
RTC-Tools Documentation

Release 2.2.0b1+113.ge0e434a

Jorn Baayen, Matthijs den Toom, et al.

Nov 15, 2018

Contents

1	Contents	3
1.1	Getting Started	3
1.2	Optimization	5
1.3	Simulation	23
1.4	Examples	26
2	Indices and tables	79



This is the documentation for RTC-Tools, the [Deltares](#) toolbox for control and optimization of environmental systems.

Visit the [RTC-Tools website](#) for a general product description and information on available services.

This first chapter covers getting the software running on your computer. The subsequent two chapters describe the RTC-Tools Python API. The fourth and final chapter discusses several illustrative examples, including the use of goal programming for multi-objective optimization, as well as the use of forecast ensembles.

1.1 Getting Started

1.1.1 Installation

For most users, the easiest way to install RTC-Tools using the `pip` package manager.

Using the Pip Package Manager

Although not required, it is recommended to install RTC-Tools in a virtual environment. See the [official Python tutorial](#) for more information on how to set up and activate a virtual environment.

RTC-Tools, including its dependencies, can be installed using the `pip` package manager:

```
# Install RTC-Tools and Channel Flow using pip package manager
pip install rtc-tools rtc-tools-channel-flow
```

From Source

The latest RTC-Tools and Channel Flow source can be downloaded using `git`:

```
# Get RTC-Tools source
git clone https://gitlab.com/deltares/rtc-tools.git

# Get RTC-Tools's Modelica library
git clone https://gitlab.com/deltares/rtc-tools-channel-flow.git
```

Then you can install this latest version as follows:

```
pip install ./rtc-tools
pip install ./rtc-tools-channel-flow
```

Or if you would like to have an editable installation (e.g. as developer):

```
pip install -e ./rtc-tools
pip install -e ./rtc-tools-channel-flow
```

1.1.2 Downloading and running examples

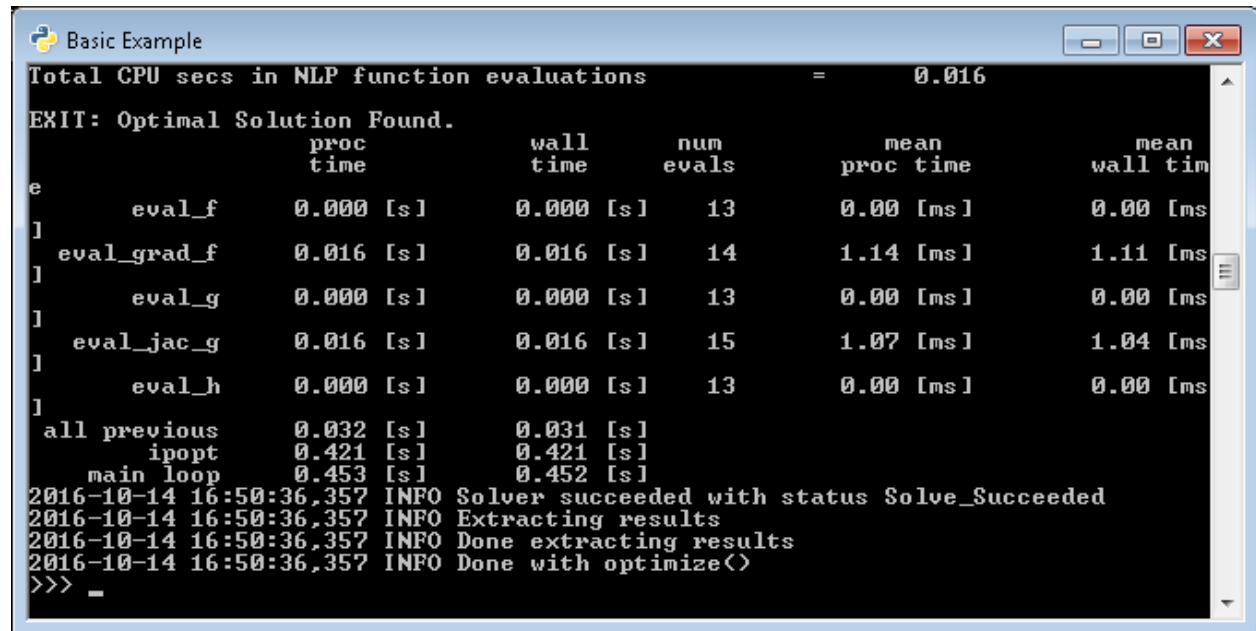
To check whether the installation was succesful, the basic example can be used. If RTC-Tools was not installed from source, the examples need to be downloaded first:

```
# Download the examples to the current folder (.)
rtc-tools-download-examples .

# Navigate to the basic example
cd rtc-tools-examples/basic/src

# Run the example
python example.py
```

If the installation was succesful, you should see that the solver succeeds:



```
Basic Example
Total CPU secs in NLP function evaluations = 0.016
EXIT: Optimal Solution Found.
      proc      wall      num      mean      mean
      time     time     evals    proc time    wall tim
e      eval_f    0.000 [s]    0.000 [s]    13      0.00 [ms]    0.00 [ms]
]      eval_grad_f 0.016 [s]    0.016 [s]    14      1.14 [ms]    1.11 [ms]
]      eval_g      0.000 [s]    0.000 [s]    13      0.00 [ms]    0.00 [ms]
]      eval_jac_g  0.016 [s]    0.016 [s]    15      1.07 [ms]    1.04 [ms]
]      eval_h      0.000 [s]    0.000 [s]    13      0.00 [ms]    0.00 [ms]
]      all previous 0.032 [s]    0.031 [s]
      ipopt      0.421 [s]    0.421 [s]
      main loop   0.453 [s]    0.452 [s]
2016-10-14 16:50:36.357 INFO Solver succeeded with status Solve_Succeeded
2016-10-14 16:50:36.357 INFO Extracting results
2016-10-14 16:50:36.357 INFO Done extracting results
2016-10-14 16:50:36.357 INFO Done with optimize()
>>> _
```

Elsewhere in this documentation we refer to the folder containing the examples as <examples directory>. Depending on the method of installation this can then either be:

- \path\to\rtc-tools-examples, when having downloaded the examples
- \path\to\source\of\rtc-tools\examples, when having installed RTC-Tools from source

1.1.3 Copying Modelica libraries

Because the Modelica libraries are distributed as pip packages, their location inside Python's site-packages can be somewhat inconvenient. To copy the Modelica libraries to a more convenient location, you can use the `rtc-tools-copy-libraries` command:


```
# Copy all Modelica libraries of RTC-Tools to the current folder (.)
rtc-tools-copy-libraries .
```

You should now have a folder `Deltares`, containing amongst others a package `.mo` file, a `ChannelFlow` folder and folders of any other RTC-Tools extensions you installed.

Elsewhere in this documentation we refer to the library folder containing the `Deltares` folder as `<library directory>`.

1.1.4 Getting OMEdit

RTC-Tools uses the Modelica language to describe the mathematics of the system we wish to optimize. There are several editors for Modelica models, but the OpenModelica Connection Editor, or OMEdit, is a free and open-source graphical connection editor that can be used to construct RTC-Tools models. To download it for windows, click here: <https://www.openmodelica.org/download/download-windows>

Once installed, you can start OMEdit by clicking:

```
Start -> All Programs -> OpenModelica -> OpenModelica Connection Editor
```

With OMEdit installed, you can start using it by following along with the basic example, *Filling a Reservoir*.

1.1.5 Running RTC-Tools

RTC-Tools is run from a command line shell. On Windows, both `PowerShell` and `cmd` can be used. On Linux/MacOS you could use the terminal application with a shell of your liking.

Once you have started the shell and loaded the correct virtual environment (if applicable), navigate to the `src` directory of the case you wish to optimize, e.g.:

```
cd \path\to\rtc-tools-examples\basic\src
```

Then, to run the case with RTC-Tools, run the `src` python script, e.g.:

```
python example.py
```

You will see the progress of RTC-Tools in your shell. All your standard shell commands can be used in the RTC-Tools shell. For example, you can use:

```
python example.py > log.txt
```

to pipe RTC-Tools output to a log file.

1.2 Optimization

Contents:

1.2.1 Basics

```
class rtctools.optimization.timeseries.Timeseries (times:  numpy.ndarray,  values:
                                                    Union[numpy.ndarray,  list,
                                                    casadi.casadi.DM])

    Bases: object
```

Time series object, bundling time stamps with values.

__init__ (*times: numpy.ndarray, values: Union[numpy.ndarray, list, casadi.casadi.DM]*)
 Create a new time series object.

Parameters

- **times** – Iterable of time stamps.
- **values** – Iterable of values.

times

Array of time stamps.

values

Array of values.

class `rtctools.optimization.optimization_problem.OptimizationProblem` (***kwargs*)
 Bases: `object`

Base class for all optimization problems.

bounds () → `rtctools._internal.alias_tools.AliasDict`

Returns variable bounds as a dictionary mapping variable names to a pair of bounds. A bound may be a constant, or a time series.

Returns A dictionary of variable names and (upper, lower) bound pairs. The bounds may be numbers or *Timeseries* objects.

Example:

```
def bounds(self):
    return {'x': (1.0, 2.0), 'y': (2.0, 3.0)}
```

constant_inputs (*ensemble_member: int*) → `rtctools._internal.alias_tools.AliasDict`

Returns a dictionary of constant inputs.

Parameters **ensemble_member** – The ensemble member index.

Returns A dictionary of constant input names and time series.

constraints (*ensemble_member: int*) → `List[Tuple[casadi.casadi.MX, Union[float, numpy.ndarray], Union[float, numpy.ndarray]]]`

Returns a list of constraints for the given ensemble member.

Call `OptimizationProblem.state_at()` to return a symbol representing a model variable at a given time.

Parameters **ensemble_member** – The ensemble member index.

Returns A list of triples (*f*, *m*, *M*), with an MX object representing the constraint function *f*, lower bound *m*, and upper bound *M*. The bounds must be numbers.

Example:

```
def constraints(self, ensemble_member):
    t = 1.0
    constraint1 = (
        2 * self.state_at('x', t, ensemble_member),
        2.0, 4.0)
    constraint2 = (
        self.state_at('x', t, ensemble_member) + self.state_at('y', t,
    ensemble_member),
```

(continues on next page)

(continued from previous page)

```
2.0, 3.0)
return [constraint1, constraint2]
```

control (*variable: str*) → casadi.casadi.MX

Returns an MX symbol for the given control input, not bound to any time.

Parameters **variable** – Variable name.

Returns MX symbol for given control input.

Raises KeyError

control_at (*variable: str, t: float, ensemble_member: int = 0, scaled: bool = False*) → casadi.casadi.MX

Returns an MX symbol representing the given control input at the given time.

Parameters

- **variable** – Variable name.
- **t** – Time.
- **ensemble_member** – The ensemble member index.
- **scaled** – True to return the scaled variable.

Returns MX symbol representing the control input at the given time.

Raises KeyError

delayed_feedback () → List[Tuple[str, str, float]]

Returns the delayed feedback mappings. These are given as a list of triples (x, y, τ) , to indicate that $y = x(t - \tau)$.

Returns A list of triples.

Example:

```
def delayed_feedback(self):
    fb1 = ['x', 'y', 0.1]
    fb2 = ['x', 'z', 0.2]
    return [fb1, fb2]
```

der (*variable: str*) → casadi.casadi.MX

Returns an MX symbol for the time derivative given state, not bound to any time.

Parameters **variable** – Variable name.

Returns MX symbol for given state.

Raises KeyError

der_at (*variable: str, t: float, ensemble_member: int = 0*) → casadi.casadi.MX

Returns an expression for the time derivative of the specified variable at time t.

Parameters

- **variable** – Variable name.
- **t** – Time.
- **ensemble_member** – The ensemble member index.

Returns MX object representing the derivative.

Raises KeyError

ensemble_member_probability (*ensemble_member: int*) → float

The probability of an ensemble member occurring.

Parameters **ensemble_member** – The ensemble member index.

Returns The probability of an ensemble member occurring.

Raises IndexError

ensemble_size

The number of ensemble members.

get_timeseries (*variable: str, ensemble_member: int = 0*) → `rtc-tools.optimization.timeseries.Timeseries`

Looks up a timeseries from the internal data store.

Parameters

- **variable** – Variable name.
- **ensemble_member** – The ensemble member index.

Returns The requested time series.

Return type *Timeseries*

Raises KeyError

history (*ensemble_member: int*) → `rtctools._internal.alias_tools.AliasDict`

Returns the state history. Uses the `initial_state()` method by default.

Parameters **ensemble_member** – The ensemble member index.

Returns A dictionary of variable names and historical time series (up to and including t0).

initial_state (*ensemble_member: int*) → `rtctools._internal.alias_tools.AliasDict`

The initial state.

The default implementation uses t0 data returned by the `history` method.

Parameters **ensemble_member** – The ensemble member index.

Returns A dictionary of variable names and initial state (t0) values.

initial_time

The initial time in seconds.

integral (*variable: str, t0: float = None, tf: float = None, ensemble_member: int = 0*) → `casadi.casadi.MX`

Returns an expression for the integral over the interval [t0, tf].

Parameters

- **variable** – Variable name.
- **t0** – Left bound of interval. If equal to None, the initial time is used.
- **tf** – Right bound of interval. If equal to None, the final time is used.
- **ensemble_member** – The ensemble member index.

Returns MX object representing the integral.

Raises KeyError

interpolate (*t: Union[float, numpy.ndarray], ts: numpy.ndarray, fs: numpy.ndarray, f_left: float = nan, f_right: float = nan, mode: int = 0*) → `Union[float, numpy.ndarray]`

Linear interpolation over time.

Parameters

- **t** (*float or vector of floats*) – Time at which to evaluate the interpolant.
- **ts** (*numpy array*) – Time stamps.
- **fs** – Function values at time stamps ts.
- **f_left** – Function value left of leftmost time stamp.
- **f_right** – Function value right of rightmost time stamp.
- **mode** – Interpolation mode.

Returns The interpolated value.

lookup_tables (*ensemble_member: int*) → `rtctools._internal.alias_tools.AliasDict`

Returns a dictionary of lookup tables.

Parameters **ensemble_member** – The ensemble member index.

Returns A dictionary of variable names and lookup tables.

objective (*ensemble_member: int*) → `casadi.casadi.MX`

The objective function for the given ensemble member.

Call `OptimizationProblem.state_at()` to return a symbol representing a model variable at a given time.

Parameters **ensemble_member** – The ensemble member index.

Returns An MX object representing the objective function.

Example:

```
def objective(self, ensemble_member):
    # Return value of state 'x' at final time:
    times = self.times()
    return self.state_at('x', times[-1], ensemble_member)
```

optimize (*preprocessing: bool = True, postprocessing: bool = True, log_solver_failure_as_error: bool = True*) → `bool`

Perform one initialize-transcribe-solve-finalize cycle.

Parameters

- **preprocessing** – True to enable a call to `pre` preceding the optimization.
- **postprocessing** – True to enable a call to `post` following the optimization.

Returns True on success.

parameters (*ensemble_member: int*) → `rtctools._internal.alias_tools.AliasDict`

Returns a dictionary of parameters.

Parameters **ensemble_member** – The ensemble member index.

Returns A dictionary of parameter names and values.

path_constraints (*ensemble_member: int*) → `List[Tuple[casadi.casadi.MX, Union[float, numpy.ndarray], Union[float, numpy.ndarray]]]`

Returns a list of path constraints.

Path constraints apply to all times and ensemble members simultaneously. Call `OptimizationProblem.state()` to return a time- and ensemble-member-independent symbol representing a model variable.

Parameters `ensemble_member` – The ensemble member index. This index may only be used to supply member-dependent bounds.

Returns A list of triples (f, m, M) , with an MX object representing the path constraint function f , lower bound m , and upper bound M . The bounds may be numbers or *Timeseries* objects.

Example:

```
def path_constraints(self, ensemble_member):
    # 2 * x must lie between 2 and 4 for every time instance.
    path_constraint1 = (2 * self.state('x'), 2.0, 4.0)
    # x + y must lie between 2 and 3 for every time instance
    path_constraint2 = (self.state('x') + self.state('y'), 2.0, 3.0)
    return [path_constraint1, path_constraint2]
```

path_objective (*ensemble_member: int*) → casadi.casadi.MX

Returns a path objective the given ensemble member.

Path objectives apply to all times and ensemble members simultaneously. Call *OptimizationProblem.state()* to return a time- and ensemble-member-independent symbol representing a model variable.

Parameters `ensemble_member` – The ensemble member index. This index is currently unused, and here for future use only.

Returns A MX object representing the path objective.

Example:

```
def path_objective(self, ensemble_member):
    # Minimize x(t) for all t
    return self.state('x')
```

post () → None

Postprocessing logic is performed here.

pre () → None

Preprocessing logic is performed here.

seed (*ensemble_member: int*) → rtctools._internal.alias_tools.AliasDict

Seeding data. The optimization algorithm is seeded with the data returned by this method.

Parameters `ensemble_member` – The ensemble member index.

Returns A dictionary of variable names and seed time series.

set_timeseries (*variable: str, timeseries: rtctools.optimization.timeseries.Timeseries, ensemble_member: int = 0, output: bool = True, check_consistency: bool = True*) → None

Sets a timeseries in the internal data store.

Parameters

- **variable** – Variable name.
- **timeseries** (iterable of floats, or *Timeseries*) – Time series data.
- **ensemble_member** – The ensemble member index.
- **output** – Whether to include this time series in output data files.
- **check_consistency** – Whether to check consistency between the time stamps on the new timeseries object and any existing time stamps.

solver_options () → Dict[str, Union[str, int, float]]

Returns a dictionary of CasADi optimization problem solver options.

The default solver for continuous problems is [Ipopt](#). The default solver for mixed integer problems is [Bonmin](#).

Returns A dictionary of solver options. See the CasADi and respective solver documentation for details.

solver_success (solver_stats: Dict[str, Union[str, bool]], log_solver_failure_as_error: bool) → Tuple[bool, int]

Translates the returned solver statistics into a boolean and log level to indicate whether the solve was succesful, and how to log it.

Parameters

- **solver_stats** – Dictionary containing information about the solver status. See explanation below.
- **log_solver_failure_as_error** – Indicates whether a solve failure Should be logged as an error or info message.

solver_stats typically consist of three fields:

- return_status: str
- secondary_return_status: str
- success: bool

By default we rely on CasADi's interpretation of the return_status (and secondary status) to the success variable, with an exception for IPOPT (see below).

The logging level is typically logging.INFO for success, and logging.ERROR for failure. Only for IPOPT an exception is made for *Not_Enough_Degrees_Of_Freedom*, which returns logging.WARNING instead. For example, this can happen when too many goals are specified, and lower priority goals cannot improve further on the current result.

Returns A tuple indicating whether or not the solver has succeeded, and what level to log it with.

state (variable: str) → casadi.casadi.MX

Returns an MX symbol for the given state, not bound to any time.

Parameters **variable** – Variable name.

Returns MX symbol for given state.

Raises KeyError

state_at (variable: str, t: float, ensemble_member: int = 0, scaled: bool = False) → casadi.casadi.MX

Returns an MX symbol representing the given variable at the given time.

Parameters

- **variable** – Variable name.
- **t** – Time.
- **ensemble_member** – The ensemble member index.
- **scaled** – True to return the scaled variable.

Returns MX symbol representing the state at the given time.

Raises KeyError

states_in (*variable: str, t0: float = None, tf: float = None, ensemble_member: int = 0*) → Iterator[casadi.casadi.MX]
 Iterates over symbols for states in the interval [t0, tf].

Parameters

- **variable** – Variable name.
- **t0** – Left bound of interval. If equal to None, the initial time is used.
- **tf** – Right bound of interval. If equal to None, the final time is used.
- **ensemble_member** – The ensemble member index.

Raises KeyError

timeseries_at (*variable: str, t: float, ensemble_member: int = 0*) → float
 Return the value of a time series at the given time.

Parameters

- **variable** – Variable name.
- **t** – Time.
- **ensemble_member** – The ensemble member index.

Returns The interpolated value of the time series.

Raises KeyError

`rtctools.util.run_optimization_problem(optimization_problem_class, base_folder='..', log_level=20, profile=False)`

Sets up and solves an optimization problem.

This function makes the following assumptions:

1. That the `base_folder` contains subfolders `input`, `output`, and `model`, containing input data, output data, and the model, respectively.
2. When using `CSVLookupTableMixin`, that the base folder contains a subfolder `lookup_tables`.
3. When using `ModelicaMixin`, that the base folder contains a subfolder `model`.
4. When using `ModelicaMixin`, that the toplevel Modelica model name equals the class name.

