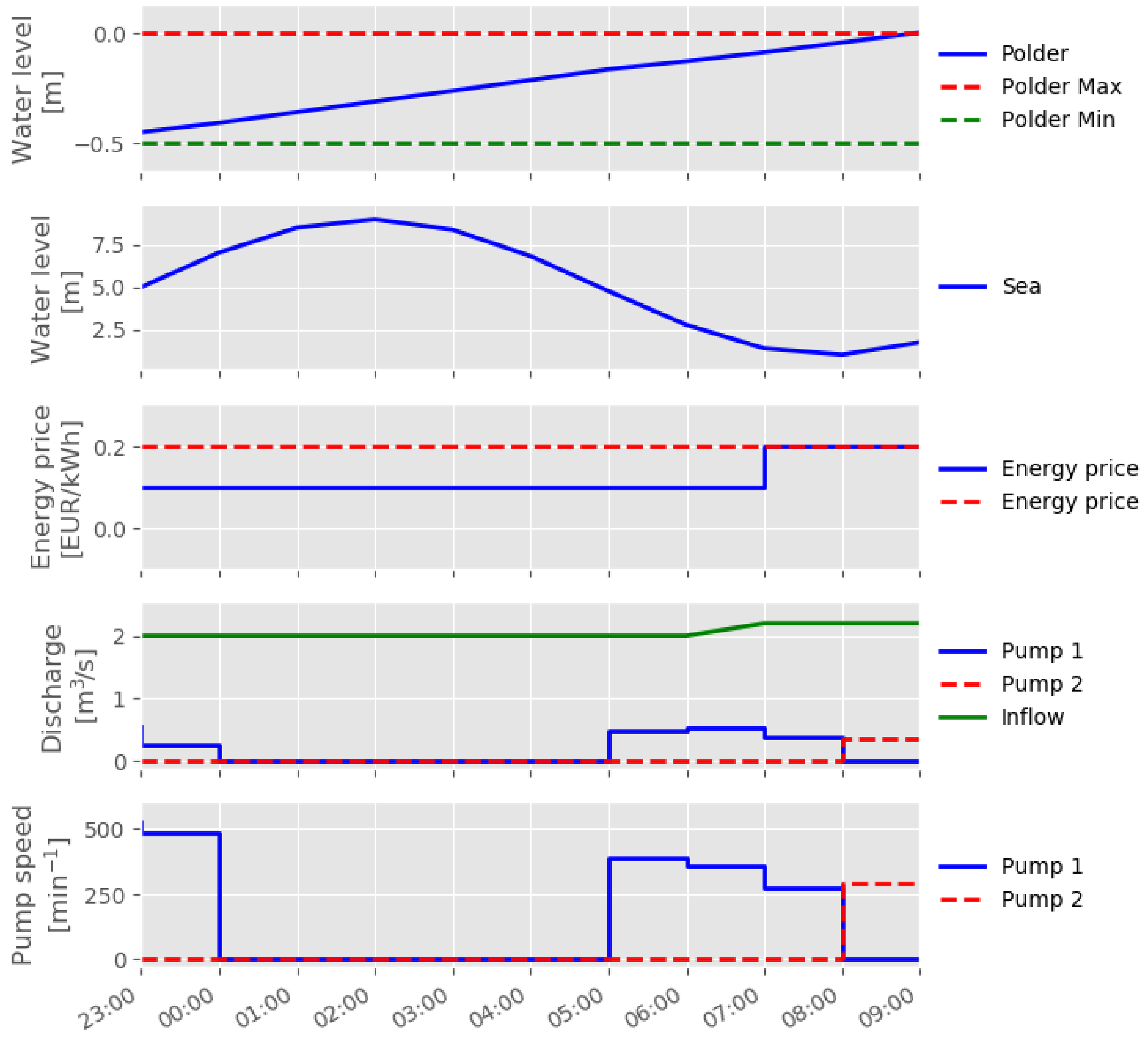
The Optimization Problem

When using multiple pumps it is important to specify the right order of pumps. This order should match the order of pumps in the `pump_switching_matrix`. The parameter `energy_price_symbol` is used to specify the energy price of each pump in the pumping station. This is either a string or a list of strings of names of the energy prices time series. If `energy_price_symbol` is a string, then all the pumps of the pumping station will have the same energy price. Else there must be as many energy price symbols as number of pumps.

```

        # Here we define a list of pumping stations, each consisting of a list
        # of pumps. In our case, there is only one pumping station containing
        # a single pump.
        self.__pumping_stations = [PumpingStation(self, 'pumpingstation1',
                                                  pump_symbols=['pumpingstation1.pump1
↪',
                                                                'pumpingstation1.pump2
↪'],
                                                  energy_price_symbols=['energy_price_
↪P1',
                                                                        'energy_price_
↪P2'],
                                                  status_history='{pump}_status_hist
↪')]
```

Results

We give an overview of the results.



As mentioned we assume that `pump1` has been on for the two previous hours, so it must be on for at least one more hour. This explains why it is on in the first time step. Beside that, the pumps discharge water when the water level is low even though the energy price is higher during that time frame.

Constant speed pump

Note: This example focuses on how to model a constant speed pump, and assumes basic exposure to RTC-Tools and the `PumpingStationMixin`. To start with basics of pump modeling, see *Basic Pumping Station*.

The purpose of this example is to understand the technical setup of a model of a constant speed pump.

The scenario of this example is equal to that of *Basic Pumping Station*, but with one constant speed pump instead of a variable speed pump. The resistance has also been removed. The folder `examples/pumping_station/two_pumps` contains the complete RTC- Tools optimization problem. The discussion below will focus on the differences from the *Basic Pumping Station*.