Parameters

- **optimization_problem_class** – Optimization problem class to solve.
- **base_folder** – Base folder.
- **log_level** – The log level to use.
- **profile** – Whether or not to enable profiling.

Returns `OptimizationProblem` instance.

1.2.2 Time discretization

class `rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegrated`
 Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Discretizes your model using a mixed collocation/integration scheme.

Collocation means that the discretized model equations are included as constraints between state variables in the optimization problem.

Note: To ensure that your optimization problem only has globally optimal solutions, any model equations that are collocated must be linear. By default, all model equations are collocated, and linearity of the model equations is verified. Working with non-linear models is possible, but discouraged.

Variables `check_collocation_linearity` – If `True`, check whether collocation constraints are linear. Default is `True`.

`integrated_states`

A list of states that are integrated rather than collocated.

Warning: This is an experimental feature.

`integrator_options()`

Configures the implicit function used for time step integration.

Returns A dictionary of CasADi `rootfinder` options. See the CasADi documentation for details.

`interpolation_method` (*variable=None*)

Interpolation method for variable.

Parameters `variable` – Variable name.

Returns Interpolation method for the given variable.

`theta`

RTC-Tools discretizes differential equations of the form

$$\dot{x} = f(x, u)$$

using the θ -method

$$x_{i+1} = x_i + \Delta t [\theta f(x_{i+1}, u_{i+1}) + (1 - \theta)f(x_i, u_i)]$$

The default is $\theta = 1$, resulting in the implicit or backward Euler method. Note that in this case, the control input at the initial time step is not used.

Set $\theta = 0$ to use the explicit or forward Euler method. Note that in this case, the control input at the final time step is not used.

Warning: This is an experimental feature for $0 < \theta < 1$.

`times` (*variable=None*)

List of time stamps for variable.

Parameters `variable` – Variable name.

Returns A list of time stamps for the given variable.

1.2.3 Modelica models

To learn the basics of modelling with Modelica, please refer to the online book [Modelica by Example](#).

class `rtctools.optimization.modelica_mixin.ModelicaMixin` (***kwargs*)
Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds a [Modelica](#) model to your optimization problem.

During preprocessing, the Modelica files located inside the `model` subfolder are loaded.

Variables `modelica_library_folders` – Folders in which any referenced Modelica libraries are to be found. Default is an empty list.

1.2.4 CSV I/O

class `rtctools.optimization.csv_mixin.CSVMixin` (***kwargs*)
Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds reading and writing of CSV timeseries and parameters to your optimization problem.

During preprocessing, files named `timeseries_import.csv`, `initial_state.csv`, and `parameters.csv` are read from the input subfolder.

During postprocessing, a file named `timeseries_export.csv` is written to the output subfolder.

In ensemble mode, a file named `ensemble.csv` is read from the input folder. This file contains two columns. The first column gives the name of the ensemble member, and the second column its probability. Furthermore, the other XML files appear one level deeper inside the filesystem hierarchy, inside subfolders with the names of the ensemble members.

Variables

- **csv_delimiter** – Column delimiter used in CSV files. Default is `,`.
- **csv_equidistant** – Whether or not the timeseries data is equidistant. Default is `True`.
- **csv_ensemble_mode** – Whether or not to use ensembles. Default is `False`.
- **csv_validate_timeseries** – Check consistency of timeseries. Default is `True`.

max_timeseries_id (*variable: str*) → `str`
Returns the name of the upper bound timeseries for the specified variable.

Parameters `variable` – Variable name.

min_timeseries_id (*variable: str*) → `str`
Returns the name of the lower bound timeseries for the specified variable.

Parameters `variable` – Variable name.

1.2.5 Delft-FEWS I/O

class `rtctools.optimization.pi_mixin.PIMixin` (***kwargs*)
Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds [Delft-FEWS Published Interface](#) I/O to your optimization problem.

During preprocessing, files named `rtcDataConfig.xml`, `timeseries_import.xml`, `rtcParameterConfig.xml`, and `rtcParameterConfig_Numerical.xml` are read from the input subfolder. `rtcDataConfig.xml` maps tuples of FEWS identifiers, including location and parameter ID, to RTC-Tools time series identifiers.

During postprocessing, a file named `timeseries_export.xml` is written to the output subfolder.

Variables

- **pi_binary_timeseries** – Whether to use PI binary timeseries format. Default is `False`.
- **pi_parameter_config_basenames** – List of parameter config file basenames to read. Default is `[rtcParameterConfig]`.
- **pi_parameter_config_numerical_basename** – Numerical config file basename to read. Default is `rtcParameterConfig_Numerical`.
- **pi_check_for_duplicate_parameters** – Check if duplicate parameters are read. Default is `True`.
- **pi_validate_timeseries** – Check consistency of timeseries. Default is `True`.

max_timeseries_id (*variable: str*) → str

Returns the name of the upper bound timeseries for the specified variable.

Parameters **variable** – Variable name

min_timeseries_id (*variable: str*) → str

Returns the name of the lower bound timeseries for the specified variable.

Parameters **variable** – Variable name

timeseries_export

pi.Timeseries object for holding the output data.

timeseries_import

pi.Timeseries object containing the input data.

timeseries_import_times

List of time stamps for which input data is specified.

The time stamps are in seconds since `t0`, and may be negative.

1.2.6 Bookkeeping of linearization parameters

class `rtctools.optimization.linearization_mixin.LinearizationMixin` (***kwargs*)

Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds linearized equation parameter bookkeeping to your optimization aproblem.

If your model contains linearized equations, this mixin will set the parameters of these equations based on the `t0` value of an associated timeseries.

The mapping between linearization parameters and time series is provided in the `linearization_parameters` method.

linearization_parameters () → Dict[str, str]

Returns A dictionary of parameter names mapping to time series identifiers.

1.2.7 Lookup tables

```
class rtctools.optimization.csv_lookup_table_mixin.LookupTable (inputs:
                                                                    List[casadi.casadi.MX],
                                                                    function:
                                                                    casadi.casadi.Function,
                                                                    tck: Tuple =
                                                                    None)
```

Bases: object

Lookup table.

__call__ (*args) → Union[float, rtctools.optimization.timeseries.Timeseries]
Evaluate the lookup table.

Parameters **args** (Float, iterable of floats, or *Timeseries*) – Input values.

Returns Lookup table evaluated at input values.

Example use:

```
y = lookup_table(1.0)
[y1, y2] = lookup_table([1.0, 2.0])
```

```
class rtctools.optimization.csv_lookup_table_mixin.CSVLookupTableMixin (**kwargs)
Bases: rtctools.optimization.optimization_problem.OptimizationProblem
```

Adds lookup tables to your optimization problem.

During preprocessing, the CSV files located inside the `lookup_tables` subfolder are read. In every CSV file, the first column contains the output of the lookup table. Subsequent columns contain the input variables.

Cubic B-Splines are used to turn the data points into continuous lookup tables.

Optionally, a file `curvefit_options.ini` may be included inside the `lookup_tables` folder. This file contains, grouped per lookup table, the following options:

- **monotonicity:**
 - is an integer, magnitude is ignored
 - if positive, causes spline to be monotonically increasing
 - if negative, causes spline to be monotonically decreasing
 - if 0, leaves spline monotonicity unconstrained
- **curvature:**
 - is an integer, magnitude is ignored
 - if positive, causes spline curvature to be positive (convex)
 - if negative, causes spline curvature to be negative (concave)
 - if 0, leaves spline curvature unconstrained

Note: Currently only one-dimensional lookup tables are fully supported. Support for two- dimensional lookup tables is experimental.

Variables

- **csv_delimiter** – Column delimiter used in CSV files. Default is , .

- **csv_lookup_table_debug** – Whether to generate plots of the spline fits. Default is `false`.
- **csv_lookup_table_debug_points** – Number of evaluation points for plots. Default is 100.

lookup_tables (*ensemble_member*)

Returns a dictionary of lookup tables.

Parameters **ensemble_member** – The ensemble member index.

Returns A dictionary of variable names and lookup tables.

1.2.8 Treatment of nonconvexities using homotopy

Using homotopy, a convex optimization problem can be continuously deformed into a non-convex problem.

class `rtctools.optimization.homotopy_mixin.HomotopyMixin` (***kwargs*)

Bases: `rtctools.optimization.optimization_problem.OptimizationProblem`

Adds homotopy to your optimization problem. A homotopy is a continuous transformation between two optimization problems, parametrized by a single parameter $\theta \in [0, 1]$.

Homotopy may be used to solve non-convex optimization problems, by starting with a convex approximation at $\theta = 0.0$ and ending with the non-convex problem at $\theta = 1.0$.

Note: It is advised to look for convex reformulations of your problem, before resorting to a use of the (potentially expensive) homotopy process.

homotopy_options () \rightarrow Dict[str, Union[str, float]]

Returns a dictionary of options controlling the homotopy process.

Option	Type	Default value
<code>delta_theta_0</code>	float	1.0
<code>delta_theta_min</code>	float	0.01
<code>homotopy_parameter</code>	string	<code>theta</code>

The homotopy process is controlled by the homotopy parameter in the model, specified by the option `homotopy_parameter`. The homotopy parameter is initialized to 0.0, and increases to a value of 1.0 with a dynamically changing step size. This step size is initialized with the value of the option `delta_theta_0`. If this step size is too large, i.e., if the problem with the increased homotopy parameter fails to converge, the step size is halved. The process of halving terminates when the step size falls below the minimum value specified by the option `delta_theta_min`.

Returns A dictionary of homotopy options.

1.2.9 Initial state estimation

class `rtctools.optimization.initial_state_estimation_mixin.InitialStateEstimationMixin` (***kwargs*)

Bases: `rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin`

Adds initial state estimation to your optimization problem *using goal programming*.

Before any other goals are evaluated, first, the deviation between initial state measurements and their respective model states is minimized in the least squares sense (IDVAR, priority -2). Secondly, the distance between pairs

of states is minimized, again in the least squares sense, so that “smooth” initial guesses are provided for states without measurements (priority -1).

Note: There are types of problems where, in addition to minimizing differences between states and measurements, it is advisable to perform a steady-state initialization using additional initial-time model equations. For hydraulic models, for instance, it is often helpful to require that the time-derivative of the flow variables vanishes at the initial time.

initial_state_measurements () → List[Union[Tuple[str, str], Tuple[str, str, float]]]

List of pairs (state, measurement_id) or triples (state, measurement_id, max_deviation), relating states to measurement time series IDs.

The default maximum deviation is 1.0.

initial_state_smoothing_pairs () → List[Union[Tuple[str, str], Tuple[str, str, float]]]

List of pairs (state1, state2) or triples (state1, state2, max_deviation), relating states the distance of which is to be minimized.

The default maximum deviation is 1.0.

1.2.10 Multi-objective optimization

class rtctools.optimization.goal_programming_mixin.Goal

Bases: object

Base class for lexicographic goal programming goals.

A goal is defined by overriding the *function()* method.

Variables

- **function_range** – Range of goal function. *Required if a target is set.*
- **function_nominal** – Nominal value of function. Used for scaling. Default is 1.
- **target_min** – Desired lower bound for goal function. Default is `numpy.nan`.
- **target_max** – Desired upper bound for goal function. Default is `numpy.nan`.
- **priority** – Integer priority of goal. Default is 1.
- **weight** – Optional weighting applied to the goal. Default is 1.0.
- **order** – Penalization order of goal violation. Default is 2.
- **critical** – If `True`, the algorithm will abort if this goal cannot be fully met. Default is `False`.
- **relaxation** – Amount of slack added to the hard constraints related to the goal. Must be a nonnegative value. Default is 0.0.

The target bounds indicate the range within the function should stay, *if possible*. Goals are, in that sense, *soft*, as opposed to standard hard constraints.

Four types of goals can be created:

1. Minimization goal if no target bounds are set:

$$\min f$$

2. Lower bound goal if `target_min` is set:

$$m \leq f$$

3. Upper bound goal if `target_max` is set:

$$f \leq M$$

4. Combined lower and upper bound goal if `target_min` and `target_max` are both set:

$$m \leq f \leq M$$

Lower priority goals take precedence over higher priority goals.

Goals with the same priority are weighted off against each other in a single objective function.

In goals where a target is set:

- The function range interval must be provided as this is used to introduce hard constraints on the value that the function can take. If one is unsure about which value the function can take, it is recommended to overestimate this interval. However, an overestimated interval will negatively influence how accurately the target bounds are met.
- The target provided must be contained in the function range.
- The function nominal is used to scale the constraints.
- If both a `target_min` and a `target_max` are set, the target maximum must be at least equal to minimum one.

In minimization goals:

- The function range is not used and therefore cannot be set.
- The function nominal is used to scale the function value in the objective function. To ensure that all goals are given a similar importance, it is crucial to provide an accurate estimate of this parameter.

The goal violation value is taken to the order'th power in the objective function of the final optimization problem.

Relaxation is used to loosen the constraints that are set after the optimization of the goal's priority. The unit of the relaxation is equal to that of the goal function.

Example definition of the point goal $x(t) \geq 1.1$ for $t = 1.0$ at priority 1:

```
class MyGoal(Goal):
    def function(self, optimization_problem, ensemble_member):
        # State 'x' at time t = 1.0
        t = 1.0
        return optimization_problem.state_at('x', t, ensemble_member)

    function_range = (1.0, 2.0)
    target_min = 1.1
    priority = 1
```

Example definition of the path goal $x(t) \geq 1.1$ for all t at priority 2:

```
class MyPathGoal(Goal):
    def function(self, optimization_problem, ensemble_member):
        # State 'x' at any point in time
        return optimization_problem.state('x')

    function_range = (1.0, 2.0)
    target_min = 1.1
    priority = 2
```

Note that for path goals, the ensemble member index is not passed to the call to `OptimizationProblem.state()`. This call returns a time-independent symbol that is also independent of the active ensemble member. Path goals are applied to all times and all ensemble members simultaneously.

function (*optimization_problem*: `rtctools.optimization.optimization_problem.OptimizationProblem`, *ensemble_member*: `int`) → `casadi.casadi.MX`
 This method returns a CasADi MX object describing the goal function.

Returns A CasADi MX object.

get_function_key (*optimization_problem*: `rtctools.optimization.optimization_problem.OptimizationProblem`, *ensemble_member*: `int`) → `str`
 Returns a key string uniquely identifying the goal function. This is used to eliminate linearly dependent constraints from the optimization problem.

has_target_bounds
 True if the user goal has min/max bounds.

has_target_max
 True if the user goal has max bounds.

has_target_min
 True if the user goal has min bounds.

class `rtctools.optimization.goal_programming_mixin.StateGoal` (*optimization_problem*)
 Bases: `rtctools.optimization.goal_programming_mixin.Goal`

Base class for lexicographic goal programming path goals that act on a single model state.

A state goal is defined by setting at least the `state` class variable.

Variables

- **state** – State on which the goal acts. *Required.*
- **target_min** – Desired lower bound for goal function. Default is `numpy.nan`.
- **target_max** – Desired upper bound for goal function. Default is `numpy.nan`.
- **priority** – Integer priority of goal. Default is 1.
- **weight** – Optional weighting applied to the goal. Default is 1.0.
- **order** – Penalization order of goal violation. Default is 2.
- **critical** – If True, the algorithm will abort if this goal cannot be fully met. Default is False.

Example definition of the goal $x(t) \geq 1.1$ for all t at priority 2:

```
class MyStateGoal(StateGoal):
    state = 'x'
    target_min = 1.1
    priority = 2
```


Contrary to ordinary Goal objects, PathGoal objects need to be initialized with an OptimizationProblem instance to allow extraction of state metadata, such as bounds and nominal values. Consequently, state goals must be instantiated as follows:

```
my_state_goal = MyStateGoal(optimization_problem)
```

Note that StateGoal is a helper class. State goals can also be defined using Goal as direct base class, by implementing the function method and providing the function_range and function_nominal class variables manually.

__init__(*optimization_problem*)
Initialize the state goal object.

Parameters *optimization_problem* – OptimizationProblem instance.

class rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin(**kwargs)
Bases: *rtctools.optimization.optimization_problem.OptimizationProblem*

Adds lexicographic goal programming to your optimization problem.

goal_programming_options() → Dict[str, Union[float, bool]]
Returns a dictionary of options controlling the goal programming process.

Option	Type	Default value
violation_relaxation	float	0.0
constraint_relaxation	float	0.0
mu_reinit	bool	True
fix_minimized_values	bool	True/False
check_monotonicity	bool	True
equality_threshold	float	1e-8
interior_distance	float	1e-6
scale_by_problem_size	bool	False
keep_soft_constraints	bool	False

Before turning a soft constraint of the goal programming algorithm into a hard constraint, the violation variable (also known as epsilon) of each goal is relaxed with the *violation_relaxation*. Use of this option is normally not required.

When turning a soft constraint of the goal programming algorithm into a hard constraint, the constraint is relaxed with *constraint_relaxation*. Use of this option is normally not required. Note that:

1. Minimization goals do not get *constraint_relaxation* applied when *fix_minimized_values* is True.
2. Because of the constraints it generates, when *keep_soft_constraints* is True, the option *fix_minimized_values* needs to be set to False for the *constraint_relaxation* to be applied at all.

A goal is considered to be violated if the violation, scaled between 0 and 1, is greater than the specified tolerance. Violated goals are fixed. Use of this option is normally not required.

When using the default solver (IPOPT), its barrier parameter *mu* is normally re-initialized a every iteration of the goal programming algorithm, unless *mu_reinit* is set to False. Use of this option is normally not required.

If *fix_minimized_values* is set to True, goal functions will be set to equal their optimized values in optimization problems generated during subsequent priorities. Otherwise, only an upper bound will be set. Use of this option is normally not required. Note that a non-zero goal relaxation overrules this option; a non-zero relaxation will always result in only an upper bound being set. Also note that the use of this

option may add non-convex constraints to the optimization problem. The default value for this parameter is `True` for the default solvers IPOPT/BONMIN. If any other solver is used, the default value is `False`.

If `check_monotonicity` is set to `True`, then it will be checked whether goals with the same function key form a monotonically decreasing sequence with regards to the target interval.

The option `equality_threshold` controls when a two-sided inequality constraint is folded into an equality constraint.

The option `interior_distance` controls the distance from the scaled target bounds, starting from which the function value is considered to lie in the interior of the target space.

If `scale_by_problem_size` is set to `True`, the objective (i.e. the sum of the violation variables) will be divided by the number of goals, and the path objective will be divided by the number of path goals and the number of time steps. This will make sure the objectives are always in the range `[0, 1]`, at the cost of solving each goal/time step less accurately.

The option `keep_soft_constraints` controls how the epsilon variables introduced in the target goals are dealt with in subsequent priorities. If `keep_soft_constraints` is set to `False`, each epsilon is replaced by its computed value and those are used to derive a new set of constraints. If `keep_soft_constraints` is set to `True`, the epsilons are kept as variables and the constraints are not modified. To ensure the goal programming philosophy, i.e., Pareto optimality, a single constraint is added to enforce that the objective function must always be at most the objective value. This method allows for a larger solution space, at the cost of having a (possibly) more complex optimization problem. Indeed, more variables are kept around throughout the optimization and any objective function is turned into a constraint for the subsequent priorities (while in the `False` option this was the case only for the function of minimization goals).

Returns A dictionary of goal programming options.

goals () → List[rtctools.optimization.goal_programming_mixin.Goal]

User problem returns list of *Goal* objects.

Returns A list of goals.

path_goals () → List[rtctools.optimization.goal_programming_mixin.Goal]

User problem returns list of path *Goal* objects.

Returns A list of path goals.

priority_completed (priority: int) → None

Called after optimization for goals of certain priority is completed.

Parameters **priority** – The priority level that was completed.

priority_started (priority: int) → None

Called when optimization for goals of certain priority is started.

Parameters **priority** – The priority level that was started.

1.2.11 Forecast uncertainty

class rtctools.optimization.control_tree_mixin.**ControlTreeMixin** (**kwargs)
 Bases: *rtctools.optimization.optimization_problem.OptimizationProblem*

Adds a stochastic control tree to your optimization problem.

control_tree_options () → Dict[str, Union[List[str], List[float], int]]

Returns a dictionary of options controlling the creation of a k-ary stochastic tree.

Option	Type	Default value
forecast_variables	list of strings	All constant inputs
branching_times	list of floats	self.times()
k	int	2

A k-ary tree is generated, branching at every interior branching time. Ensemble members are clustered to paths through the tree based on average distance over all forecast variables.

Returns A dictionary of control tree generation options.

1.3 Simulation

Note: For a simulation example, see *Simulation examples*

Contents:

1.3.1 Basics

class rtctools.simulation.simulation_problem.**SimulationProblem**(**kwargs)

Bases: object

Implements the **BMI** Interface.

Base class for all Simulation problems. Loads the Modelica Model.

Variables **modelica_library_folders** – Folders containing any referenced Modelica libraries. Default is an empty list.

get_current_time()

Return current time of simulation.

Returns The current simulation time.

get_end_time()

Return end time of experiment.

Returns The end time of the experiment.

get_start_time()

Return start time of experiment.

Returns The start time of the experiment.

get_var(name)

Return a numpy array from FMU.

Parameters **name** – Variable name.

Returns The value of the variable.

get_var_count()

Return the number of variables in the model.

Returns The number of variables in the model.

get_var_name(i)

Returns the name of a variable.

Parameters *i* – Index in ordered dictionary returned by method `get_variables`.

Returns The name of the variable.

get_var_rank (*name*)

Not implemented

get_var_shape (*name*)

Not implemented

get_var_type (*name*)

Return type, compatible with numpy.

Parameters *name* – String variable name.

Returns The numpy-compatible type of the variable.

Raises `KeyError`

get_variables ()

Return all variables (both internal and user defined)

Returns An ordered dictionary of all variables supported by the model.

initialize (*config_file=None*)

Initialize state vector with default values

Parameters *config_file* – Path to an initialization file.

post ()

Any postprocessing takes place here.

pre ()

Any preprocessing takes place here.

reset ()

Reset the FMU.

setup_experiment (*start, stop, dt*)

Method for subclasses (PIMixin, CSVMixin, or user classes) to set timing information for a simulation run.

Parameters

- **start** – Start time for the simulation.
- **stop** – Final time for the simulation.
- **dt** – Time step size.

simulate ()

Run model from *start_time* to *end_time*.

update (*dt*)

Performs one timestep.

The methods `setup_experiment` and `initialize` must have been called before.

Parameters *dt* – Time step size.

`rtctools.util.run_simulation_problem` (*simulation_problem_class*, *base_folder='..'*,
log_level=20)

Sets up and runs a simulation problem.

Parameters

- **simulation_problem_class** – Optimization problem class to solve.

- **base_folder** – Folder within which subfolders “input”, “output”, and “model” exist, containing input and output data, and the model, respectively.
- **log_level** – The log level to use.

Returns `SimulationProblem` instance.

1.3.2 CSV I/O

class `rtctools.simulation.csv_mixin.CSVMixin` (***kwargs*)

Bases: `rtctools.simulation.simulation_problem.SimulationProblem`

Adds reading and writing of CSV timeseries and parameters to your simulation problem.

During preprocessing, files named `timeseries_import.csv`, `initial_state.csv`, and `parameters.csv` are read from the input subfolder.

During postprocessing, a file named `timeseries_export.csv` is written to the output subfolder.

Variables

- **csv_delimiter** – Column delimiter used in CSV files. Default is `,`.
- **csv_validate_timeseries** – Check consistency of timeseries. Default is `True`.

timeseries_at (*variable, t*)

Return the value of a timeseries at the given time.

Parameters

- **variable** – Variable name.
- **t** – Time.

Returns The interpolated value of the time series.

Raises `KeyError`

1.3.3 Delft-FEWS I/O

class `rtctools.simulation.pi_mixin.PIMixin` (***kwargs*)

Bases: `rtctools.simulation.simulation_problem.SimulationProblem`

Adds Delft-FEWS Published Interface I/O to your simulation problem.

During preprocessing, files named `rtcDataConfig.xml`, `timeseries_import.xml`, and “`rtcParameterConfig.xml`” are read from the input subfolder. `rtcDataConfig.xml` maps tuples of FEWS identifiers, including location and parameter ID, to RTC-Tools time series identifiers.

During postprocessing, a file named `timeseries_export.xml` is written to the output subfolder.

Variables

- **pi_binary_timeseries** – Whether to use PI binary timeseries format. Default is `False`.
- **pi_parameter_config_basenames** – List of parameter config file basenames to read. Default is `[rtcParameterConfig]`.
- **pi_check_for_duplicate_parameters** – Check if duplicate parameters are read. Default is `True`.
- **pi_validate_timeseries** – Check consistency of timeseries. Default is `True`.

timeseries_at (*variable*, *t*)

Return the value of a time series at the given time.

Parameters

- **variable** – Variable name.
- **t** – Time.

Returns The interpolated value of the time series.

Raises KeyError

1.4 Examples

This section provides examples demonstrating key features of RTC-Tools.

1.4.1 Optimization examples

This section provides examples demonstrating key features of RTC-Tools optimization.

Filling a Reservoir



Overview

The purpose of this example is to understand the technical setup of an RTC-Tools model, how to run the model, and how to interpret the results.

The scenario is the following: A reservoir operator is trying to fill a reservoir. They are given a six-day forecast of inflows given in 12-hour increments. The operator wants to save as much of the inflows as possible, but does not want to end up with too much water at the end of the six days. They have chosen to use RTC-Tools to calculate how much water to release and when to release it.

If you installed using source, the library and examples directory are available in the git repositories. If you installed using pip directly, you first need to download/copy the examples and libraries to a convenient location. See [Downloading and running examples](#) and [Copying Modelica libraries](#) for detailed instructions.

The folder `<examples directory>\basic` contains a complete RTC-Tools optimization problem. An RTC-Tools directory has the following structure:

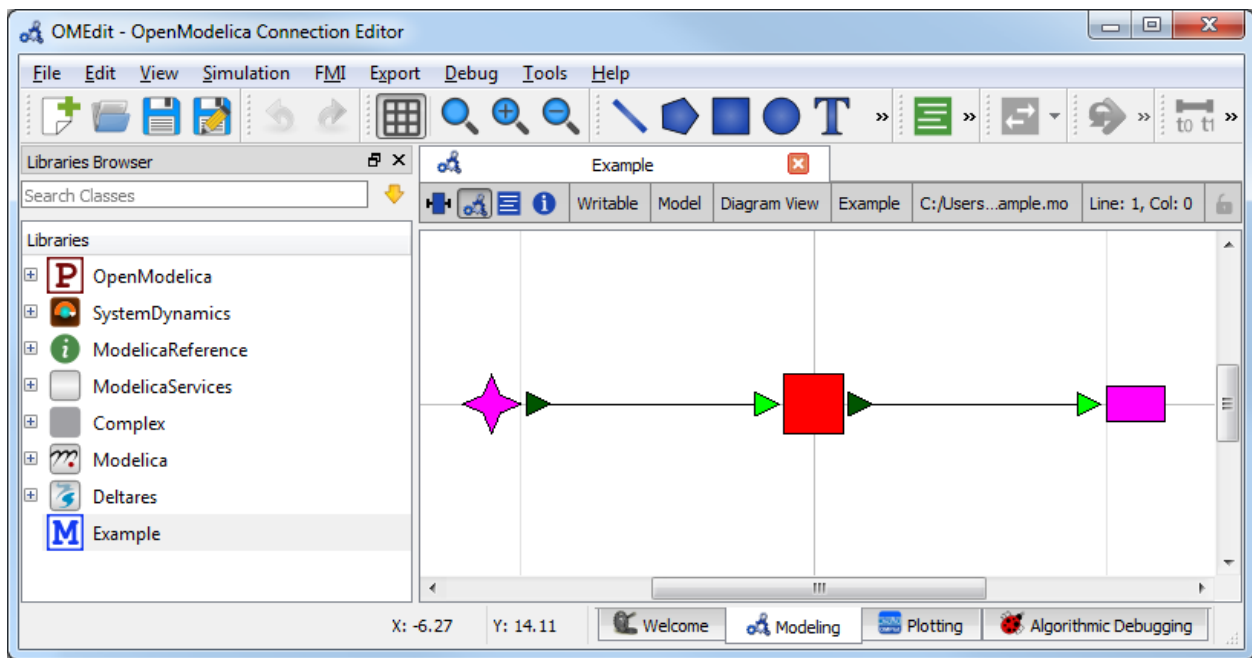
- `input`: This folder contains the model input data. These are several files in comma separated value format, `csv`.
- `model`: This folder contains the Modelica model. The Modelica model contains the physics of the RTC-Tools model.
- `output`: The folder where the output is saved in the file `timeseries_export.csv`.
- `src`: This folder contains a Python file. This file contains the configuration of the model and is used to run the model.

The Model

The first step is to develop a physical model of the system. The model can be viewed and edited using the OpenModelica Connection Editor (OMEdit) program. For how to download and start up OMEdit, see [Getting OMEdit](#).

1. Load the Deltares library into OMEdit
 - Using the menu bar: *File -> Open Model/Library File(s)*
 - Select `<library directory>\Deltares\package.mo`
2. Load the example model into OMEdit
 - Using the menu bar: *File -> Open Model/Library File(s)*
 - Select `<examples directory>\basic\model\Example.mo`

Once loaded, we have an OpenModelica Connection Editor window that looks like this:



The model `Example.mo` represents a simple system with the following elements:

- a reservoir, modeled as storage element `Deltares.ChannelFlow.SimpleRouting.Storage.Storage`,
- an inflow boundary condition `Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow`,
- an outfall boundary condition `Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Terminal`,
- connectors (black lines) connecting the elements.

You can use the mouse-over feature help to identify the predefined models from the Deltares library. You can also drag the elements around- the connectors will move with the elements. Adding new elements is easy- just drag them in from the Deltares Library on the sidebar. Connecting the elements is just as easy- click and drag between the ports on the elements.

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

```

1 model Example
2   Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow inflow;
3   Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(V(nominal=4e5, min=2e5,
4   ↪max=6e5));
5   Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Terminal outfall;
6   input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
7   input Modelica.SIunits.VolumeFlowRate Q_release(fixed = false, min = 0.0, max = 6.
8   ↪5);
9   output Modelica.SIunits.Volume V_storage;
10  equation
11    connect (inflow.QOut, storage.QIn);
12    connect (storage.QOut, outfall.QIn);
13    storage.Q_release = Q_release;
14    inflow.Q = Q_in;
15    V_storage = storage.V;
16 end Example;
```

The three water system elements (storage, inflow, and outfall) appear under the `model Example` statement. The `equation` part connects these three elements with the help of connections. Note that `storage` extends the partial model `QISO` which contains the connectors `QIn` and `QOut`. With `QISO`, `storage` can be connected on two sides. The `storage` element also has a variable `Q_release`, which is the decision variable the operator controls.

OpenModelica Connection Editor will automatically generate the element and connector entries in the text file. Defining inputs and outputs requires editing the text file directly. Relationships between the inputs and outputs and the library elements must also be defined in the `equation` section.

In addition to elements, the input variables `Q_in` and `Q_release` are also defined. `Q_in` is determined by the forecast and the operator cannot control it, so we set `Q_in(fixed = true)`. The actual values of `Q_in` are stored in `timeseries_import.csv`. In the `equation` section, equations are defined to relate the inputs to the appropriate water system elements.

Because we want to view the water volume in the storage element in the output file, we also define an output variable `V_storage`.

The Optimization Problem

The python script is created and edited in a text editor. In general, the python script consists of the following blocks:

- Import of packages
- Definition of the optimization problem class

- Constructor
 - Objective function
 - Definition of constraints
 - Any additional configuration
- A run statement

Importing Packages

Packages are imported using `from ... import ...` at the top of the file. In our script, we import the classes we want the class to inherit, the package `run_optimization_problem` from the `rtctools.util` package, and any extra packages we want to use. For this example, the import block looks like:

```
1 from rtctools.optimization.collocated_integrated_optimization_problem \
2     import CollocatedIntegratedOptimizationProblem
3 from rtctools.optimization.csv_mixin import CSVMixin
4 from rtctools.optimization.modelica_mixin import ModelicaMixin
5 from rtctools.util import run_optimization_problem
```

Optimization Problem

The next step is to define the optimization problem class. We construct the class by declaring the class and inheriting the desired parent classes. The parent classes each perform different tasks related to importing and exporting data and solving the optimization problem. Each imported class makes a set of methods available to the our optimization class.

```
8 class Example(CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
```

Next, we define an objective function. This is a class method that returns the value that needs to be minimized.

```
12 def objective(self, ensemble_member):
13     # Minimize water pumped. The total water pumped is the integral of the
14     # water pumped from the starting time until the stoping time. In
15     # practice, self.integral() is a summation of all the discrete states.
16     return self.integral('Q_release', ensemble_member)
```

Constraints can be declared by declaring the `path_constraints()` method. Path constraints are constraints that are applied every timestep. To set a constraint at an individual timestep, we could define it inside the `constraints()` method.

Other parent classes also declare this method, so we call the `super()` method so that we don't overwrite their behaviour.

```
18 def path_constraints(self, ensemble_member):
19     # Call super() class to not overwrite default behaviour
20     constraints = super().path_constraints(ensemble_member)
21     # Constrain the volume of storage between 380000 and 420000 m^3
22     constraints.append((self.state('storage.V'), 380000, 420000))
23     return constraints
```

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.

```
27 run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

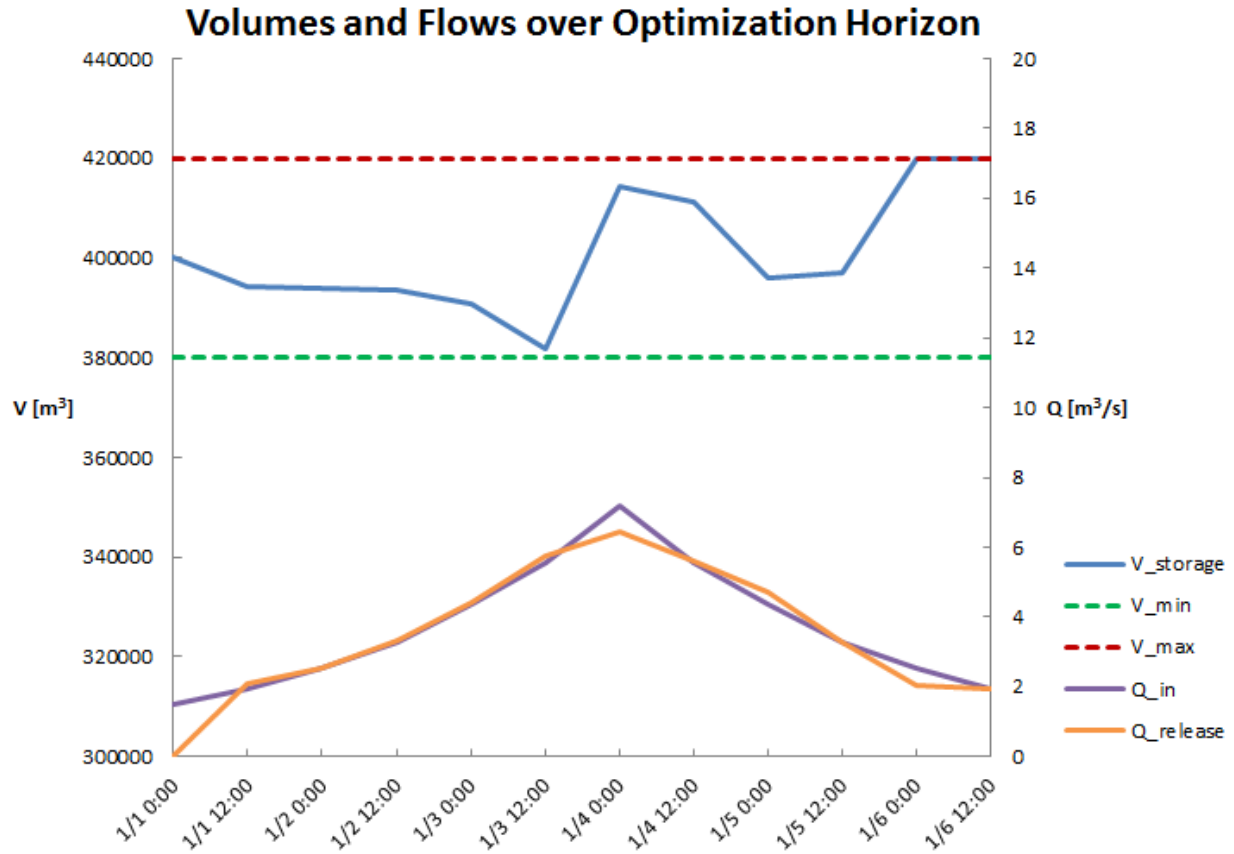
```
1 from rtctools.optimization.collocated_integrated_optimization_problem \
2     import CollocatedIntegratedOptimizationProblem
3 from rtctools.optimization.csv_mixin import CSVMixin
4 from rtctools.optimization.modelica_mixin import ModelicaMixin
5 from rtctools.util import run_optimization_problem
6
7
8 class Example(CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
9     """
10     A basic example for introducing users to RTC-Tools 2
11     """
12     def objective(self, ensemble_member):
13         # Minimize water pumped. The total water pumped is the integral of the
14         # water pumped from the starting time until the stoping time. In
15         # practice, self.integral() is a summation of all the discrete states.
16         return self.integral('Q_release', ensemble_member)
17
18     def path_constraints(self, ensemble_member):
19         # Call super() class to not overwrite default behaviour
20         constraints = super().path_constraints(ensemble_member)
21         # Constrain the volume of storage between 380000 and 420000 m^3
22         constraints.append((self.state('storage.V'), 380000, 420000))
23         return constraints
24
25
26 # Run
27 run_optimization_problem(Example)
```

Running RTC-Tools

To run this basic example in RTC-Tools, navigate to the basic example `src` directory in the RTC-Tools shell and run the example using `python example.py`. For more details about using RTC-Tools, see [Running RTC-Tools](#).

Extracting Results

The results from the run are found in `output\timeseries_export.csv`. Any CSV-reading software can import it, but this is what the results look like when plotted in Microsoft Excel:



This plot shows that the operator is able to keep the water level within the bounds over the entire time horizon and end with a full reservoir.

Feel free to experiment with this example. See what happens if you change the max of Q_{release} (in the Modelica file) or if you make the objective function negative (in the python script).

Mixed Integer Optimization: Pumps and Orifices



Note: This example focuses on how to incorporate mixed integer components into a hydraulic model, and assumes basic exposure to RTC-Tools. To start with basics, see *Filling a Reservoir*.

Note: By default, if you define any integer or boolean variables in the model, RTC-Tools will switch from IPOPT to BONMIN. You can modify solver options by overriding the `solver_options()` method. Refer to CasADi's nlpso interface for a list of supported solvers.

The Model

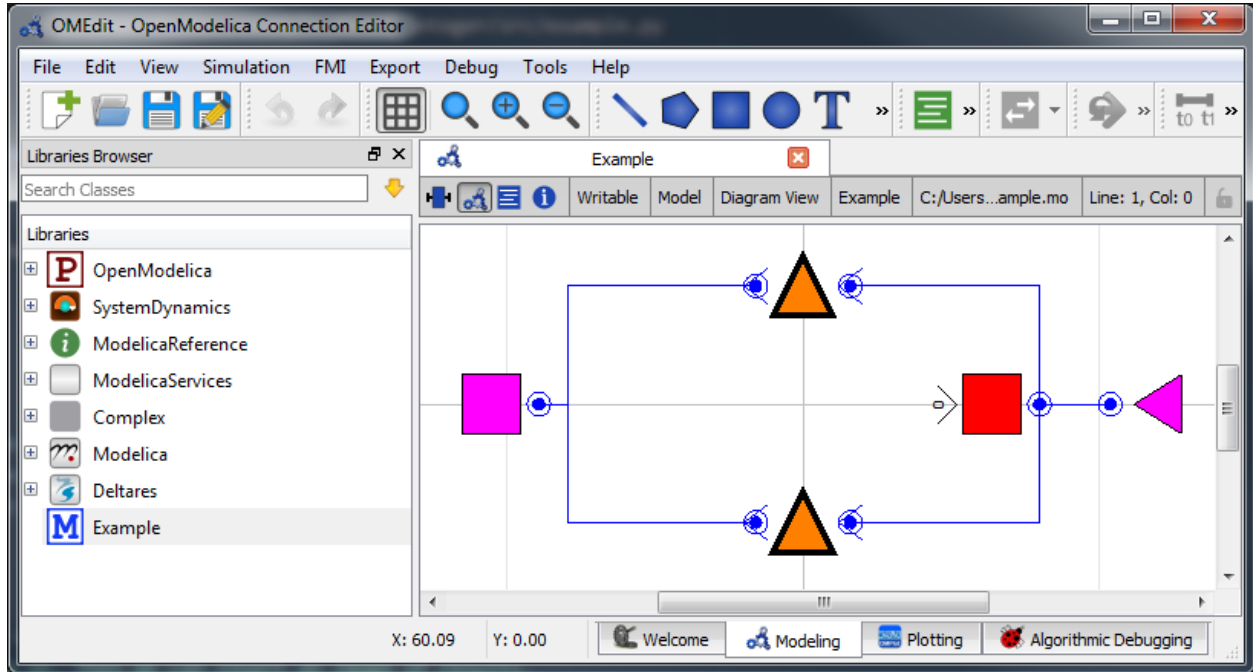
For this example, the model represents a typical setup for the dewatering of lowland areas. Water is routed from the hinterland (modeled as discharge boundary condition, right side) through a canal (modeled as storage element) towards the sea (modeled as water level boundary condition on the left side). 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 free, while 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 whenever possible and activates the pump only when necessary.

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 directory>\mixed_integer\model\Example.mo`. The model `Example.mo` represents a simple water system with the following elements:

- a canal segment, 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 pump `Deltares.ChannelFlow.Hydraulic.Structures.Pump`
- an orifice modeled as a pump `Deltares.ChannelFlow.Hydraulic.Structures.Pump`



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