Note: The resistance has been removed because it created an artificial head loss to be able to pump at lower power energy prices.

The Model

The constant speed pump is modeled the same way as the variable speed one. The difference is the working area and the power approximation. This can be seen in the file `Example.mo`:

```
  Deltares.HydraulicStructures.PumpingStation.Pump pump1(
    power_coefficients = {{{-39095.7484857 ,  66783.26438029},
                           {  7281.58399033,       0.0}},
                          {{-27859.72396609,  52991.89678362},
                           {  6547.78028277,       0.0}},
                          {{-23247.11759041,  47371.07046571},
                           {  6106.48287875,       0.0}},
                          {{-33451.06236192,  59110.131313  },
                           {  7034.0025483,       0.0}},
                          {{-39861.93265223,  67925.88638803},
```

(continues on next page)

(continued from previous page)

```
                                  {   7281.81109575,       0.0}},
                                 {{-18905.97159263,  40725.7120928 },
                                  {   5785.42670305,       0.0}},
                                 {{-69080.26073112,  97844.92974535},
                                  {   8776.27578528,       0.0}},
                                 {{-27357.47031974,  52938.41395038},
                                  {   6406.26137405,       0.0}},
                                 {{-49670.43493511,  78653.65042992},
                                  {   7840.68764271,       0.0}},
```

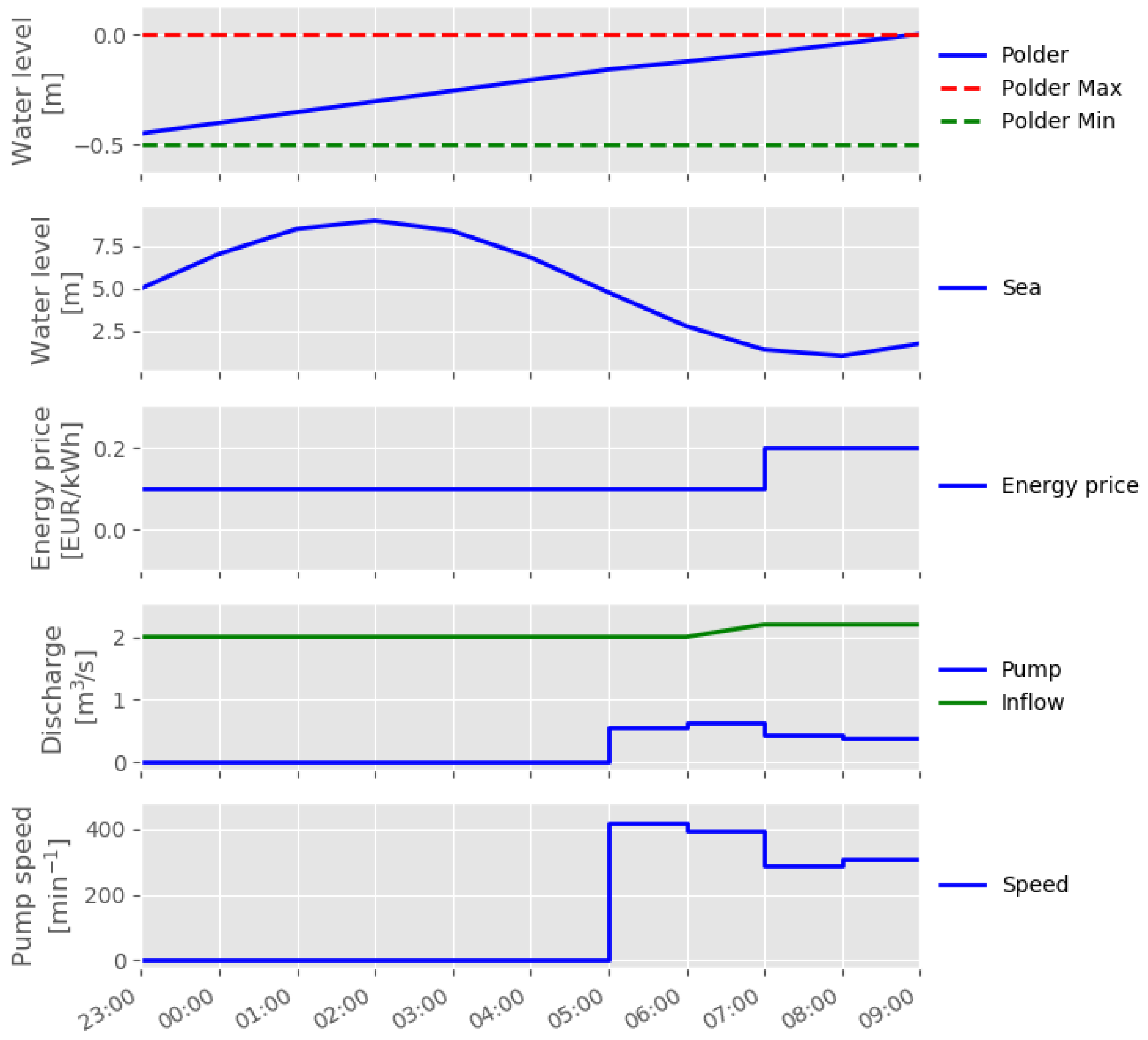
The interpretation and the calculation of these coefficients is explained in *Modelica API*. For constant speed pumps, the polynomial that defines the minimum and maximum speed is the same.

The Optimization Problem

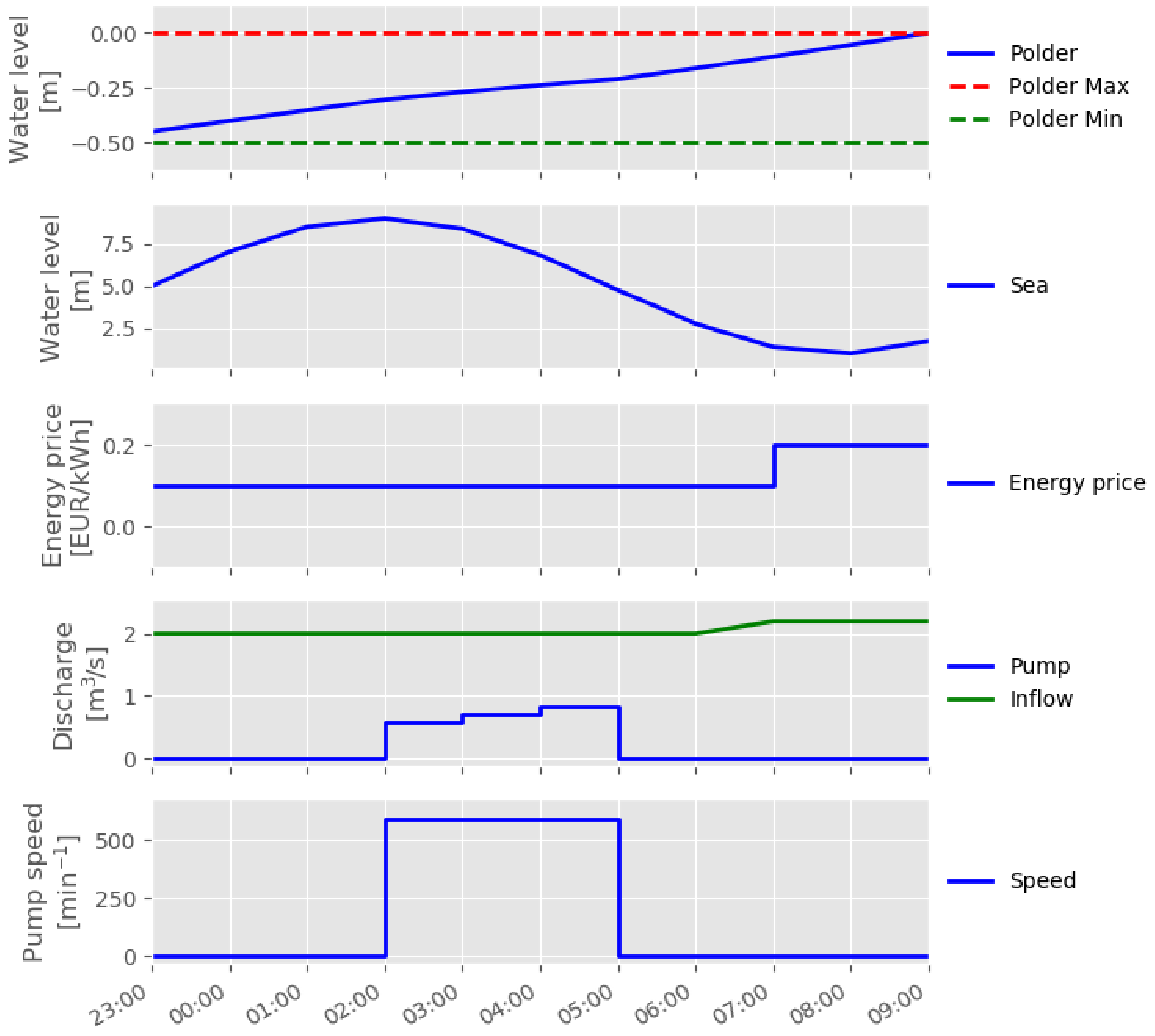
The optimization problem is exactly the same as for a variable speed pump.

Results

The same example is calculated with a variable and a constant speed pump. The constant speed pump's speed is equal to the maximum speed of the variable speed pump. The results with the variable speed pump