```

1  model Example
2    // Declare Model Elements
3    Deltares.ChannelFlow.Hydraulic.Storage.Linear storage(A=1.0e6, H_b=0.0, HQ.H(min=0.
4    ↪0, max=0.5));
5    Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge discharge;
6    Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level level;
7    Deltares.ChannelFlow.Hydraulic.Structures.Pump pump;
8    Deltares.ChannelFlow.Hydraulic.Structures.Pump orifice;
9
10   // Define Input/Output Variables and set them equal to model variables
11   input Modelica.SIunits.VolumeFlowRate Q_pump(fixed=false, min=0.0, max=7.0) = pump.
12   ↪Q;
13   input Boolean is_downhill;
14   input Modelica.SIunits.VolumeFlowRate Q_in(fixed=true) = discharge.Q;
15   input Modelica.SIunits.Position H_sea(fixed=true) = level.H;
16   input Modelica.SIunits.VolumeFlowRate Q_orifice(fixed=false, min=0.0, max=10.0) =
17   ↪orifice.Q;
18   output Modelica.SIunits.Position storage_level = storage.HQ.H;
19   output Modelica.SIunits.Position sea_level = level.H;
20 equation
21   // Connect Model Elements
22   connect(orifice.HQDown, level.HQ);
23   connect(storage.HQ, orifice.HQUp);
24   connect(storage.HQ, pump.HQUp);
25   connect(discharge.HQ, storage.HQ);
26   connect(pump.HQDown, level.HQ);
27 end Example;

```

The five water system elements (storage, discharge boundary condition, water level boundary condition, pump, and orifice) appear under the `model Example` statement. 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 hydraulic sense here), so it can be connected to one other element only. It extends the `HQOnePort` which again inherits from the connector `HQPort`.

In addition to elements, the input variables `Q_in`, `H_sea`, `Q_pump`, and `Q_orifice` are also defined. Because we want to view the water levels in the storage element in the output file, we also define output variables `storage_level` and `sea_level`. It is usually easiest to set input and output variables equal to their corresponding model variable in the same line.

To maintain the linearity of the model, we input the Boolean `is_downhill` as a way to keep track of whether water can flow by gravity to the sea. This variable is not used directly in the hydraulics, but we use it later in the constraints in the python file.

The Optimization Problem

The python script consists of the following blocks:

- Import of packages
- Definition of the optimization problem class
 - Constructor
 - Objective function
 - Definition of constraints
 - Additional configuration of the solver
- A run statement

Importing Packages

For this example, the import block is as follows:

```
1 import numpy as np
2
3 from rtctools.optimization.collocated_integrated_optimization_problem \
4     import CollocatedIntegratedOptimizationProblem
5 from rtctools.optimization.csv_mixin import CSVMixin
6 from rtctools.optimization.modelica_mixin import ModelicaMixin
```

Note that we are also importing `inf` from `numpy`. We will use this later in the constraints.

Optimization Problem

Next, we construct the class by declaring it and inheriting the desired parent classes.

```
10 class Example(CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
```

Now we define an objective function. This is a class method that returns the value that needs to be minimized. Here we specify that we want to minimize the volume pumped:

```

18 def objective(self, ensemble_member):
19     # Minimize water pumped. The total water pumped is the integral of the
20     # water pumped from the starting time until the stopping time. In
21     # practice, self.integral() is a summation of all the discrete states.
22     return self.integral('Q_pump', ensemble_member)

```

Constraints can be declared by declaring the `path_constraints()` method. Path constraints are constraints that are applied every timestep. To set a constraint at an individual timestep, define it inside the `constraints` method.

The orifice `BooleanSubmergedOrifice` requires special constraints to be set in order to work. They are implemented below in the `path_constraints()` method. their parent classes also declare this method, so we call the `super()` method so that we don't overwrite their behaviour.

```

26 def path_constraints(self, ensemble_member):
27     # Call super to get default constraints
28     constraints = super().path_constraints(ensemble_member)
29     M = 2 # The so-called "big-M"
30
31     # Release through orifice downhill only. This constraint enforces the
32     # fact that water only flows downhill.
33     constraints.append(
34         (self.state('Q_orifice') + (1 - self.state('is_downhill')) * 10,
35          0.0, 10.0))
36
37     # Make sure is_downhill is true only when the sea is lower than the
38     # water level in the storage.
39     constraints.append((self.state('H_sea') - self.state('storage.HQ.H') -
40                        (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
41     constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
42                        self.state('is_downhill') * M, 0.0, np.inf))
43
44     # Orifice flow constraint. Uses the equation:
45     #  $Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ 0.5$ 
46     # Note that this equation is only valid for orifices that are submerged
47     # units: description:
48     w = 3.0 # m width of orifice
49     d = 0.8 # m hight of orifice
50     C = 1.0 # none orifice constant
51     g = 9.8 # m/s^2 gravitational acceleration
52     constraints.append(
53         (((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) +
54          self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -
55          M * (1 - self.state('is_downhill')),
56          -np.inf, 0.0))
57
58     return constraints

```

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

```

61 def solver_options(self):
62     options = super().solver_options()
63     # Restrict solver output
64     solver = options['solver']
65     options[solver]['print_level'] = 1
66     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.

```
70 run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```
1 import numpy as np
2
3 from rtctools.optimization.collocated_integrated_optimization_problem \
4     import CollocatedIntegratedOptimizationProblem
5 from rtctools.optimization.csv_mixin import CSVMixin
6 from rtctools.optimization.modelica_mixin import ModelicaMixin
7 from rtctools.util import run_optimization_problem
8
9
10 class Example(CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
11     """
12     This class is the optimization problem for the Example. Within this class,
13     the objective, constraints and other options are defined.
14     """
15
16     # This is a method that returns an expression for the objective function.
17     # RTC-Tools always minimizes the objective.
18     def objective(self, ensemble_member):
19         # Minimize water pumped. The total water pumped is the integral of the
20         # water pumped from the starting time until the stoping time. In
21         # practice, self.integral() is a summation of all the discrete states.
22         return self.integral('Q_pump', ensemble_member)
23
24     # A path constraint is a constraint where the values in the constraint are a
25     # Timeseries rather than a single number.
26     def path_constraints(self, ensemble_member):
27         # Call super to get default constraints
28         constraints = super().path_constraints(ensemble_member)
29         M = 2 # The so-called "big-M"
30
31         # Release through orifice downhill only. This constraint enforces the
32         # fact that water only flows downhill.
33         constraints.append(
34             (self.state('Q_orifice') + (1 - self.state('is_downhill')) * 10,
35              0.0, 10.0))
36
37         # Make sure is_downhill is true only when the sea is lower than the
38         # water level in the storage.
39         constraints.append((self.state('H_sea') - self.state('storage.HQ.H') -
40                             (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
41         constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
42                             self.state('is_downhill') * M, 0.0, np.inf))
43
44         # Orifice flow constraint. Uses the equation:
45         #  $Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ {0.5}$ 
```

(continues on next page)

(continued from previous page)

```

46     # Note that this equation is only valid for orifices that are submerged
47     #         units:  description:
48     w = 3.0 # m        width of orifice
49     d = 0.8 # m        hight of orifice
50     C = 1.0 # none     orifice constant
51     g = 9.8 # m/s^2    gravitational acceleration
52     constraints.append(
53         (((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) +
54          self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -
55          M * (1 - self.state('is_downhill'))),
56         -np.inf, 0.0))
57
58     return constraints
59
60     # Any solver options can be set here
61     def solver_options(self):
62         options = super().solver_options()
63         # Restrict solver output
64         solver = options['solver']
65         options[solver]['print_level'] = 1
66         return options
67
68
69     # Run
70     run_optimization_problem(Example)

```

Running the Optimization Problem

Note: An explanation of bonmin behaviour and output goes here.

Extracting Results

The results from the run are found in `output/timeseries_export.csv`. Any CSV-reading software can import it, but this is how results can be plotted using the python library matplotlib:

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime

data_path = '../..../examples/mixed_integer/output/timeseries_export.csv'
delimiter = ','

# Import Data
ncols = len(np.genfromtxt(data_path, max_rows=1, delimiter=delimiter))
datefunc = lambda x: datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
results = np.genfromtxt(data_path, converters={0: datefunc}, delimiter=delimiter,
                        dtype='object' + ',float' * (ncols - 1), names=True,
                        encoding=None) [1:]

```

(continues on next page)

(continued from previous page)

```

# Generate Plot
f, axarr = plt.subplots(2, sharex=True)
axarr[0].set_title('Water Level and Discharge')

# Upper subplot
axarr[0].set_ylabel('Water Level [m]')
axarr[0].plot(results['time'], results['storage_level'], label='Storage',
              linewidth=2, color='b')
axarr[0].plot(results['time'], results['sea_level'], label='Sea',
              linewidth=2, color='m')
axarr[0].plot(results['time'], 0.5 * np.ones_like(results['time']), label='Storage Max
→',
              linewidth=2, color='r', linestyle='--')

# Lower Subplot
axarr[1].set_ylabel('Flow Rate [m³/s]')
axarr[1].plot(results['time'], results['Q_orifice'], label='Orifice',
              linewidth=2, color='g')
axarr[1].plot(results['time'], results['Q_pump'], label='Pump',
              linewidth=2, color='r')
axarr[1].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 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)

plt.autoscale(enable=True, axis='x', tight=True)

# Output Plot
plt.show()

```

Observations

Note that in the results plotted above, the pump runs with a constantly varying throughput. To smooth out the flow through the pump, consider using goal programming to apply a path goal minimizing the derivative of the pump at each timestep. For an example, see the third goal in [Declaring Goals](#).

Goal Programming: Defining Multiple Objectives

Note: This example focuses on how to implement multi-objective optimization in RTC-Tools using Goal Programming. It assumes basic exposure to RTC-Tools. If you are a first-time user of RTC-Tools, see [Filling a Reservoir](#).

Goal programming is a way to satisfy (sometimes conflicting) goals by ranking the goals by priority. The optimization algorithm will attempt to optimize each goal one at a time, starting with the goal with the highest priority and moving down through the list. Even if a goal cannot be satisfied, the goal programming algorithm will move on when it has found the best possible answer. Goals can be roughly divided into two types:

- As long as we satisfy the goal, we do not care by how much. If we cannot satisfy a goal, any lower priority goals are not allowed to increase the amount by which we exceed (which is equivalent to not allowing any change at

all to the exceedance).

- We try to achieve as low a value as possible. Any lower priority goals are not allowed to result in an increase of this value (which is equivalent to not allowing any change at all).

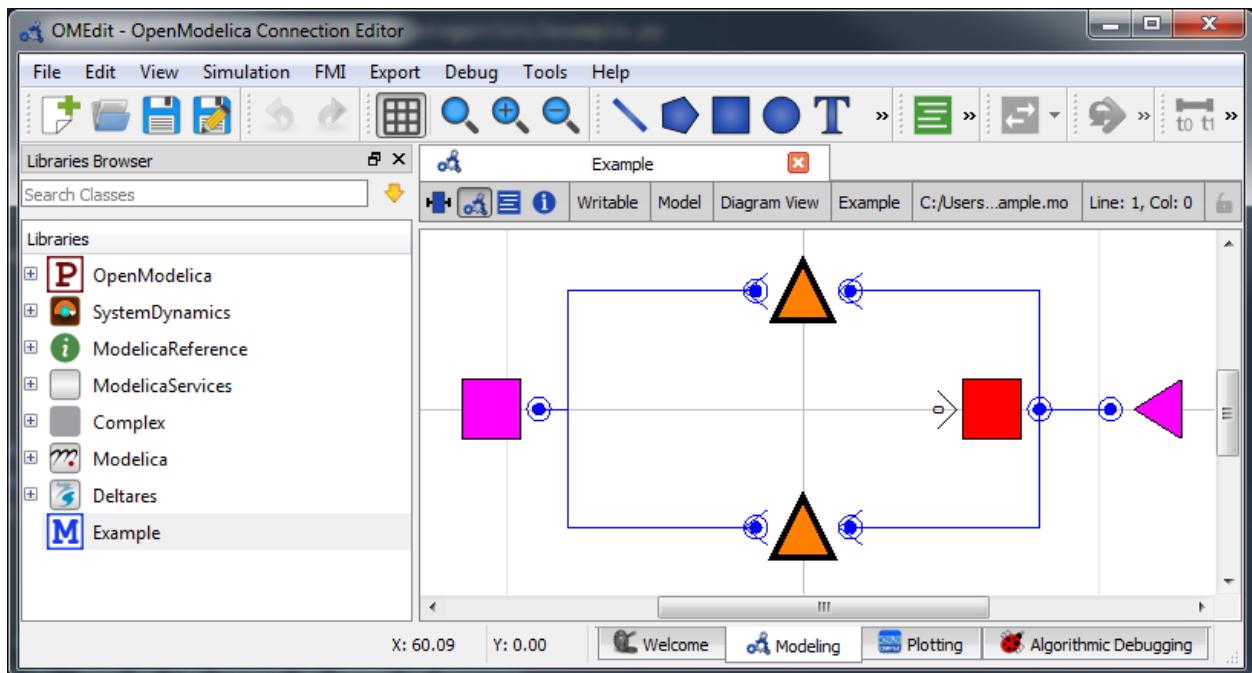
In this example, we will be specifying two goals, one for each type. The higher priority goal will be to maintain the water level of the storage element between two levels. The lower priority goal will be to minimize the total volume pumped.

The Model

Note: This example uses the same hydraulic model as the MILP example. For a detailed explanation of the hydraulic model, including how to formulate mixed integers in your model, see *Mixed Integer Optimization: Pumps and Orifices*.

For this example, 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.

In OpenModelica Connection Editor, the model looks like this:



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

```
1 model Example
2   // Declare Model Elements
3   Deltares.ChannelFlow.Hydraulic.Storage.Linear storage(A=1.0e6, H_b=0.0, HQ.H(min=0.
4     ↳0, max=0.5));
5   Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge discharge;
```

(continues on next page)

(continued from previous page)

```

5   Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level level;
6   Deltares.ChannelFlow.Hydraulic.Structures.Pump pump;
7   Deltares.ChannelFlow.Hydraulic.Structures.Pump orifice;
8
9   // Define Input/Output Variables and set them equal to model variables
10  input Modelica.SIunits.VolumeFlowRate Q_pump(fixed=false, min=0.0, max=7.0) = pump.
    ↳Q;
11  input Boolean is_downhill;
12  input Modelica.SIunits.VolumeFlowRate Q_in(fixed=true) = discharge.Q;
13  input Modelica.SIunits.Position H_sea(fixed=true) = level.H;
14  input Modelica.SIunits.VolumeFlowRate Q_orifice(fixed=false, min=0.0, max=10.0) =
    ↳orifice.Q;
15  output Modelica.SIunits.Position storage_level = storage.HQ.H;
16  output Modelica.SIunits.Position sea_level = level.H;
17  equation
18    // Connect Model Elements
19    connect(orifice.HQDown, level.HQ);
20    connect(storage.HQ, orifice.HQUp);
21    connect(storage.HQ, pump.HQUp);
22    connect(discharge.HQ, storage.HQ);
23    connect(pump.HQDown, level.HQ);
24  end Example;

```

The Optimization Problem

When using goal programming, the python script consists of the following blocks:

- Import of packages
- Declaration of Goals
- Declaration of the optimization problem class
 - Constructor
 - Declaration of constraint methods
 - Specification of Goals
 - Declaration of a `priority_completed()` method
 - Declaration of a `pre()` method
 - Declaration of a `post()` method
 - Additional configuration of the solver
- A run statement

Importing Packages

For this example, the import block is as follows:

```

1  import numpy as np
2
3  from rtctools.optimization.collocated_integrated_optimization_problem \
4    import CollocatedIntegratedOptimizationProblem

```

(continues on next page)

(continued from previous page)

```

5 from rtctools.optimization.csv_mixin import CSVMixin
6 from rtctools.optimization.goal_programming_mixin \
7     import Goal, GoalProgrammingMixin, StateGoal
8 from rtctools.optimization.modelica_mixin import ModelicaMixin

```

Declaring Goals

Goals are defined as classes that inherit the `Goal` parent class. The components of goals can be found in *Multi-objective optimization*. In this example, we demonstrate three ways to define a goal in RTC-Tools.

First, we have a high priority goal to keep the water level within a minimum and maximum. Since we are applying this goal to a specific state (model variable) in our model at every time step, we can inherit a special helper class to define this goal, called a `StateGoal`:

```

12 class WaterLevelRangeGoal(StateGoal):
13     # Applying a state goal to every time step is easily done by defining a goal
14     # that inherits StateGoal. StateGoal is a helper class that uses the state
15     # to determine the function, function range, and function nominal
16     # automatically.
17     state = 'storage.HQ.H'
18     # One goal can introduce a single or two constraints (min and/or max). Our
19     # target water level range is 0.43 - 0.44. We might not always be able to
20     # realize this, but we want to try.
21     target_min = 0.43
22     target_max = 0.44
23
24     # Because we want to satisfy our water level target first, this has a
25     # higher priority (=lower number).
26     priority = 1

```

We also want to save energy, so we define a goal to minimize the integral of `Q_pump`. This goal has a lower priority than the water level range goal. This goal does not use a helper class:

```

29 class MinimizeQpumpGoal(Goal):
30     # This goal does not use a helper class, so we have to define the function
31     # method, range and nominal explicitly. We do not specify a target_min or
32     # target_max in this class, so the goal programming mixin will try to
33     # minimize the expression returned by the function method.
34     def function(self, optimization_problem, ensemble_member):
35         return optimization_problem.integral('Q_pump')
36
37     # The nominal is used to scale the value returned by
38     # the function method so that the value is on the order of 1.
39     function_nominal = 100.0
40     # The lower the number returned by this function, the higher the priority.
41     priority = 2
42     # The penalty variable is taken to the order'th power.
43     order = 1

```

We add a third goal minimizing the changes in “`Q_pump`”, and give it the least priority. This goal smooths out the operation of the pump so that it changes state as few times as possible. To get an idea of what the pump would have done without this goal, see Mixed Integer: *Observations*. The order of this goal must be 2, so that it penalizes both positive and negative derivatives. Order of 2 is the default, but we include it here explicitly for the sake of clarity.

```

46 class MinimizeChangeInQpumpGoal(Goal):
47     # To reduce pump power cycles, we add a third goal to minimize changes in
48     # Q_pump. This will be passed into the optimization problem as a path goal
49     # because it is an individual goal that should be applied at every time
50     # step.
51     def function(self, optimization_problem, ensemble_member):
52         return optimization_problem.der('Q_pump')
53     function_nominal = 5.0
54     priority = 3
55     # Default order is 2, but we want to be explicit
56     order = 2

```

Optimization Problem

Next, we construct the class by declaring it and inheriting the desired parent classes.

```

59 class Example(GoalProgrammingMixin, CSVMixin, ModelicaMixin,
60               CollocatedIntegratedOptimizationProblem):

```

Constraints can be declared by declaring the `path_constraints()` method. Path constraints are constraints that are applied every timestep. To set a constraint at an individual timestep, define it inside the `constraints()` method.

The “orifice” requires special constraints to be set in order to work. They are implemented below in the `path_constraints()` method. Other parent classes also declare this method, so we call the `super()` method so that we don’t overwrite their behaviour.

```

64 def path_constraints(self, ensemble_member):
65     # We want to add a few hard constraints to our problem. The goal
66     # programming mixin however also generates constraints (and objectives)
67     # from on our goals, so we have to call super() here.
68     constraints = super().path_constraints(ensemble_member)
69
70     # Release through orifice downhill only. This constraint enforces the
71     # fact that water only flows downhill
72     constraints.append((self.state('Q_orifice') +
73                        (1 - self.state('is_downhill')) * 10, 0.0, 10.0))
74
75     # Make sure is_downhill is true only when the sea is lower than the
76     # water level in the storage.
77     M = 2 # The so-called "big-M"
78     constraints.append((self.state('H_sea') - self.state('storage.HQ.H') -
79                        (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
80     constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
81                        self.state('is_downhill') * M, 0.0, np.inf))
82
83     # Orifice flow constraint. Uses the equation:
84     # Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ 0.5
85     # Note that this equation is only valid for orifices that are submerged
86     # units:  description:
87     w = 3.0 # m      width of orifice
88     d = 0.8 # m      hight of orifice
89     C = 1.0 # none   orifice constant
90     g = 9.8 # m/s^2  gravitational acceleration
91     constraints.append(
92         ((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) +
93         self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -

```

(continues on next page)

(continued from previous page)

```

94         M * (1 - self.state('is_downhill')),
95         -np.inf, 0.0))
96
97     return constraints

```

Now we pass in the goals. There are path goals and normal goals, so we have to pass them in using separate methods. A path goal is a specific kind of goal that applies to a particular variable at an individual time step, but that we want to set for all the timesteps.

Non-path goals are more general goals that are not iteratively applied at every timestep. We use the `goals()` method to pass a list of these goals to the optimizer.

```

99     def goals(self):
100         return [MinimizeQpumpGoal()]

```

For the goals that want to apply our goals to every timestep, so we use the `path_goals()` method. This is a method that returns a list of the path goals we defined above. Note that with path goals, each timestep is implemented as an independant goal- if we cannot satisfy our min/max on time step A, it will not affect our desire to satisfy the goal at time step B. Goals that inherit `StateGoal` are always path goals and must always be initialized with the parameter `self`.

```

102     def path_goals(self):
103         # Sorting goals on priority is done in the goal programming mixin. We
104         # do not have to worry about order here.
105         return [WaterLevelRangeGoal(self), MinimizeChangeInQpumpGoal()]

```

If all we cared about were the results, we could end our class declaration here. However, it is usually helpful to track how the solution changes after optimizing each priority level. To track these changes, we need to add three methods.

The method `pre()` is already defined in RTC-Tools, but we would like to add a line to it to create a variable for storing intermediate results. To do this, we declare a new `pre()` method, call `super().pre()` to ensure that the original method runs unmodified, and add in a variable declaration to store our list of intermediate results:

```

107     def pre(self):
108         # Call super() class to not overwrite default behaviour
109         super().pre()
110         # We keep track of our intermediate results, so that we can print some
111         # information about the progress of goals at the end of our run.
112         self.intermediate_results = []

```

Next, we define the `priority_completed()` method to inspect and summarize the results. These are appended to our intermediate results variable after each priority is completed.

```

114     def priority_completed(self, priority):
115         # We want to show that the results of our highest priority goal (water
116         # level) are remembered. The other information we want to see is how our
117         # lower priority goal (Q_pump) progresses. We can write some code that
118         # sumerizes the results and stores it.
119
120         # A little bit of tolerance when checking for acceptance, because
121         # strictly speaking 0.4299... is smaller than 0.43.
122         _min = 0.43 - 1e-4
123         _max = 0.44 + 1e-4
124
125         results = self.extract_results()
126         n_level_satisfied = sum(

```

(continues on next page)

(continued from previous page)

```

127         1 for x in results['storage.HQ.H'] if _min <= x <= _max)
128         q_pump_integral = sum(results['Q_pump'])
129         q_pump_sum_changes = np.sum(np.diff(results['Q_pump'])*2)
130         self.intermediate_results.append(
131             (priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes))

```

We want some way to output our intermediate results. This is accomplished using the `post()` method. Again, we need to call the `super()` method to avoid overwriting the internal method.

```

133     def post(self):
134         # Call super() class to not overwrite default behaviour
135         super().post()
136         for priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes \
137             in self.intermediate_results:
138             print('\nAfter finishing goals of priority {}'.format(priority))
139             print('Level goal satisfied at {} of {} time steps'.format(
140                 n_level_satisfied, len(self.times())))
141             print('Integral of Q_pump = {:.2f}'.format(q_pump_integral))
142             print('Sum of squares of changes in Q_pump: {:.2f}'.format(q_pump_sum_
143                 ↪ changes))

```

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

```

145     def solver_options(self):
146         options = super().solver_options()
147         solver = options['solver']
148         options[solver]['print_level'] = 1
149         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.

```

153 run_optimization_problem(Example)

```

The Whole Script

All together, the whole example script is as follows:

```

1  import numpy as np
2
3  from rtctools.optimization.collocated_integrated_optimization_problem \
4      import CollocatedIntegratedOptimizationProblem
5  from rtctools.optimization.csv_mixin import CSVMixin
6  from rtctools.optimization.goal_programming_mixin \
7      import Goal, GoalProgrammingMixin, StateGoal
8  from rtctools.optimization.modelica_mixin import ModelicaMixin
9  from rtctools.util import run_optimization_problem
10
11
12  class WaterLevelRangeGoal(StateGoal):
13      # Applying a state goal to every time step is easily done by defining a goal

```

(continues on next page)

(continued from previous page)

```

14     # that inherits StateGoal. StateGoal is a helper class that uses the state
15     # to determine the function, function range, and function nominal
16     # automatically.
17     state = 'storage.HQ.H'
18     # One goal can introduce a single or two constraints (min and/or max). Our
19     # target water level range is 0.43 - 0.44. We might not always be able to
20     # realize this, but we want to try.
21     target_min = 0.43
22     target_max = 0.44
23
24     # Because we want to satisfy our water level target first, this has a
25     # higher priority (=lower number).
26     priority = 1
27
28
29 class MinimizeQpumpGoal(Goal):
30     # This goal does not use a helper class, so we have to define the function
31     # method, range and nominal explicitly. We do not specify a target_min or
32     # target_max in this class, so the goal programming mixin will try to
33     # minimize the expression returned by the function method.
34     def function(self, optimization_problem, ensemble_member):
35         return optimization_problem.integral('Q_pump')
36
37     # The nominal is used to scale the value returned by
38     # the function method so that the value is on the order of 1.
39     function_nominal = 100.0
40     # The lower the number returned by this function, the higher the priority.
41     priority = 2
42     # The penalty variable is taken to the order'th power.
43     order = 1
44
45
46 class MinimizeChangeInQpumpGoal(Goal):
47     # To reduce pump power cycles, we add a third goal to minimize changes in
48     # Q_pump. This will be passed into the optimization problem as a path goal
49     # because it is an an individual goal that should be applied at every time
50     # step.
51     def function(self, optimization_problem, ensemble_member):
52         return optimization_problem.der('Q_pump')
53     function_nominal = 5.0
54     priority = 3
55     # Default order is 2, but we want to be explicit
56     order = 2
57
58
59 class Example(GoalProgrammingMixin, CSVMixin, ModelicaMixin,
60               CollocatedIntegratedOptimizationProblem):
61     """
62     An introductory example to goal programming in RCT-Tools
63     """
64     def path_constraints(self, ensemble_member):
65         # We want to add a few hard constraints to our problem. The goal
66         # programming mixin however also generates constraints (and objectives)
67         # from on our goals, so we have to call super() here.
68         constraints = super().path_constraints(ensemble_member)
69
70         # Release through orifice downhill only. This constraint enforces the

```

(continues on next page)

(continued from previous page)

```

71     # fact that water only flows downhill
72     constraints.append((self.state('Q_orifice') +
73                         (1 - self.state('is_downhill')) * 10, 0.0, 10.0))
74
75     # Make sure is_downhill is true only when the sea is lower than the
76     # water level in the storage.
77     M = 2 # The so-called "big-M"
78     constraints.append((self.state('H_sea') - self.state('storage.HQ.H') -
79                         (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
80     constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
81                         self.state('is_downhill') * M, 0.0, np.inf))
82
83     # Orifice flow constraint. Uses the equation:
84     #  $Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ 0.5$ 
85     # Note that this equation is only valid for orifices that are submerged
86     #      units:  description:
87     w = 3.0 # m      width of orifice
88     d = 0.8 # m      hight of orifice
89     C = 1.0 # none   orifice constant
90     g = 9.8 # m/s^2  gravitational acceleration
91     constraints.append(
92         ((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) +
93         self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -
94         M * (1 - self.state('is_downhill')),
95         -np.inf, 0.0))
96
97     return constraints
98
99     def goals(self):
100         return [MinimizeQpumpGoal()]
101
102     def path_goals(self):
103         # Sorting goals on priority is done in the goal programming mixin. We
104         # do not have to worry about order here.
105         return [WaterLevelRangeGoal(self), MinimizeChangeInQpumpGoal()]
106
107     def pre(self):
108         # Call super() class to not overwrite default behaviour
109         super().pre()
110         # We keep track of our intermediate results, so that we can print some
111         # information about the progress of goals at the end of our run.
112         self.intermediate_results = []
113
114     def priority_completed(self, priority):
115         # We want to show that the results of our highest priority goal (water
116         # level) are remembered. The other information we want to see is how our
117         # lower priority goal (Q_pump) progresses. We can write some code that
118         # sumerizes the results and stores it.
119
120         # A little bit of tolerance when checking for acceptance, because
121         # strictly speaking 0.4299... is smaller than 0.43.
122         _min = 0.43 - 1e-4
123         _max = 0.44 + 1e-4
124
125         results = self.extract_results()
126         n_level_satisfied = sum(
127             1 for x in results['storage.HQ.H'] if _min <= x <= _max)

```

(continues on next page)

(continued from previous page)

```

128     q_pump_integral = sum(results['Q_pump'])
129     q_pump_sum_changes = np.sum(np.diff(results['Q_pump'])*2)
130     self.intermediate_results.append(
131         (priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes))
132
133     def post(self):
134         # Call super() class to not overwrite default behaviour
135         super().post()
136         for priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes \
137             in self.intermediate_results:
138             print('\nAfter finishing goals of priority {}'.format(priority))
139             print('Level goal satisfied at {} of {} time steps'.format(
140                 n_level_satisfied, len(self.times())))
141             print('Integral of Q_pump = {:.2f}'.format(q_pump_integral))
142             print('Sum of squares of changes in Q_pump: {:.2f}'.format(q_pump_sum_
143 →changes))
144
145         # Any solver options can be set here
146         def solver_options(self):
147             options = super().solver_options()
148             solver = options['solver']
149             options[solver]['print_level'] = 1
150             return options
151
152     # Run
153     run_optimization_problem(Example)

```

Running the Optimization Problem

Following the execution of the optimization problem, the `post()` method should print out the following lines:

```

After finishing goals of priority 1:
Level goal satisfied at 19 of 21 time steps
Integral of Q_pump = 74.18
Sum of Changes in Q_pump: 7.83

After finishing goals of priority 2:
Level goal satisfied at 19 of 21 time steps
Integral of Q_pump = 60.10
Sum of Changes in Q_pump: 11.70

After finishing goals of priority 3:
Level goal satisfied at 19 of 21 time steps
Integral of Q_pump = 60.10
Sum of Changes in Q_pump: 10.07

```

As the output indicates, while optimizing for the priority 1 goal, no attempt was made to minimize the integral of `Q_pump`. The only objective was to minimize the number of states in violation of the water level goal.

After optimizing for the priority 2 goal, the solver was able to find a solution that reduced the integral of `Q_pump` without increasing the number of timesteps where the water level exceeded the limit. However, this solution induced additional variation into the operation of `Q_pump`.

After optimizing the priority 3 goal, the integral of `Q_pump` is the same and the level goal has not improved. Without hurting any higher priority goals, RTC-Tools was able to smooth out the operation of the pump.

Extracting Results

The results from the run are found in `output/timeseries_export.csv`. Any CSV-reading software can import it, but this is how results can be plotted using the python library `matplotlib`:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime

data_path = '../.../examples/goal_programming/output/timeseries_export.csv'
delimiter = ','

# Import Data
ncols = len(np.genfromtxt(data_path, max_rows=1, delimiter=delimiter))
datefunc = lambda x: datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
results = np.genfromtxt(data_path, converters={0: datefunc}, delimiter=delimiter,
                        dtype='object' + ',float' * (ncols - 1), names=True,
                        encoding=None)[1:]

# Generate Plot
n_subplots = 3
f, axarr = plt.subplots(n_subplots, sharex=True, figsize=(8, 3 * n_subplots))
axarr[0].set_title('Water Level and Discharge')

# Upper subplot
axarr[0].set_ylabel('Water Level [m]')
axarr[0].plot(results['time'], results['storage_level'], label='Storage',
              linewidth=2, color='b')
axarr[0].plot(results['time'], results['sea_level'], label='Sea',
              linewidth=2, color='m')

# Middle subplot
axarr[1].set_ylabel('Water Level [m]')
axarr[1].plot(results['time'], results['storage_level'], label='Storage',
              linewidth=2, color='b')
axarr[1].plot(results['time'], 0.44 * np.ones_like(results['time']), label='Storage_
↳Max',
              linewidth=2, color='r', linestyle='--')
axarr[1].plot(results['time'], 0.43 * np.ones_like(results['time']), label='Storage_
↳Min',
              linewidth=2, color='g', linestyle='--')

# Lower Subplot
axarr[2].set_ylabel('Flow Rate [m³/s]')
axarr[2].plot(results['time'], results['Q_orifice'], label='Orifice',
              linewidth=2, color='g')
axarr[2].plot(results['time'], results['Q_pump'], label='Pump',
              linewidth=2, color='r')
axarr[2].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 i in range(n_subplots):
    box = axarr[i].get_position()
    axarr[i].set_position([box.x0, box.y0, box.width * 0.8, box.height])
```

(continues on next page)

(continued from previous page)

```
axarr[i].legend(loc='center left', bbox_to_anchor=(1, 0.5), frameon=False)

plt.autoscale(enable=True, axis='x', tight=True)

# Output Plot
plt.show()
```

Using Lookup Tables

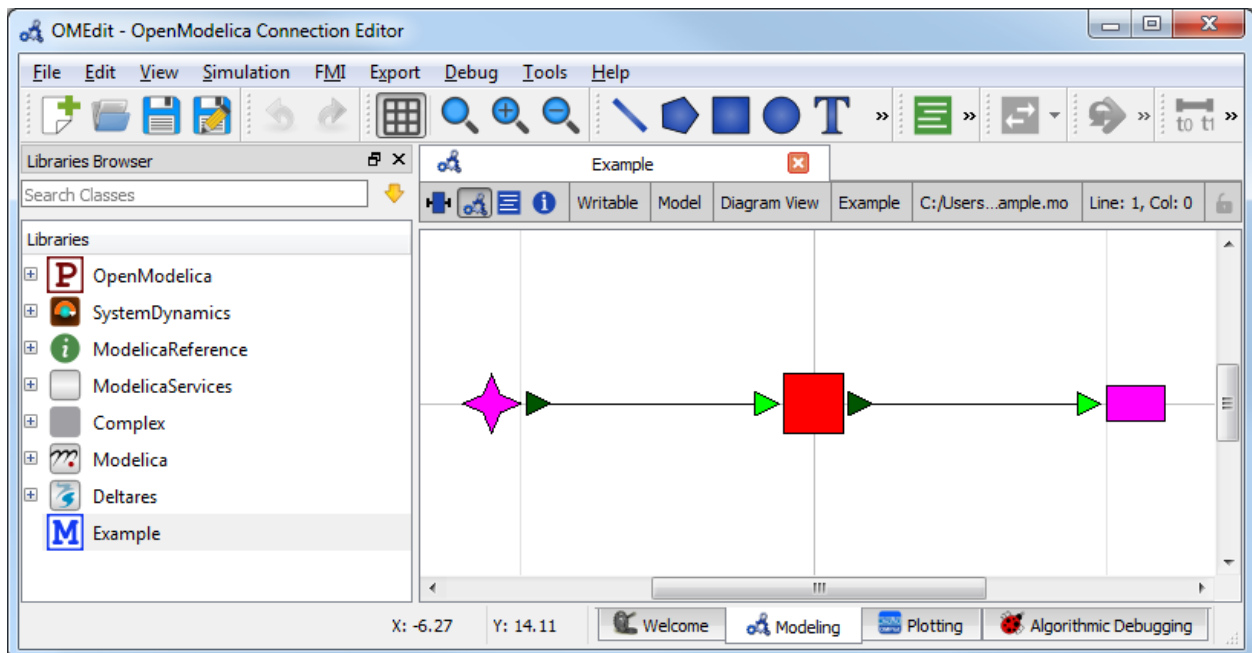
Note: This example focuses on how to implement non-linear storage elements in RTC-Tools using lookup tables. It assumes basic exposure to RTC-Tools. If you are a first-time user of RTC-Tools, see [Filling a Reservoir](#).

This example also uses goal programming in the formulation. If you are unfamiliar with goal programming, please see [Goal Programming: Defining Multiple Objectives](#).

The Model

Note: This example uses the same hydraulic model as the basic example. For a detailed explanation of the hydraulic model, see [Filling a Reservoir](#).

In OpenModelica Connection Editor, the model looks like this:



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

```
1 model Example
2   Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow inflow;
3   Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage (V(nominal=4e5, min=2e5,
  max=6e5));
```

(continues on next page)

(continued from previous page)

```

4   Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Terminal outfall;
5   input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
6   input Modelica.SIunits.VolumeFlowRate Q_release(fixed = false, min = 0.0, max = 10.
  ↪ 0);
7   equation
8     connect(inflow.QOut, storage.QIn);
9     connect(storage.QOut, outfall.QIn);
10    storage.Q_release = Q_release;
11    inflow.Q = Q_in;
12  end Example;

```

The Optimization Problem

The python script consists of the following blocks:

- Import of packages
- Declaration of Goals
- Declaration of the optimization problem class
 - Constructor
 - Declaration of a `pre()` method
 - Specification of Goals
 - Declaration of a `priority_completed()` method
 - Declaration of a `post()` method
 - Additional configuration of the solver
- A run statement

Importing Packages

For this example, the import block is as follows:

```

1  import numpy as np
2
3  from rtctools.optimization.collocated_integrated_optimization_problem \
4      import CollocatedIntegratedOptimizationProblem
5  from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
6  from rtctools.optimization.csv_mixin import CSVMixin
7  from rtctools.optimization.goal_programming_mixin \
8      import GoalProgrammingMixin, StateGoal
9  from rtctools.optimization.modelica_mixin import ModelicaMixin

```

Declaring Goals

Goals are defined as classes that inherit the `Goal` parent class. The components of goals can be found in *Multi-objective optimization*. In this example, we use the helper goal class, `StateGoal`.

First, we have a high priority goal to keep the water volume within a minimum and maximum. We use a water volume goal instead of a water level goal when the volume-storage relation of the storage element is non-linear. The volume of water in the storage element behaves linearly, while the water level does not.

However, goals are usually defined in the form of water level goals. We will convert the water level goals into volume goals within the optimization problem class, so we define the `__init__()` method so we can pass the values of the goals in later. We call the `super()` method to avoid overwriting the `__init__()` method of the parent class.

```

13 class WaterVolumeRangeGoal(StateGoal):
14     # We want to add a water volume range goal to our optimization. However, at
15     # the time of defining this goal we still do not know what the value of the
16     # min and max are. We add an __init__() method so that the values of these
17     # goals can be defined when the optimization problem class instantiates
18     # this goal.
19     def __init__(self, optimization_problem):
20         # Assign V_min and V_max the the target range
21         self.target_min = optimization_problem.get_timeseries('V_min')
22         self.target_max = optimization_problem.get_timeseries('V_max')
23         super().__init__(optimization_problem)
24         state = 'storage.V'
25         priority = 1

```

We also want to save energy, so we define a goal to minimize `Q_release`. This goal has a lower priority.