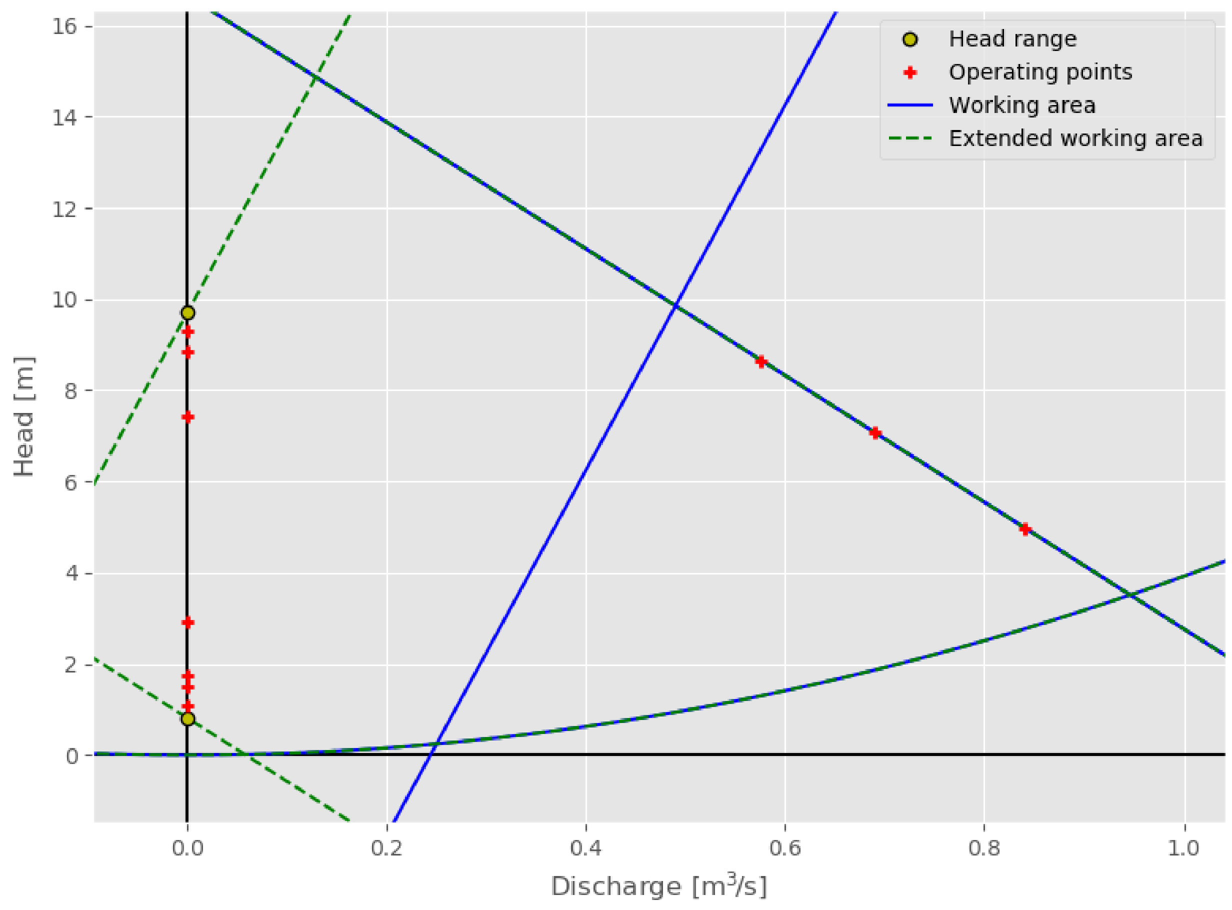


and the constant speed pump are shown below.



It can be seen that the constant speed pump was trying in a shorter time but pumping higher discharge. In this situation the constant speed pump therefore using three times more energy.

In the Q-H plot of the operating points we clearly see the pump operating on the linearized maximum pump speed line.



1.5.2 Weir

Basic Weir

Note: This example focuses on how to implement a controllable weir in RTC-Tools using the Hydraulic Structures library. It assumes basic exposure to RTC- Tools. If you are a first-time user of RTC-Tools, please refer to the

RTC-Tools documentation.

The weir structure is valid for two flow conditions:

- Free (critical) flow
- No flow

Warning: Submerged flow is not supported.

Modeling

Building a model with a weir

In this example we are considering a system of two branches and a controllable weir in between. On the upstream side is a prescribed input flow, and on the downstream side is a prescribed output flow. The weir should move in such way that the water level in both branches is kept within the desired limits.

To build this model, we need the following blocks:

- upstream and downstream discharge boundary conditions
- two branches
- a weir

By putting the blocks from the Modelica editor, the code is automatically generated (Note: this code snippet excludes the lines about the annotation and location):

```
  output Modelica.SIunits.Volume branch_2_water_level;
  Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge Upstream;
  Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge Downstream;
  Deltares.ChannelFlow.Hydraulic.Reservoir.Linear Branch1(A = 50, H_b = 0, H(nominal=1, min=0, max=100));
  Deltares.ChannelFlow.Hydraulic.Reservoir.Linear Branch2(A = 100, H_b = 0, H(nominal=1, min=0, max=100));
  Deltares.HydraulicStructures.Weir.Weir weir1(hw_max = 3, hw_min = 1.7, q_max = 1, q_min = 0, width = 10);
```

For the weir block, the dimensions of the weir should be set. It can be done either by double clicking to the block, or in the text editor. A controllable weir is represented with a weir block. This block has discharge and water level as input, and also as output. When a block is placed, the following parameters can be given: - `width`: the width of the crest in meters - `hw_min`: the minimum crest height - `hw_max`: the maximum crest height - `q_min`: the minimum expected discharge - `q_max`: the maximum expected discharge

The last two values should be estimated in such way that the discharge will not be able to go outside these bounds. However, for some linearization purposes, they should be as tight as possible. The values set by the text editor look like the line above.

The input variables are the upstream and downstream (known) discharges. The control variable - the variable that the algorithm changes until it achieves the desired results - is the discharge between the two branches. In practice, the weir height is the variable that we are interested in, but as it depends on the discharge between the two branches and the upstream water level, it will only be calculated in post processing. The input variables for the model are:

```
  input Modelica.SIunits.VolumeFlowRate upstream_q_ext(fixed = true);
  input Modelica.SIunits.VolumeFlowRate downstream_q_ext(fixed = true);
  input Modelica.SIunits.VolumeFlowRate WeirFlow1(fixed = false, nominal=1, min=0,␣
↪max=2.5);
```

Important: The min, max and nominal the values should always be meaningful. For nominal, set the value that the variable most likely takes.