```

28 class MinimizeQreleaseGoal(StateGoal):
29     # GoalProgrammingMixin will try to minimize the following state:
30     state = 'Q_release'
31     # The lower the number returned by this function, the higher the priority.
32     priority = 2
33     # The penalty variable is taken to the order'th power.
34     order = 1

```

Optimization Problem

Next, we construct the class by declaring it and inheriting the desired parent classes.

```

37 class Example(GoalProgrammingMixin, CSVLookupTableMixin, CSVMixin,
38               ModelicaMixin, CollocatedIntegratedOptimizationProblem):

```

The method `pre()` is already defined in RTC-Tools, but we would like to add a line to it to create a variable for storing intermediate results. To do this, we declare a new `pre()` method, call `super().pre()` to ensure that the original method runs unmodified, and add in a variable declaration to store our list of intermediate results.

We also want to convert our water level rane goal into a water volume range goal. We can access the spline function describing the water level-storage relation using the `lookup_table()` method. We cache the functions for convenience. The `lookup_storage_V()` method can convert timeseries objects, and we save the water volume goal bounds as timeseries.

```

44     def pre(self):
45         super().pre()
46         # Empty list for storing intermediate_results
47         self.intermediate_results = []
48
49         # Cache lookup tables for convenience and legibility
50         _lookup_tables = self.lookup_tables(ensemble_member=0)
51         self.lookup_storage_V = _lookup_tables['storage_V']

```

(continues on next page)

(continued from previous page)

```

52
53     # Non-varying goals can be implemented as a timeseries like this:
54     self.set_timeseries('H_min', np.ones_like(self.times()) * 0.44, output=False)
55
56     # Q_in is a varying input and is defined in timeseries_import.csv
57     # However, if we set it again here, it will be added to the output file
58     self.set_timeseries('Q_in', self.get_timeseries('Q_in'))
59
60     # Convert our water level constraints into volume constraints
61     self.set_timeseries('V_max',
62                        self.lookup_storage_V(self.get_timeseries('H_max')))
63     self.set_timeseries('V_min',
64                        self.lookup_storage_V(self.get_timeseries('H_min')))

```

Notice that `H_max` was not defined in `pre()`. This is because it was defined as a timeseries import. We access timeseries using `get_timeseries()` and store them using `set_timeseries()`. Once a timeseries is set, we can access it later. In addition, all timeseries that are set are automatically included in the output file. You can find more information on timeseries here [Basics](#).

Now we pass in the goals. We want to apply our goals to every timestep, so we use the `path_goals()` method. This is a method that returns a list of the goals we defined above. The `WaterVolumeRangeGoal` needs to be instantiated with the new water volume timeseries we just defined.

```

66     def path_goals(self):
67         g = []
68         g.append(WaterVolumeRangeGoal(self))
69         g.append(MinimizeQreleaseGoal(self))
70         return g

```

If all we cared about were the results, we could end our class declaration here. However, it is usually helpful to track how the solution changes after optimizing each priority level. To track these changes, we need to add three methods.

We define the `priority_completed()` method to inspect and summarize the results. These are appended to our intermediate results variable after each priority is completed.

```

75     def priority_completed(self, priority):
76         results = self.extract_results()
77         self.set_timeseries('storage_V', results['storage.V'])
78
79         _max = self.get_timeseries('V_max').values
80         _min = self.get_timeseries('V_min').values
81         storage_V = self.get_timeseries('storage_V').values
82
83         # A little bit of tolerance when checking for acceptance.
84         tol = 10
85         _max += tol
86         _min -= tol
87         n_level_satisfied = sum(
88             np.logical_and(_min <= storage_V, storage_V <= _max))
89         q_release_integral = sum(results['Q_release'])
90         self.intermediate_results.append(
91             (priority, n_level_satisfied, q_release_integral))

```

We output our intermediate results using the `post()` method. Again, we need to call the `super()` method to avoid overwriting the internal method.


```

93     def post(self):
94         # Call super() class to not overwrite default behaviour
95         super().post()
96         for priority, n_level_satisfied, q_release_integral in self.intermediate_
↪results:
97             print("\nAfter finishing goals of priority {}".format(priority))
98             print("Volume goal satisfied at {} of {} time steps".format(
99                 n_level_satisfied, len(self.times())))
100             print("Integral of Q_release = {:.2f}".format(q_release_integral))

```

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

```

103     def solver_options(self):
104         options = super().solver_options()
105         solver = options['solver']
106         options[solver]['print_level'] = 1
107         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.

```

111 run_optimization_problem(Example)

```

The Whole Script

All together, the whole example script is as follows:

```

1  import numpy as np
2
3  from rtctools.optimization.collocated_integrated_optimization_problem \
4      import CollocatedIntegratedOptimizationProblem
5  from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
6  from rtctools.optimization.csv_mixin import CSVMixin
7  from rtctools.optimization.goal_programming_mixin \
8      import GoalProgrammingMixin, StateGoal
9  from rtctools.optimization.modelica_mixin import ModelicaMixin
10 from rtctools.util import run_optimization_problem
11
12
13 class WaterVolumeRangeGoal(StateGoal):
14     # We want to add a water volume range goal to our optimization. However, at
15     # the time of defining this goal we still do not know what the value of the
16     # min and max are. We add an __init__() method so that the values of these
17     # goals can be defined when the optimization problem class instantiates
18     # this goal.
19     def __init__(self, optimization_problem):
20         # Assign V_min and V_max the the target range
21         self.target_min = optimization_problem.get_timeseries('V_min')
22         self.target_max = optimization_problem.get_timeseries('V_max')
23         super().__init__(optimization_problem)
24         state = 'storage.V'
25         priority = 1

```

(continues on next page)

(continued from previous page)

```

26
27
28 class MinimizeQreleaseGoal(StateGoal):
29     # GoalProgrammingMixin will try to minimize the following state:
30     state = 'Q_release'
31     # The lower the number returned by this function, the higher the priority.
32     priority = 2
33     # The penalty variable is taken to the order'th power.
34     order = 1
35
36
37 class Example(GoalProgrammingMixin, CSVLookupTableMixin, CSVMixin,
38               ModelicaMixin, CollocatedIntegratedOptimizationProblem):
39     """
40     An extension of the goal programming example that shows how to incorporate
41     non-linear storage elements in the model.
42     """
43
44     def pre(self):
45         super().pre()
46         # Empty list for storing intermediate_results
47         self.intermediate_results = []
48
49         # Cache lookup tables for convenience and legibility
50         _lookup_tables = self.lookup_tables(ensemble_member=0)
51         self.lookup_storage_V = _lookup_tables['storage_V']
52
53         # Non-varying goals can be implemented as a timeseries like this:
54         self.set_timeseries('H_min', np.ones_like(self.times()) * 0.44, output=False)
55
56         # Q_in is a varying input and is defined in timeseries_import.csv
57         # However, if we set it again here, it will be added to the output file
58         self.set_timeseries('Q_in', self.get_timeseries('Q_in'))
59
60         # Convert our water level constraints into volume constraints
61         self.set_timeseries('V_max',
62                             self.lookup_storage_V(self.get_timeseries('H_max')))
63         self.set_timeseries('V_min',
64                             self.lookup_storage_V(self.get_timeseries('H_min')))
65
66     def path_goals(self):
67         g = []
68         g.append(WaterVolumeRangeGoal(self))
69         g.append(MinimizeQreleaseGoal(self))
70         return g
71
72     # We want to print some information about our goal programming problem. We
73     # store the useful numbers temporarily, and print information at the end of
74     # our run (see post() method below).
75     def priority_completed(self, priority):
76         results = self.extract_results()
77         self.set_timeseries('storage_V', results['storage.V'])
78
79         _max = self.get_timeseries('V_max').values
80         _min = self.get_timeseries('V_min').values
81         storage_V = self.get_timeseries('storage_V').values
82

```

(continues on next page)

(continued from previous page)

```

83     # A little bit of tolerance when checking for acceptance.
84     tol = 10
85     _max += tol
86     _min -= tol
87     n_level_satisfied = sum(
88         np.logical_and(_min <= storage_V, storage_V <= _max))
89     q_release_integral = sum(results['Q_release'])
90     self.intermediate_results.append(
91         (priority, n_level_satisfied, q_release_integral))
92
93     def post(self):
94         # Call super() class to not overwrite default behaviour
95         super().post()
96         for priority, n_level_satisfied, q_release_integral in self.intermediate_
↪results:
97             print("\nAfter finishing goals of priority {}".format(priority))
98             print("Volume goal satisfied at {} of {} time steps".format(
99                 n_level_satisfied, len(self.times())))
100             print("Integral of Q_release = {:.2f}".format(q_release_integral))
101
102     # Any solver options can be set here
103     def solver_options(self):
104         options = super().solver_options()
105         solver = options['solver']
106         options[solver]['print_level'] = 1
107         return options
108
109
110 # Run
111 run_optimization_problem(Example)

```

Running the Optimization Problem

Following the execution of the optimization problem, the `post()` method should print out the following lines:

```

After finishing goals of priority 1:
Volume goal satisfied at 12 of 12 time steps
Integral of Q_release = 42.69

After finishing goals of priority 2:
Volume goal satisfied at 12 of 12 time steps
Integral of Q_release = 42.58

```

As the output indicates, while optimizing for the priority 1 goal, no attempt was made to minimize the integral of `Q_release`. The only objective was to minimize the number of states in violation of the water level goal.

After optimizing for the priority 2 goal, the solver was able to find a solution that reduced the integral of `Q_release` without increasing the number of timesteps where the water level exceeded the limit.

Extracting Results

The results from the run are found in `output/timeseries_export.csv`. Any CSV-reading software can import it, but this is how results can be plotted using the python library `matplotlib`:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime

data_path = '../.../examples/lookup_table/output/timeseries_export.csv'
delimiter = ','

# Import Data
ncols = len(np.genfromtxt(data_path, max_rows=1, delimiter=delimiter))
datefunc = lambda x: datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
results = np.genfromtxt(data_path, converters={0: datefunc}, delimiter=delimiter,
                        dtype='object' + ',float' * (ncols - 1), names=True,
                        encoding=None)[1:]

# Generate Plot
n_subplots = 2
f, axarr = plt.subplots(n_subplots, sharex=True, figsize=(8, 3 * n_subplots))
axarr[0].set_title('Water Volume and Discharge')
f.autofmt_xdate()

# Upper subplot
axarr[0].set_ylabel('Water Volume [m³]')
axarr[0].ticklabel_format(style='sci', axis='y', scilimits=(0, 0))
axarr[0].plot(results['time'], results['storage_V'], label='Storage',
              linewidth=2, color='b')
axarr[0].plot(results['time'], results['V_max'], label='Storage Max',
              linewidth=2, color='r', linestyle='--')
axarr[0].plot(results['time'], results['V_min'], label='Storage Min',
              linewidth=2, color='g', linestyle='--')

# Lower Subplot
axarr[1].set_ylabel('Flow Rate [m³/s]')
axarr[1].plot(results['time'], results['Q_in'], label='Inflow',
              linewidth=2, color='g')
axarr[1].plot(results['time'], results['Q_release'], label='Release',
              linewidth=2, color='r')

# Shrink each axis by 20% and put a legend to the right of the axis
for i in range(n_subplots):
    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)

plt.autoscale(enable=True, axis='x', tight=True)

# Output Plot
plt.show()
```

Using an Ensemble Forecast

Note: This example is an extension of *Using Lookup Tables*. It assumes prior knowledge of goal programming and the lookup tables components of RTC-Tools. If you are a first-time user of RTC-Tools, see *Filling a Reservoir*.

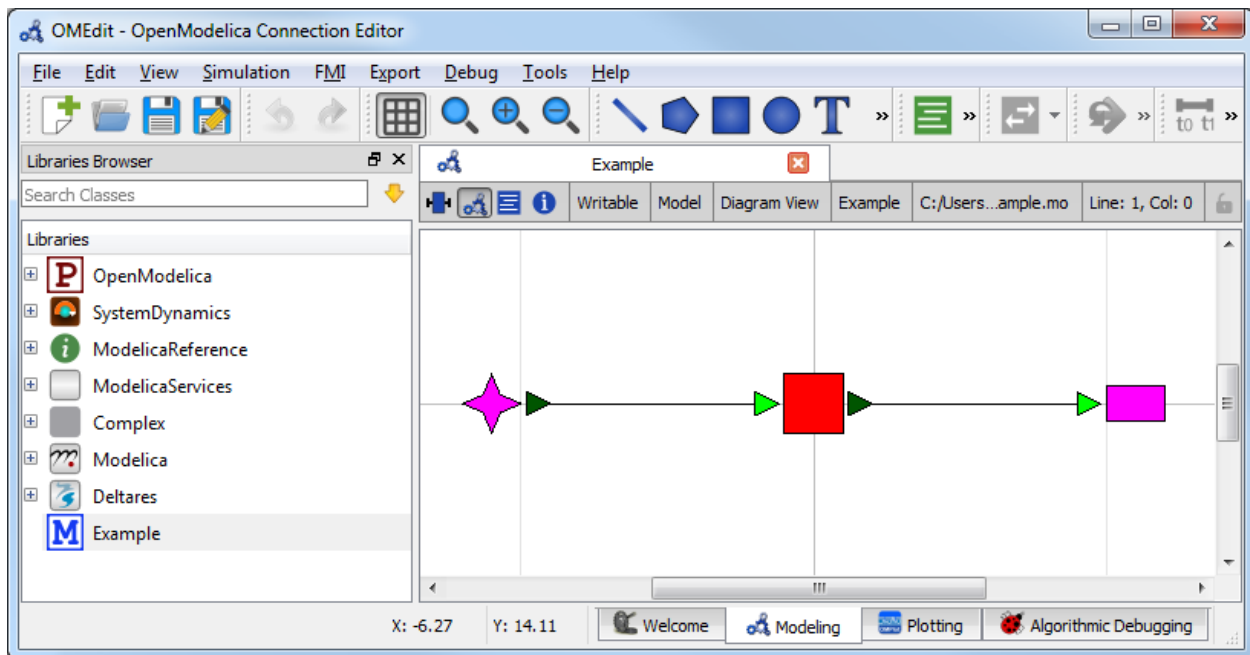
Then biggest change to RTC-Tools when using an ensemble is the structure of the directory. The folder `<examples directory>\ensemble` contains a complete RTC-Tools ensemble optimization problem. An RTC-Tools ensemble directory has the following structure:

- `model`: This folder contains the Modelica model. The Modelica model contains the physics of the RTC-Tools model.
- `src`: This folder contains a Python file. This file contains the configuration of the model and is used to run the model.
- `input`: This folder contains the model input data pertaining to each ensemble member:
 - `ensemble.csv`: a file where the names and probabilities of the ensemble members are defined
 - `forecast1`
 - * the file `timeseries_import.csv`
 - * the file `initial_state.csv`
 - `forecast2`
 - * `timeseries_import.csv`
 - * `initial_state.csv`
 - ...
- `output`: The folder where the output is saved:
 - `forecast1`
 - * `timeseries_export.csv`
 - `forecast2`
 - * `timeseries_export.csv`
 - ...

The Model

Note: This example uses the same hydraulic model as the basic example. For a detailed explanation of the hydraulic model, see [Filling a Reservoir](#).

In OpenModelica Connection Editor, the model looks like this:



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

```

1  model Example
2    Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow inflow;
3    Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(V(nominal=4e5, min=2e5,
4    ↳max=6e5));
5    Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Terminal outfall;
6    input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
7    input Modelica.SIunits.VolumeFlowRate Q_release(fixed = false, min = 0.0, max = 6.
8    ↳0);
9  equation
10    connect(inflow.QOut, storage.QIn);
11    connect(storage.QOut, outfall.QIn);
12    storage.Q_release = Q_release;
13    inflow.Q = Q_in;
14  end Example;

```

The Optimization Problem

The python script consists of the following blocks:

- Import of packages
- Declaration of Goals
- Declaration of the optimization problem class
 - Constructor
 - Set `csv_ensemble_mode = True`
 - Declaration of a `pre()` method
 - Specification of Goals
 - Declaration of a `priority_completed()` method

- Declaration of a `post()` method
- Additional configuration of the solver
- A run statement

Importing Packages

For this example, the import block is as follows:

```

1 import numpy as np
2
3 from rtctools.optimization.collocated_integrated_optimization_problem \
4     import CollocatedIntegratedOptimizationProblem
5 from rtctools.optimization.control_tree_mixin import ControlTreeMixin
6 from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
7 from rtctools.optimization.csv_mixin import CSVMixin
8 from rtctools.optimization.goal_programming_mixin \
9     import GoalProgrammingMixin, StateGoal
10 from rtctools.optimization.modelica_mixin import ModelicaMixin

```

Declaring Goals

First, we have a high priority goal to keep the water volume within a minimum and maximum.

```

14 class WaterVolumeRangeGoal(StateGoal):
15     def __init__(self, optimization_problem):
16         # Assign V_min and V_max the the target range
17         self.target_min = optimization_problem.get_timeseries('V_min')
18         self.target_max = optimization_problem.get_timeseries('V_max')
19         super().__init__(optimization_problem)
20         state = 'storage.V'
21         priority = 1

```

We also want to save energy, so we define a goal to minimize `Q_release`. This goal has a lower priority.

```

24 class MinimizeQreleaseGoal(StateGoal):
25     # GoalProgrammingMixin will try to minimize the following state
26     state = 'Q_release'
27     # The lower the number returned by this function, the higher the priority.
28     priority = 2
29     # The penalty variable is taken to the order'th power.
30     order = 1

```

Optimization Problem

Next, we construct the class by declaring it and inheriting the desired parent classes.

```

33 class Example(GoalProgrammingMixin, CSVMixin, CSVLookupTableMixin, ModelicaMixin,
34               ControlTreeMixin, CollocatedIntegratedOptimizationProblem):

```

We turn on ensemble mode by setting `csv_ensemble_mode = True`:

```
ensemble_member: [] for ensemble_member in range(self.ensemble_size)}
```

The method `pre()` is already defined in RTC-Tools, but we would like to add a line to it to create a variable for storing intermediate results. To do this, we declare a new `pre()` method, call `super().pre()` to ensure that the original method runs unmodified, and add in a variable declaration to store our list of intermediate results. This variable is a dict, reflecting the need to store results from multiple ensemble.

Because the timeseries we set will be the same for both ensemble members, we also make sure that the timeseries we set are set for both ensemble members using for loops.

```
def pre(self):
    # Do the standard preprocessing
    super().pre()

    # Create a dict of empty lists for storing intermediate results from
    # each ensemble
    self.intermediate_results = {
        ensemble_member: [] for ensemble_member in range(self.ensemble_size)}

    # Cache lookup tables for convenience and code legibility
    _lookup_tables = self.lookup_tables(ensemble_member=0)
    self.lookup_storage_V = _lookup_tables['storage_V']

    # Non-varying goals can be implemented as a timeseries
    for e_m in range(self.ensemble_size):
        self.set_timeseries('H_min', np.ones_like(self.times()) * 0.44,
                           ensemble_member=e_m)
        self.set_timeseries('H_max', np.ones_like(self.times()) * 0.46,
                           ensemble_member=e_m)

        # Q_in is a varying input and is defined in each timeseries_import.csv
        # However, if we set it again here, it will be added to the output files
        self.set_timeseries('Q_in',
                           self.get_timeseries('Q_in', ensemble_member=e_m),
                           ensemble_member=e_m)

        # Convert our water level goals into volume goals
        self.set_timeseries('V_max',
                           self.lookup_storage_V(self.get_timeseries('H_max')),
                           ensemble_member=e_m)
        self.set_timeseries('V_min',
                           self.lookup_storage_V(self.get_timeseries('H_min')),
                           ensemble_member=e_m)
```

Now we pass in the goals:

```
def path_goals(self):
    g = []
    g.append(WaterVolumeRangeGoal(self))
    g.append(MinimizeQreleaseGoal(self))
    return g
```

In order to better demonstrate the way that ensembles are handled in RTC-Tools, we modify the `control_tree_options()` method. The default setting allows the control tree to split at every time, but we override this method and force it to split at a single timestep. See [Observations](#) at the bottom of the page for more information.


```

82 def control_tree_options(self):
83     # We want to modify the control tree options, so we override the default
84     # control_tree_options method. We call super() to get the default options
85     options = super().control_tree_options()
86     # Change the branching_times list to only contain the fifth timestep
87     options['branching_times'] = [self.times()[5]]
88     return options

```

We define the `priority_completed()` method. We ensure that it stores the results from both ensemble members.