As output, we are interested in the water level in the two branches:

```
  output Modelica.SIunits.Volume branch_1_water_level;
  output Modelica.SIunits.Volume branch_2_water_level;
```

Now we have to define the equations. We have to set the boundary conditions. First the discharge should be read from the external files:

```
  Upstream.Q = upstream_q_ext;
  Downstream.Q = downstream_q_ext;
```

And then the water level should be defined equal to the water level in the branch:

```
  Branch1.HQDown.H=Branch1.H;
  Branch2.HQDown.H=Branch2.H;
```

As we use reservoirs for branches, the variables we do not need should be zero:

```
  Branch1.Q_turbine=0;
  Branch2.Q_turbine=0;
```

Finally the outputs are set:

```
  branch_1_water_level = Branch1.H;
  branch_2_water_level = Branch2.H;
```

and the control variable as well:

```
  WeirFlow1 = weir1.Q;
```

The whole model file looks like this:

```
model WeirExample
  input Modelica.SIunits.VolumeFlowRate upstream_q_ext(fixed = true);
  input Modelica.SIunits.VolumeFlowRate downstream_q_ext(fixed = true);
  input Modelica.SIunits.VolumeFlowRate WeirFlow1(fixed = false, nominal=1, min=0,␣
↪max=2.5);
  output Modelica.SIunits.Volume branch_1_water_level;
  output Modelica.SIunits.Volume branch_2_water_level;
  Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge Upstream;
  Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge Downstream;
  Deltares.ChannelFlow.Hydraulic.Reservoir.Linear Branch1(A = 50, H_b = 0,␣
↪H(nominal=1, min=0, max=100));
  Deltares.ChannelFlow.Hydraulic.Reservoir.Linear Branch2(A = 100, H_b = 0,␣
↪H(nominal=1, min=0, max=100));
  Deltares.HydraulicStructures.Weir.Weir weir1(hw_max = 3, hw_min = 1.7, q_max = 1, q_
↪min = 0, width = 10);
```

(continues on next page)

(continued from previous page)

```
equation
  connect(weir1.HQDown, Branch2.HQUp);
  connect(Branch1.HQDown, weir1.HQUp);
  connect(Branch2.HQDown, Downstream.HQ);
  connect(Upstream.HQ, Branch1.HQUp);
  Branch1.HQDown.H=Branch1.H;
  Branch2.HQDown.H=Branch2.H;
  Branch1.Q_turbine=0;
  Branch2.Q_turbine=0;
  Upstream.Q = upstream_q_ext;
  Downstream.Q = downstream_q_ext;
  WeirFlow1 = weir1.Q;
  branch_1_water_level = Branch1.H;
  branch_2_water_level = Branch2.H;
end WeirExample;
```

Optimization

In this example, we would like to achieve that the water levels in the branches stay in the prescribed limits. The easiest way to achieve this objective is through goal programming. We will define two goals, one goal for each branch. The goal is that the water level should be higher than the given minimum and lower than the given maximum. Any solution satisfying these criteria is equally attractive for us. In practice, in goal programming the goal violation value is taken to the order'th power in the objective function (see: Goal Programming). In our example, we use the file `WeirExample.py`. We define a class, and apart from the usual classes that we import for optimization problems, we also have to import the class `WeirMixin`:

```
class WeirExample(WeirMixin, GoalProgrammingMixin, CSVMixin, ModelicaMixin,␣
↪CollocatedIntegratedOptimizationProblem):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Now we have to define the weirs: in quotation marks should be the same name as used for the Modelica model. Now there is only one weir, and the definition looks like:

```
        self.__weirs = [Weir(self, 'weir1')]
```

In case of more weirs, the names can be separated with a comma, for example:

```
self._weirs = [Weir('weir1'), Weir('weir2')]
```

Lastly we have to override the abstract method the returns the list of weirs:

```
    def weirs(self):
        return self.__weirs
```

Adding goals

In this example there are two branches connected with a weir. On the upstream side is a prescribed input flow, and on the downstream side is a prescribed output flow. The weir should move in such way, that the water level in both branches kept within the desired limits. We can add a water level goal for the upstream branch:

```python
class WLRangeGoal_01(StateGoal):

    def __init__(self, optimization_problem):
        self.state = 'Branch1.H'
        self.priority = 1

        self.target_min = optimization_problem.get_timeseries('h_min_branch1')
        self.target_max = optimization_problem.get_timeseries('h_max_branch1')

        super().__init__(optimization_problem)
```

A similar goal can be added to the downstream branch.

Setting the solver

As it is a mixed integer problem, it is handy to set some options to control the solver. In this example, we set the `allowable_gap` to 0.005. It is used to specify the value of absolute gap under which the algorithm stops. This is bigger than the default. This gives lower expectations for the acceptable solution, and in this way, the time of iteration is less. This value might be different for every problem and might be adjusted a trial-and-error basis. For more information, see the documentation of the BONMIN solver User's Manual)

The solver setting is the following:

```python
    def solver_options(self):
        options = super().solver_options()
        solver = options['solver']
        options[solver]['allowable_gap'] = 0.005
        options[solver]['print_level'] = 2
        return options
```

Input data

In order to run the optimization, we need to give the boundary conditions and the water level bounds. This data is given as time-series in the file `timeseries_import.csv`.

The whole python file

The optimization file looks like:

```python
from datetime import timedelta

from rtctools.optimization.collocated_integrated_optimization_problem \
    import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin,␣
↪StateGoal
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

from rtctools_hydraulic_structures.weir_mixin import Weir, WeirMixin, plot_operating_
↪points

```

(continues on next page)

(continued from previous page)