```

90 def priority_completed(self, priority):
91     # We want to print some information about our goal programming problem.
92     # We store the useful numbers temporarily, and print information at the
93     # end of our run.
94     for e_m in range(self.ensemble_size):
95         results = self.extract_results(e_m)
96         self.set_timeseries('V_storage', results['storage.V'], ensemble_member=e_
↪m)
97
98         _max = self.get_timeseries('V_max', ensemble_member=e_m).values
99         _min = self.get_timeseries('V_min', ensemble_member=e_m).values
100         V_storage = self.get_timeseries('V_storage', ensemble_member=e_m).values
101
102         # A little bit of tolerance when checking for acceptance. This
103         # tolerance must be set greater than the tolerance of the solver.
104         tol = 10
105         _max += tol
106         _min -= tol
107         n_level_satisfied = sum(
108             np.logical_and(_min <= V_storage, V_storage <= _max))
109         q_release_integral = sum(results['Q_release'])
110         self.intermediate_results[e_m].append((priority, n_level_satisfied,
111             q_release_integral))

```

We output our intermediate results using the `post()` method:

```

113 def post(self):
114     super().post()
115     for e_m in range(self.ensemble_size):
116         print('\n\nResults for Ensemble Member {}'.format(e_m))
117         for priority, n_level_satisfied, q_release_integral in \
118             self.intermediate_results[e_m]:
119             print("\nAfter finishing goals of priority {}".format(priority))
120             print("Level goal satisfied at {} of {} time steps".format(
121                 n_level_satisfied, len(self.times())))
122             print("Integral of Q_release = {:.2f}".format(q_release_integral))

```

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

```

125 def solver_options(self):
126     options = super().solver_options()
127     # When mumps_scaling is not zero, errors occur. RTC-Tools does its own
128     # scaling, so mumps scaling is not critical. Proprietary HSL solvers
129     # do not exhibit this error.
130     solver = options['solver']
131     options[solver]['mumps_scaling'] = 0
132     options[solver]['print_level'] = 1
133     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.

```
137 run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```
1 import numpy as np
2
3 from rtctools.optimization.collocated_integrated_optimization_problem \
4     import CollocatedIntegratedOptimizationProblem
5 from rtctools.optimization.control_tree_mixin import ControlTreeMixin
6 from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
7 from rtctools.optimization.csv_mixin import CSVMixin
8 from rtctools.optimization.goal_programming_mixin \
9     import GoalProgrammingMixin, StateGoal
10 from rtctools.optimization.modelica_mixin import ModelicaMixin
11 from rtctools.util import run_optimization_problem
12
13
14 class WaterVolumeRangeGoal(StateGoal):
15     def __init__(self, optimization_problem):
16         # Assign V_min and V_max the the target range
17         self.target_min = optimization_problem.get_timeseries('V_min')
18         self.target_max = optimization_problem.get_timeseries('V_max')
19         super().__init__(optimization_problem)
20         state = 'storage.V'
21         priority = 1
22
23
24 class MinimizeQreleaseGoal(StateGoal):
25     # GoalProgrammingMixin will try to minimize the following state
26     state = 'Q_release'
27     # The lower the number returned by this function, the higher the priority.
28     priority = 2
29     # The penalty variable is taken to the order'th power.
30     order = 1
31
32
33 class Example(GoalProgrammingMixin, CSVMixin, CSVLookupTableMixin, ModelicaMixin,
34               ControlTreeMixin, CollocatedIntegratedOptimizationProblem):
35     """
36     An extention of the goal programming and lookuptable examples that
37     demonstrates how to work with ensembles.
38     """
39     # Override default csv_ensemble_mode = False from CSVMixin before calling pre()
40     csv_ensemble_mode = True
41
42     def pre(self):
43         # Do the standard preprocessing
44         super().pre()
45
```

(continues on next page)

(continued from previous page)

```

46     # Create a dict of empty lists for storing intermediate results from
47     # each ensemble
48     self.intermediate_results = {
49         ensemble_member: [] for ensemble_member in range(self.ensemble_size)}
50
51     # Cache lookup tables for convenience and code legibility
52     _lookup_tables = self.lookup_tables(ensemble_member=0)
53     self.lookup_storage_V = _lookup_tables['storage_V']
54
55     # Non-varying goals can be implemented as a timeseries
56     for e_m in range(self.ensemble_size):
57         self.set_timeseries('H_min', np.ones_like(self.times()) * 0.44,
58                             ensemble_member=e_m)
59         self.set_timeseries('H_max', np.ones_like(self.times()) * 0.46,
60                             ensemble_member=e_m)
61
62         # Q_in is a varying input and is defined in each timeseries_import.csv
63         # However, if we set it again here, it will be added to the output files
64         self.set_timeseries('Q_in',
65                             self.get_timeseries('Q_in', ensemble_member=e_m),
66                             ensemble_member=e_m)
67
68         # Convert our water level goals into volume goals
69         self.set_timeseries('V_max',
70                             self.lookup_storage_V(self.get_timeseries('H_max')),
71                             ensemble_member=e_m)
72         self.set_timeseries('V_min',
73                             self.lookup_storage_V(self.get_timeseries('H_min')),
74                             ensemble_member=e_m)
75
76     def path_goals(self):
77         g = []
78         g.append(WaterVolumeRangeGoal(self))
79         g.append(MinimizeQreleaseGoal(self))
80         return g
81
82     def control_tree_options(self):
83         # We want to modify the control tree options, so we override the default
84         # control_tree_options method. We call super() to get the default options
85         options = super().control_tree_options()
86         # Change the branching_times list to only contain the fifth timestep
87         options['branching_times'] = [self.times()[5]]
88         return options
89
90     def priority_completed(self, priority):
91         # We want to print some information about our goal programming problem.
92         # We store the useful numbers temporarily, and print information at the
93         # end of our run.
94         for e_m in range(self.ensemble_size):
95             results = self.extract_results(e_m)
96             self.set_timeseries('V_storage', results['storage.V'], ensemble_member=e_
97 ↪m)
98
99             _max = self.get_timeseries('V_max', ensemble_member=e_m).values
100             _min = self.get_timeseries('V_min', ensemble_member=e_m).values
101             V_storage = self.get_timeseries('V_storage', ensemble_member=e_m).values

```

(continues on next page)

(continued from previous page)

```

102     # A little bit of tolerance when checking for acceptance. This
103     # tolerance must be set greater than the tolerance of the solver.
104     tol = 10
105     _max += tol
106     _min -= tol
107     n_level_satisfied = sum(
108         np.logical_and(_min <= V_storage, V_storage <= _max))
109     q_release_integral = sum(results['Q_release'])
110     self.intermediate_results[e_m].append((priority, n_level_satisfied,
111         q_release_integral))
112
113     def post(self):
114         super().post()
115         for e_m in range(self.ensemble_size):
116             print('\n\nResults for Ensemble Member {}'.format(e_m))
117             for priority, n_level_satisfied, q_release_integral in \
118                 self.intermediate_results[e_m]:
119                 print("\nAfter finishing goals of priority {}".format(priority))
120                 print("Level goal satisfied at {} of {} time steps".format(
121                     n_level_satisfied, len(self.times())))
122                 print("Integral of Q_release = {:.2f}".format(q_release_integral))
123
124     # Any solver options can be set here
125     def solver_options(self):
126         options = super().solver_options()
127         # When mumps_scaling is not zero, errors occur. RTC-Tools does its own
128         # scaling, so mumps scaling is not critical. Proprietary HSL solvers
129         # do not exhibit this error.
130         solver = options['solver']
131         options[solver]['mumps_scaling'] = 0
132         options[solver]['print_level'] = 1
133         return options
134
135     # Run
136     run_optimization_problem(Example)
137

```

Running the Optimization Problem

Following the execution of the optimization problem, the `post()` method should print out the following lines:

```

Results for Ensemble Member 0:

After finishing goals of priority 1:
Level goal satisfied at 10 of 12 time steps
Integral of Q_release = 17.34

After finishing goals of priority 2:
Level goal satisfied at 9 of 12 time steps
Integral of Q_release = 17.12

Results for Ensemble Member 1:

After finishing goals of priority 1:

```

(continues on next page)

(continued from previous page)

```

Level goal satisfied at 10 of 12 time steps
Integral of Q_release = 20.82

After finishing goals of priority 2:
Level goal satisfied at 9 of 12 time steps
Integral of Q_release = 20.60

```

This is the same output as the output for *Mixed Integer Optimization: Pumps and Orifices*, except now the output for each ensemble is printed.

Extracting Results

The results from the run are found in `output/forecast1/timeseries_export.csv` and `output/forecast2/timeseries_export.csv`. Any CSV-reading software can import it, but this is how results can be plotted using the python library matplotlib:

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime
from pylab import get_cmap

forecast_names = ['forecast1', 'forecast2']

# Import Data
def get_results(forecast_name):
    output_dir = '../../../examples/ensemble/output/'
    data_path = output_dir + forecast_name + '/timeseries_export.csv'
    delimiter = ','
    ncols = len(np.genfromtxt(data_path, max_rows=1, delimiter=delimiter))
    datefunc = lambda x: datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
    return np.genfromtxt(data_path, converters={0: datefunc}, delimiter=delimiter,
                        dtype='object' + ',float' * (ncols - 1), names=True,
                        encoding=None)

# Generate Plot
n_subplots = 2
f, axarr = plt.subplots(n_subplots, sharex=True, figsize=(8, 4 * n_subplots))
axarr[0].set_title('Water Volume and Discharge')
cmaps = ['Blues', 'Greens']
shades = [0.5, 0.8]
f.autofmt_xdate()

# Upper Subplot
axarr[0].set_ylabel('Water Volume in Storage [m³]')
axarr[0].ticklabel_format(style='sci', axis='y', scilimits=(0, 0))

# Lower Subplot
axarr[1].set_ylabel('Flow Rate [m³/s]')

# Plot Ensemble Members
for idx, forecast in enumerate(forecast_names):
    # Upper Subplot
    results = get_results(forecast)
    if idx == 0:

```

(continues on next page)

(continued from previous page)

```
axarr[0].plot(results['time'], results['V_max'], label='Max',
              linewidth=2, color='r', linestyle='--')
axarr[0].plot(results['time'], results['V_min'], label='Min',
              linewidth=2, color='g', linestyle='--')
axarr[0].plot(results['time'], results['V_storage'], label=forecast + ':Volume',
              linewidth=2, color=get_cmap(cmaps[idx])(shades[1]))

# Lower Subplot
axarr[1].plot(results['time'], results['Q_in'], label='{:Inflow'.
→format(forecast),
              linewidth=2, color=get_cmap(cmaps[idx])(shades[0]))
axarr[1].plot(results['time'], results['Q_release'], label='{:Release'.
→format(forecast),
              linewidth=2, color=get_cmap(cmaps[idx])(shades[1]))

# Shrink each axis by 30% 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 * 0.7, box.height])
    axarr[i].legend(loc='center left', bbox_to_anchor=(1, 0.5), frameon=False)

plt.autoscale(enable=True, axis='x', tight=True)

# Output Plot
plt.show()
```

Observations

Note that in the results plotted above, the control tree follows a single path and does not branch until it arrives at the first branching time. Up until the branching point, RTC-Tools is making decisions that enhance the flexibility of the system so that it can respond as ideally as possible to whichever future emerges. In the case of two forecasts, this means taking a path that falls between the two possible futures. This will cause the water level to diverge from the ideal levels as time progresses. While this appears to be suboptimal, it is preferable to simply gambling on one of the forecasts coming true and ignoring the other. Once the branching time is reached, RTC-Tools is allowed to optimize for each individual branch separately. Immediately, RTC-Tools applies the corrective control needed to get the water levels into the acceptable range. If the operator simply picks a forecast to use and guesses wrong, the corrective control will have to be much more drastic and potentially catastrophic.

Cascading Channels: Modeling Channel Hydraulics



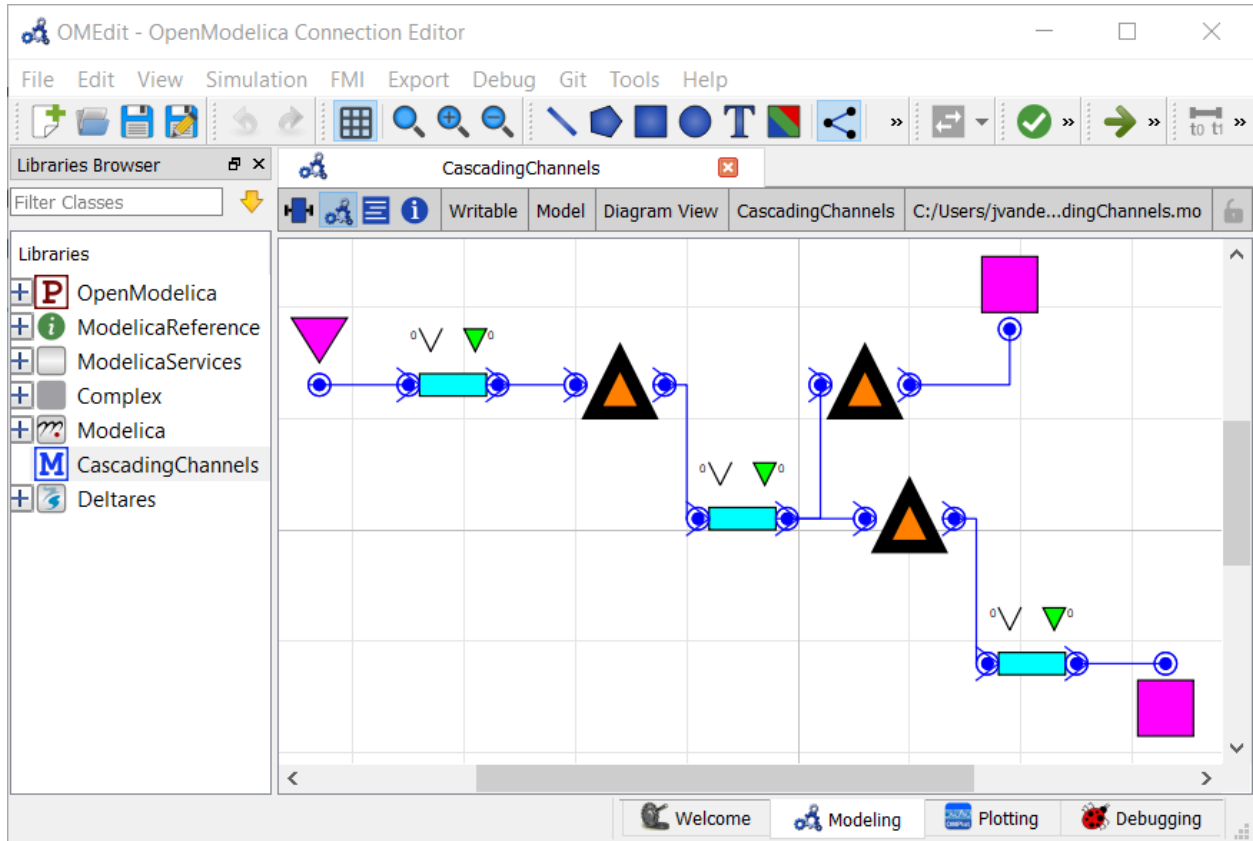
Note: This is a more advanced example that implements multi-objective optimization in RTC-Tools. It also capitalizes on the homotopy techniques available in RTC-Tools. If you are a first-time user of RTC-Tools, see [Filling a Reservoir](#).

Goal programming is a way to satisfy (sometimes conflicting) goals by ranking the goals by priority. In this example, we specify two goals. The higher priority goal will be to maintain the water levels in the channels within a desired band. The lower priority goal will be to extract water to meet a forecasted drinking water demand.

The Model

For this example, water is flowing through a multilevel channel system. The model has three channel sections. There is an extraction pump at the downstream end of the middle channel. The algorithm will first attempt to maintain water levels in the channels within the desired water level band. Using the remaining flexibility in the model, the algorithm will attempt to meet the diurnal demand pattern as best as it can with the extraction pump.

In OpenModelica Connection Editor, the model looks like this:



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

```

1 model Example
2   // Model Elements
3   Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Discharge Inflow;
4   Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level DrinkingWaterPlant(H = 10.);
5   Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level Level(H = 0.);
6   Deltares.ChannelFlow.Hydraulic.Branches.HomotopicLinear LowerChannel(H(each max = 1.
7     ↳0), H_b_down = -2.0, H_b_up = -1.5, friction_coefficient = 10., length = 2000.,
8     ↳theta = theta, uniform_nominal_depth = 1.75, width_down = 10., width_up = 10.);
9   Deltares.ChannelFlow.Hydraulic.Branches.HomotopicLinear MiddleChannel(H(each max =
10     ↳1.5), H_b_down = -1.5, H_b_up = -1.0, friction_coefficient = 10., length = 2000.,
11     ↳theta = theta, uniform_nominal_depth = 1.75, width_down = 10., width_up = 10.);
12   Deltares.ChannelFlow.Hydraulic.Branches.HomotopicLinear UpperChannel(H(each max = 2.
13     ↳0), H_b_down = -1.0, H_b_up = -0.5, friction_coefficient = 10., length = 2000.,
14     ↳theta = theta, uniform_nominal_depth = 1.75, width_down = 10., width_up = 10.);
15   Deltares.ChannelFlow.Hydraulic.Structures.Pump DrinkingWaterExtractionPump;
16   Deltares.ChannelFlow.Hydraulic.Structures.Pump LowerControlStructure;
17   Deltares.ChannelFlow.Hydraulic.Structures.Pump UpperControlStructure;
18   // Parameters
19   parameter Real theta;
20   // Inputs
21   input Real Inflow_Q(fixed = true) = Inflow.Q;
22   input Real UpperControlStructure_Q(fixed = false, min = 0., max = 4.) =
23     ↳UpperControlStructure.Q;
24   input Real LowerControlStructure_Q(fixed = false, min = 0., max = 4.) =
25     ↳LowerControlStructure.Q;
26   input Real DrinkingWaterExtractionPump_Q(fixed = false, min = 0., max = 2.) =
27     ↳DrinkingWaterExtractionPump.Q;

```

(continues on next page)

(continued from previous page)

```

19 equation
20   connect (DrinkingWaterExtractionPump.HQDown, DrinkingWaterPlant.HQ);
21   connect (Inflow.HQ, UpperChannel.HQUp);
22   connect (LowerChannel.HQDown, Level.HQ);
23   connect (LowerControlStructure.HQDown, LowerChannel.HQUp);
24   connect (MiddleChannel.HQDown, DrinkingWaterExtractionPump.HQUp);
25   connect (MiddleChannel.HQDown, LowerControlStructure.HQUp);
26   connect (UpperChannel.HQDown, UpperControlStructure.HQUp);
27   connect (UpperControlStructure.HQDown, MiddleChannel.HQUp);
28 end Example;

```

The Optimization Problem

The python script consists of the following blocks:

- Import of packages
- Declaration of Goals
- Declaration of the optimization problem class
 - Constructor
 - Implementation of `pre()` method
 - Implementation of `path_goals()` method
- A run statement

Goals

In this model, we define two generic StateGoal subclasses:

```

13 class RangeGoal(StateGoal):
14     def __init__(self, opt_prob, state, priority):
15         self.state = state
16         self.target_min = opt_prob.get_timeseries(state + "_min")
17         self.target_max = opt_prob.get_timeseries(state + "_max")
18         self.violation_timeseries_id = state + "_target_violation"
19         self.function_value_timeseries_id = state
20         self.priority = priority
21         super().__init__(opt_prob)

```

```

24 class TargetGoal(StateGoal):
25     def __init__(self, opt_prob, state, priority):
26         self.state = state
27         self.target_min = opt_prob.get_timeseries(state + "_target")
28         self.target_max = self.target_min
29         self.violation_timeseries_id = state + "_target_violation"
30         self.function_value_timeseries_id = state
31         self.priority = priority
32         super().__init__(opt_prob)

```

These goals are actually really similar. The only difference is that the TargetGoal uses the same timeseries for its `target_max` and `target_min` attributes. This goal will try to minimize the difference between the target and the

goal's state. This is in contrast to the `RangeGoal`, which has a separate min and max that define an acceptable range of values.

You can read more about the components of goals in the documentation: [Multi-objective optimization](#).

Optimization Problem

We construct the class by declaring it and inheriting the desired parent classes.