```
# There are two water level targets, with different priority.
# The water level should stay in the required range during all the simulation


class WLRangeGoal_01(StateGoal):

    def __init__(self, optimization_problem):
        self.state = 'Branch1.H'
        self.priority = 1

        self.target_min = optimization_problem.get_timeseries('h_min_branch1')
        self.target_max = optimization_problem.get_timeseries('h_max_branch1')

        super().__init__(optimization_problem)


class WLRangeGoal_02(StateGoal):

    def __init__(self, optimization_problem):
        self.state = 'Branch2.H'
        self.priority = 2

        self.target_min = optimization_problem.get_timeseries('h_min_branch2')
        self.target_max = optimization_problem.get_timeseries('h_max_branch2')

        super().__init__(optimization_problem)


class WeirExample(WeirMixin, GoalProgrammingMixin, CSVMixin, ModelicaMixin,␣
↪CollocatedIntegratedOptimizationProblem):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__weirs = [Weir(self, 'weir1')]
        self.__output_folder = kwargs['output_folder']  # So we can write our␣
↪pictures to it

    def weirs(self):
        return self.__weirs

    def solver_options(self):
        options = super().solver_options()
        solver = options['solver']
        options[solver]['allowable_gap'] = 0.005
        options[solver]['print_level'] = 2
        return options

    def path_goals(self):
        goals = super().path_goals()
        goals.append(WLRangeGoal_01(self))
        goals.append(WLRangeGoal_02(self))
        return goals

    def post(self):
        super().post()
        results = self.extract_results()

```

(continues on next page)

(continued from previous page)

```python
        # Make plots
        import matplotlib.dates as mdates
        import matplotlib.pyplot as plt
        import os

        plt.style.use('ggplot')

        def unite_legends(axes):
            handles, labels = [], []
            for ax in axes:
                tmp = ax.get_legend_handles_labels()
                handles.extend(tmp[0])
                labels.extend(tmp[1])
            return handles, labels

        # Plot #1: Data over time. X-axis is always time.
        f, axarr = plt.subplots(4, sharex=True)

        times = [self.io.reference_datetime + timedelta(seconds=s) for s in self.times()]
        weir = self.weirs()[0]

        axarr[0].set_ylabel('Water level\n[m]')
        axarr[0].plot(times, results['branch_1_water_level'], label='Upstream',
                      linewidth=2, color='b')
        axarr[0].plot(times, self.get_timeseries('h_min_branch1').values, label='Upstream Max',
                      linewidth=2, color='r', linestyle='--')
        axarr[0].plot(times, self.get_timeseries('h_max_branch1').values, label='Upstream Min',
                      linewidth=2, color='g', linestyle='--')
        ymin, ymax = axarr[0].get_ylim()
        axarr[0].set_ylim(ymin - 0.1, ymax + 0.1)

        axarr[1].set_ylabel('Water level\n[m]')
        axarr[1].plot(times, results['branch_2_water_level'], label='Downstream',
                      linewidth=2, color='b')
        axarr[1].plot(times, self.get_timeseries('h_max_branch2').values, label='Downstream Max',
                      linewidth=2, color='r', linestyle='--')
        axarr[1].plot(times, self.get_timeseries('h_min_branch2').values, label='Downstream Min',
                      linewidth=2, color='g', linestyle='--')
        ymin, ymax = axarr[1].get_ylim()
        axarr[1].set_ylim(ymin - 0.1, ymax + 0.1)

        axarr[2].set_ylabel('Discharge\n[$\\mathdefault{m^3\\!/s}$]')
        # We need the first point for plotting, but its value does not
        # necessarily make sense as it is not included in the optimization.
        weir_flow = results['WeirFlow1']
        weir_height = results["weir1_height"]
        minimum_water_level_above_weir = (weir_flow-weir.q_nom)/weir.slope + weir.h_nom

        minimum_weir_height = minimum_water_level_above_weir - ((weir_flow/weir.c_weir)**(2.0/3.0))

```

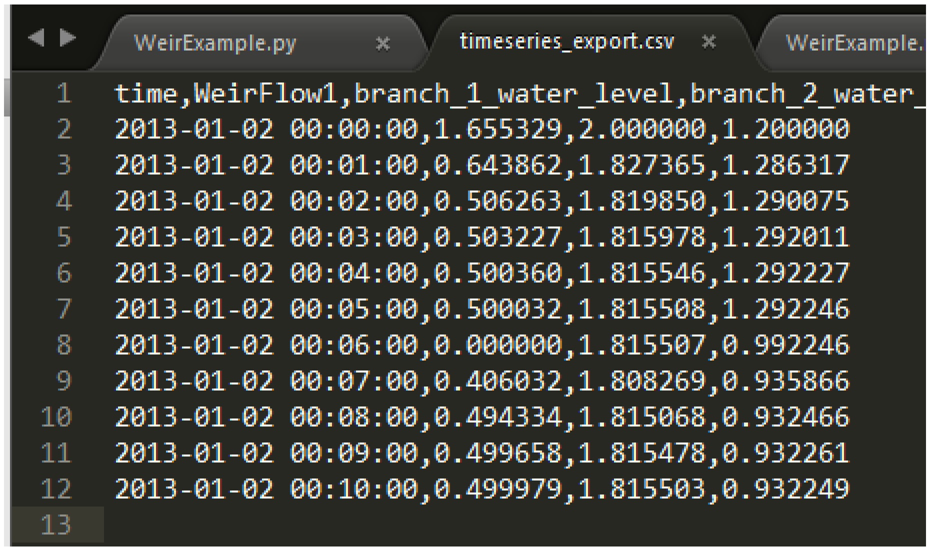
(continues on next page)

(continued from previous page)

```
        minimum_weir_height[0] = minimum_weir_height[1]

        weir_flow = results['WeirFlow1']
        weir_flow[0] = weir_flow[1]

        axarr[2].step(times, weir_flow, label='Weir',
                      linewidth=2, color='b')
        axarr[2].step(times, self.get_timeseries('upstream_q_ext').values, label=
↪'Inflow',
                      linewidth=2, color='r', linestyle='--')
        axarr[2].step(times, -1 * self.get_timeseries('downstream_q_ext').values,␣
↪label='Outflow',
                      linewidth=2, color='g', linestyle='--')
        ymin, ymax = axarr[2].get_ylim()
        axarr[2].set_ylim(-0.1, ymax + 0.1)

        weir_height = results["weir1_height"]
        weir_height[0] = weir_height[1]

        axarr[3].set_ylabel('Weir height\n[m]')
        axarr[3].step(times, weir_height, label='Weir',
                      linewidth=2, color='b')
        ymin, ymax = axarr[3].get_ylim()
        ymargin = 0.1 * (ymax - ymin)
        axarr[3].set_ylim(ymin - ymargin, ymax + ymargin)
        axarr[3].xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
        f.autofmt_xdate()

        # Shrink each axis by 20% and put a legend to the right of the axis
        for ax in axarr:
            box = ax.get_position()
            ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
            ax.set_xlim(min(times), max(times))
            ax.legend(loc='center left', bbox_to_anchor=(1, 0.5), frameon=False)

        # Output Plot
        f.set_size_inches(8, 9)
        plt.savefig(os.path.join(self.__output_folder, 'overall_results.png'),
                    bbox_inches='tight', pad_inches=0.1)

        plot_operating_points(self, self.__output_folder, results)


if __name__ == "__main__":
    run_optimization_problem(WeirExample)
```

Results

After successful optimization the results are printed in the time series export file. After running this example, the following results are expected:

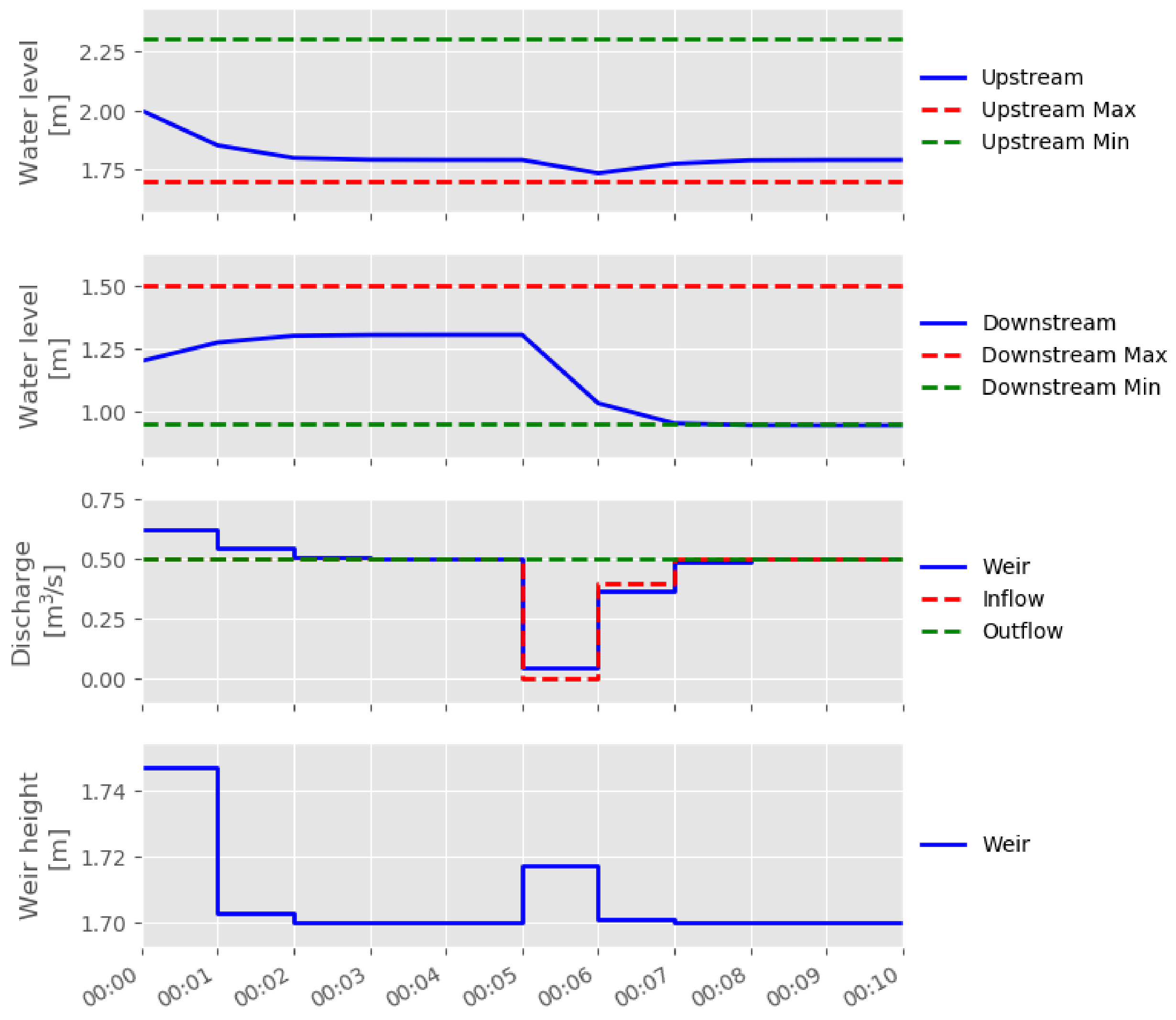


```
time,WeirFlow1,branch_1_water_level,branch_2_water_
2013-01-02 00:00:00,1.655329,2.000000,1.200000
2013-01-02 00:01:00,0.643862,1.827365,1.286317
2013-01-02 00:02:00,0.506263,1.819850,1.290075
2013-01-02 00:03:00,0.503227,1.815978,1.292011
2013-01-02 00:04:00,0.500360,1.815546,1.292227
2013-01-02 00:05:00,0.500032,1.815508,1.292246
2013-01-02 00:06:00,0.000000,1.815507,0.992246
2013-01-02 00:07:00,0.406032,1.808269,0.935866
2013-01-02 00:08:00,0.494334,1.815068,0.932466
2013-01-02 00:09:00,0.499658,1.815478,0.932261
2013-01-02 00:10:00,0.499979,1.815503,0.932249
```

The file found in the example folder includes some visualization routines.

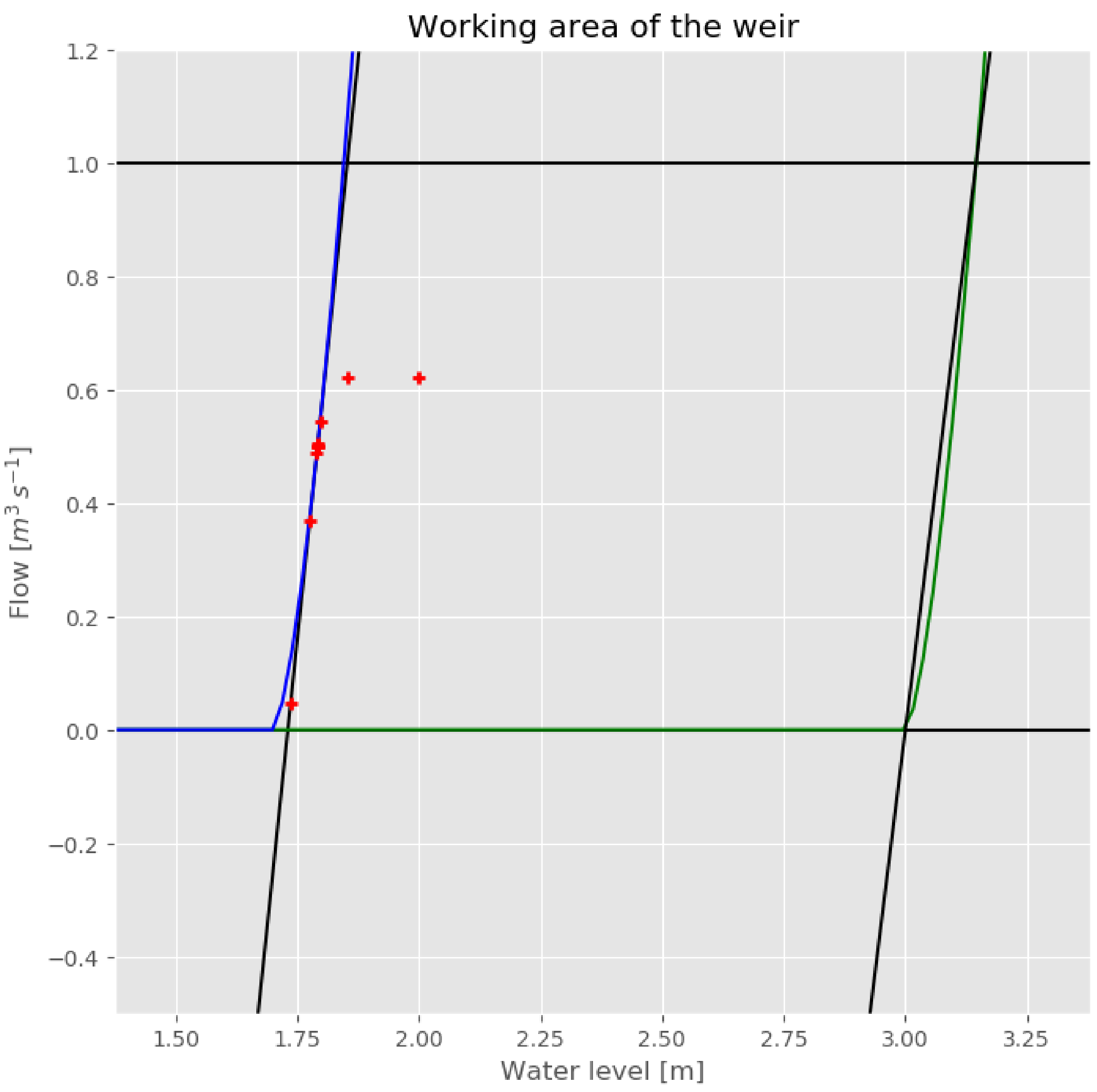
Interpretation of the results

The results of this simulation are summarized in the following figure:



In this example, the input flow increases after 6 minutes, while the downstream flow is kept constant. While the inflow drops after 6 minutes, the result is not seen in the upstream branch, because the weir moves up to compensate it. After the weir moved up, the water level drops in the downstream branch.

Using `plot_operating_points()` it is possible to generate a Q-H plot of the weir's working area and operating points, such as shown below. Here we see that it would have been possible to choose a slightly lower upper bound for the working area, to include more of the possible working area in the lower left corner. We also see that the weir tends to operate almost entirely on the left edge of the working area, i.e. at the lowest possible head difference.


Working area of the weir
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
Flow [$m^3 s^{-1}$]
1.50
1.75
2.00
2.25
2.50
2.75
3.00
3.25
Water level [m]

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index