```

35 class Example(
36     HomotopyMixin,
37     GoalProgrammingMixin,
38     CSVMixin,
39     ModelicaMixin,
40     CollocatedIntegratedOptimizationProblem,
41 ):

```

In our new class, we implement the `pre()` method. This method is a good place to do some preprocessing of the data to make sure it is all there when the model runs.

```

45 def pre(self):
46     super().pre()
47     # Generate handy tuples to iterate over
48     self.channel_node_indices = tuple(range(1, self.channel_n_level_nodes + 1))
49     self.channel_level_nodes = tuple(
50         "{}.H[{}]" .format(c, n)
51         for c, n in itertools.product(self.channels, self.channel_node_indices)
52     )
53     # Expand channel water level goals to all nodes
54     for channel in self.channels:
55         channel_max = self.get_timeseries(channel + "_max")
56         channel_min = self.get_timeseries(channel + "_min")
57         for i in self.channel_node_indices:
58             self.set_timeseries("{} .H[{}]_max" .format(channel, i), channel_max)
59             self.set_timeseries("{} .H[{}]_min" .format(channel, i), channel_min)
60     # Make input series appear in output csv
61     self.set_timeseries("Inflow_Q", self.get_timeseries("Inflow_Q"))
62     self.set_timeseries(
63         "DrinkingWaterExtractionPump_Q_target",
64         self.get_timeseries("DrinkingWaterExtractionPump_Q_target"),
65     )

```

Next, we instantiate the goals. The highest priority goal in this example will be to keep the water levels within a desired range. We apply this goal iteratively over all the water level states, and give them a priority of 1. The second goal is to track a target extraction flow rate with the extraction pump. We give this goal a priority of 2.

```

67 def path_goals(self):
68     g = super().path_goals()
69
70     # Add RangeGoal on water level states with a priority of 1
71     for node in self.channel_level_nodes:
72         g.append(RangeGoal(self, node, 1))
73
74     # Add TargetGoal on Extraction Pump with a priority of 2
75     g.append(TargetGoal(self, "DrinkingWaterExtractionPump_Q", 2))
76
77     return g

```

We want to apply these goals to every timestep, so we use the `path_goals()` method. This is a method that returns a list of the path goals we defined above. Note that with path goals, each timestep is implemented as an independent goal— if we cannot satisfy our min/max on time step A, it will not affect our desire to satisfy the goal at time step B. Goals that inherit `StateGoal` are always path goals.

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.

```
84 run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```
1 import itertools
2
3 from rtctools.optimization.collocated_integrated_optimization_problem import (
4     CollocatedIntegratedOptimizationProblem
5 )
6 from rtctools.optimization.csv_mixin import CSVMixin
7 from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin,
8     ↳ StateGoal
9 from rtctools.optimization.homotopy_mixin import HomotopyMixin
10 from rtctools.optimization.modelica_mixin import ModelicaMixin
11 from rtctools.util import run_optimization_problem
12
13 class RangeGoal(StateGoal):
14     def __init__(self, opt_prob, state, priority):
15         self.state = state
16         self.target_min = opt_prob.get_timeseries(state + "_min")
17         self.target_max = opt_prob.get_timeseries(state + "_max")
18         self.violation_timeseries_id = state + "_target_violation"
19         self.function_value_timeseries_id = state
20         self.priority = priority
21         super().__init__(opt_prob)
22
23
24 class TargetGoal(StateGoal):
25     def __init__(self, opt_prob, state, priority):
26         self.state = state
27         self.target_min = opt_prob.get_timeseries(state + "_target")
28         self.target_max = self.target_min
29         self.violation_timeseries_id = state + "_target_violation"
30         self.function_value_timeseries_id = state
31         self.priority = priority
32         super().__init__(opt_prob)
33
34
35 class Example(
36     HomotopyMixin,
37     GoalProgrammingMixin,
```

(continues on next page)

(continued from previous page)

```

38     CSVMixin,
39     ModelicaMixin,
40     CollocatedIntegratedOptimizationProblem,
41 ):
42     channels = "LowerChannel", "MiddleChannel", "UpperChannel"
43     channel_n_level_nodes = 2
44
45     def pre(self):
46         super().pre()
47         # Generate handy tuples to iterate over
48         self.channel_node_indices = tuple(range(1, self.channel_n_level_nodes + 1))
49         self.channel_level_nodes = tuple(
50             "{}.H[{}]" .format(c, n)
51             for c, n in itertools.product(self.channels, self.channel_node_indices)
52         )
53         # Expand channel water level goals to all nodes
54         for channel in self.channels:
55             channel_max = self.get_timeseries(channel + "_max")
56             channel_min = self.get_timeseries(channel + "_min")
57             for i in self.channel_node_indices:
58                 self.set_timeseries("{} .H[{}]_max" .format(channel, i), channel_max)
59                 self.set_timeseries("{} .H[{}]_min" .format(channel, i), channel_min)
60             # Make input series appear in output csv
61             self.set_timeseries("Inflow_Q", self.get_timeseries("Inflow_Q"))
62             self.set_timeseries(
63                 "DrinkingWaterExtractionPump_Q_target",
64                 self.get_timeseries("DrinkingWaterExtractionPump_Q_target"),
65             )
66
67     def path_goals(self):
68         g = super().path_goals()
69
70         # Add RangeGoal on water level states with a priority of 1
71         for node in self.channel_level_nodes:
72             g.append(RangeGoal(self, node, 1))
73
74         # Add TargetGoal on Extraction Pump with a priority of 2
75         g.append(TargetGoal(self, "DrinkingWaterExtractionPump_Q", 2))
76
77         return g
78
79     def post(self):
80         super().post()
81
82
83 # Run
84 run_optimization_problem(Example)

```

Extracting Results

The results from the run are found in `output/timeseries_export.csv`. Any CSV-reading software can import it, but this is how results can be plotted using the python library `matplotlib`:

1.4.2 Simulation examples

This section provides examples demonstrating key features of RTC-Tools simulation.

Tracking a Setpoint



Overview

The purpose of this example is to understand the technical setup of an RTC- Tools simulation model, how to run the model, and how to access the results.

The scenario is the following: A reservoir operator is trying to keep the reservoir's volume close to a given target volume. They are given a six-day forecast of inflows given in 12-hour increments. To keep things simple, we ignore the waterlevel-storage relation of the reservoir and head-discharge relationships in this example. To make things interesting, the reservoir operator is only able to release water at a few discrete flow rates, and only change the discrete flow rate every 12 hours. They have chosen to use the RTC- Tools simulator to see if a simple proportional controller will be able to keep the system close enough to the target water volume.

The folder `<examples directory>\simulation` contains a complete RTC-Tools simulation problem. An RTC-Tools directory has the following structure:

- `input`: This folder contains the model input data. These are several files in comma separated value format, `csv`.
- `model`: This folder contains the Modelica model. The Modelica model contains the physics of the RTC-Tools model.
- `output`: The folder where the output is saved in the file `timeseries_export.csv`.
- `src`: This folder contains a Python file. This file contains the configuration of the model and is used to run the model.

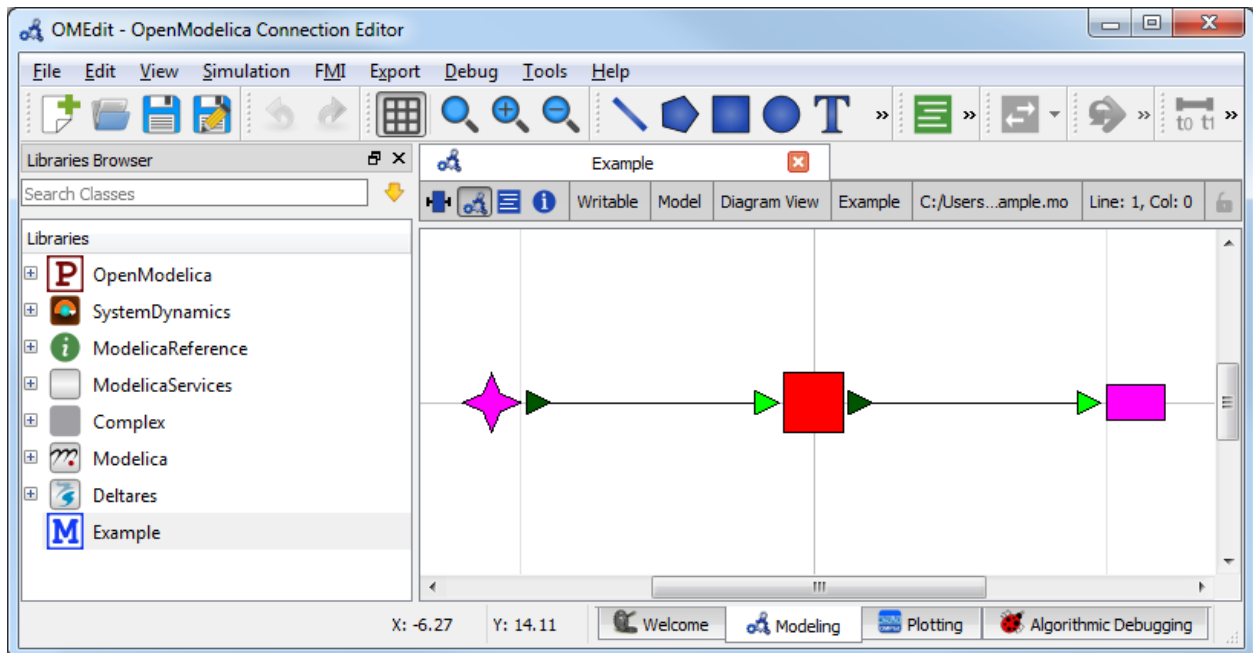
The Model

The first step is to develop a physical model of the system. The model can be viewed and edited using the OpenModelica Connection Editor (OMEdit) program. For how to download and start up OMEdit, see [Getting OMEdit](#).

Make sure to load the Deltares library before loading the example:

1. Load the Deltares library into OMEdit
 - Using the menu bar: *File -> Open Model/Library File(s)*
 - Select `<library directory>\Deltares\package.mo`
2. Load the example model into OMEdit
 - Using the menu bar: *File -> Open Model/Library File(s)*
 - Select `<examples directory>\simulation\model\Example.mo`

Once loaded, we have an OpenModelica Connection Editor window that looks like this:



The model `Example.mo` represents a simple system with the following elements:

- a reservoir, modeled as storage element `Deltares.ChannelFlow.SimpleRouting.Storage.Storage`,
- an inflow boundary condition `Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow`,
- an outfall boundary condition `Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Terminal`,
- connectors (black lines) connecting the elements.

You can use the mouse-over feature help to identify the predefined models from the Deltares library. You can also drag the elements around- the connectors will move with the elements. Adding new elements is easy- just drag them in from the Deltares Library on the sidebar. Connecting the elements is just as easy- click and drag between the ports on the elements.

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

```

1 model Example
2   // Elements
3   Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow inflow(Q = Q_in);
4   Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(Q_release = P_control,
5   ↪ V(start=storage_V_init, fixed=true, nominal=4e5));
6   Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Terminal outfall;
7   // Initial States
8   parameter Modelica.SIunits.Volume storage_V_init;
9   // Inputs
10  input Modelica.SIunits.VolumeFlowRate P_control(fixed = true);
11  input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
12  input Modelica.SIunits.VolumeFlowRate storage_V_target(fixed = true);
13  // Outputs
14  output Modelica.SIunits.Volume storage_V = storage.V;
15  output Modelica.SIunits.VolumeFlowRate Q_release = P_control;
16  equation
17    connect(inflow.QOut, storage.QIn);
18    connect(storage.QOut, outfall.QIn);
19 end Example;

```

The three water system elements (storage, inflow, and outfall) appear under the `model Example` statement. The `equation` part connects these three elements with the help of connections. Note that `storage` extends the partial model `QISO` which contains the connectors `QIn` and `QOut`. With `QISO`, `storage` can be connected on two sides. The `storage` element also has a variable `Q_release`, which is the decision variable the operator controls.

OpenModelica Connection Editor will automatically generate the element and connector entries in the text file. Defining inputs and outputs requires editing the text file directly and assigning the inputs to the appropriate element variables. For example, `inflow(Q = Q_in)` sets the `Q` variable of the `inflow` element equal to `Q_in`.

In addition to elements, the input variables `Q_in` and `P_control` are also defined. `Q_in` is determined by the forecast and the operator cannot control it, so we set `Q_in(fixed = true)`. The actual values of `Q_in` are stored in `timeseries_import.csv`. `P_control` is not defined anywhere in the model or inputs- we will dynamically assign its value every timestep in the python script, `\src\example.py`.

Because we want to view the water volume in the storage element in the output file, we also define an output variable `storage_V` and set it equal to the corresponding state variable `storage.V`. Dito for `Q_release = P_control`.

The Simulation Problem

The python script is created and edited in a text editor. In general, the python script consists of the following blocks:

- Import of packages
- Definition of the simulation problem class
 - Any additional configuration (e.g. overriding methods)
- A run statement

Importing Packages

Packages are imported using `from ... import ...` at the top of the file. In our script, we import the classes we want the class to inherit, the package `run_simulation_problem` from the `rtctools.util` package, and any extra packages we want to use. For this example, the import block looks like:


```
1 import logging
2
3 from rtctools.simulation.csv_mixin import CSVMixin
4 from rtctools.simulation.simulation_problem import SimulationProblem
5 from rtctools.util import run_simulation_problem
6
7 logger = logging.getLogger("rtctools")
8
```

Simulation Problem

The next step is to define the simulation problem class. We construct the class by declaring the class and inheriting the desired parent classes. The parent classes each perform different tasks related to importing and exporting data and running the simulation problem. Each imported class makes a set of methods available to the our simulation class.

```
10 class Example(CSVMixin, SimulationProblem):
```

The next, we override any methods where we want to specify non-default behaviour. In our simulation problem, we want to define a proportional controller. In its simplest form, we load the current values of the volume and target volume variables, calculate their difference, and set `P_control` to be as close as possible to eliminating that difference during the upcoming timestep.

```
24     def update(self, dt):
25
26         # Get the time step
27         if dt < 0:
28             dt = self.get_time_step()
29
30         # Get relevant model variables
31         volume = self.get_var('storage.V')
32         target = self.get_var('storage.V_target')
33
34         # Calucate error in storage.V
35         error = target - volume
36
37         # Calculate the desired control
38         control = -error / dt
39
40         # Get the closest feasible setting.
41         bounded_control = min(max(control, self.min_release), self.max_release)
42
43         # Set the control variable as the control for the next step of the simulation
44         self.set_var('P_control', bounded_control)
45
46         # Call the super class so that everything else continues as normal
47         super().update(dt)
```

Run the Simulation Problem

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

```
51 run_simulation_problem(Example, log_level=logging.DEBUG)
```


The Whole Script

All together, the whole example script is as follows:

```

1 import logging
2
3 from rtctools.simulation.csv_mixin import CSVMixin
4 from rtctools.simulation.simulation_problem import SimulationProblem
5 from rtctools.util import run_simulation_problem
6
7 logger = logging.getLogger("rtctools")
8
9
10 class Example(CSVMixin, SimulationProblem):
11     """
12     A basic example for introducing users to RTC-Tools 2 Simulation
13     """
14
15     def initialize(self):
16         self.set_var('P_control', 0.0)
17         super().initialize()
18
19     # Min and Max flow rate that the storage is capable of releasing
20     min_release, max_release = 0.0, 8.0 # m^3/s
21
22     # Here is an example of overriding the update() method to show how control
23     # can be build into the python script
24     def update(self, dt):
25
26         # Get the time step
27         if dt < 0:
28             dt = self.get_time_step()
29
30         # Get relevant model variables
31         volume = self.get_var('storage.V')
32         target = self.get_var('storage_V_target')
33
34         # Calucate error in storage.V
35         error = target - volume
36
37         # Calculate the desired control
38         control = -error / dt
39
40         # Get the closest feasible setting.
41         bounded_control = min(max(control, self.min_release), self.max_release)
42
43         # Set the control variable as the control for the next step of the simulation
44         self.set_var('P_control', bounded_control)
45
46         # Call the super class so that everything else continues as normal
47         super().update(dt)
48
49
50 # Run
51 run_simulation_problem(Example, log_level=logging.DEBUG)

```

Running RTC-Tools

To run this basic example in RTC-Tools, navigate to the basic example `src` directory in the RTC-Tools shell and run the example using `python example.py`. For more details about using RTC-Tools, see [Running RTC-Tools](#).

Extracting Results

The results from the run are found in `output\timeseries_export.csv`. Any CSV-reading software can import it. Here we used `matplotlib` to generate this plot.

Observations

This plot shows that the operator is not able to keep the water level within the bounds over the entire time horizon. They may need to increase the controller timestep, use a more complete PID controller, or use model predictive control such as the RTC-Tools optimization library to get the results they want.

CHAPTER 2

Indices and tables

- `genindex`
- `search`

Symbols

[__call__\(\)](#) (rtctools.optimization.csv_lookup_table_mixin.LookupTable method), 16
[__init__\(\)](#) (rtctools.optimization.goal_programming_mixin.StateGoal method), 21
[__init__\(\)](#) (rtctools.optimization.timeseries.Timeseries method), 6

B

[bounds\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 6

C

[CollocatedIntegratedOptimizationProblem](#) (class in rtctools.optimization.collocated_integrated_optimization_problem), 12
[constant_inputs\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 6
[constraints\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 6
[control\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 7
[control_at\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 7
[control_tree_options\(\)](#) (rtctools.optimization.control_tree_mixin.ControlTreeMixin method), 22
[ControlTreeMixin](#) (class in rtctools.optimization.control_tree_mixin), 22
[CSVLookupTableMixin](#) (class in rtctools.optimization.csv_lookup_table_mixin), 16
[CSVMixin](#) (class in rtctools.optimization.csv_mixin), 14
[CSVMixin](#) (class in rtctools.simulation.csv_mixin), 25

D

[delayed_feedback\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 7

[der\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 7
[der_at\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 7

E

[ensemble_member_probability\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 8
[ensemble_size\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem attribute), 8

F

[function\(\)](#) (rtctools.optimization.goal_programming_mixin.Goal method), 20

G

[get_current_time\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 23
[get_elapsed_time\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 23
[get_function_value\(\)](#) (rtctools.optimization.goal_programming_mixin.Goal method), 20
[get_start_time\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 23
[get_timeseries\(\)](#) (rtctools.optimization.optimization_problem.OptimizationProblem method), 8
[get_var\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 23
[get_var_count\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 23
[get_var_name\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 23
[get_var_rank\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 24
[get_var_shape\(\)](#) (rtctools.simulation.simulation_problem.SimulationProblem method), 24

get_var_type() (rtctools.simulation.simulation_problem.SimulationProblem method), 24

get_variables() (rtctools.simulation.simulation_problem.SimulationProblem method), 24

Goal (class in rtctools.optimization.goal_programming_mixin), 18

goal_programming_options() (rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 21

GoalProgrammingMixin (class in rtctools.optimization.goal_programming_mixin), 21

goals() (rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 22

has_target_bounds (rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin attribute), 20

has_target_max (rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin attribute), 20

has_target_min (rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin attribute), 20

history() (rtctools.optimization.optimization_problem.OptimizationProblem method), 8

homotopy_options() (rtctools.optimization.homotopy_mixin.HomotopyMixin method), 17

HomotopyMixin (class in rtctools.optimization.homotopy_mixin), 17

initial_state() (rtctools.optimization.optimization_problem.OptimizationProblem method), 8

initial_state_measurements() (rtctools.optimization.initial_state_estimation_mixin.InitialStateEstimationMixin method), 18

initial_state_smoothing_pairs() (rtctools.optimization.initial_state_estimation_mixin.InitialStateEstimationMixin method), 18

initial_time (rtctools.optimization.optimization_problem.OptimizationProblem attribute), 8

initialize() (rtctools.simulation.simulation_problem.SimulationProblem method), 24

InitialStateEstimationMixin (class in rtctools.optimization.initial_state_estimation_mixin), 17

integral() (rtctools.optimization.optimization_problem.OptimizationProblem method), 8

integrated_states (rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem attribute), 13

integrator_options() (rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem method), 13

lookup_tables() (rtctools.optimization.csv_lookup_table_mixin.CSVLookupTableMixin method), 17

lookup_tables() (rtctools.optimization.optimization_problem.OptimizationProblem method), 9

has_target_bounds (rtctools.optimization.csv_lookup_table_mixin.CSVLookupTableMixin attribute), 16

has_target_max (rtctools.optimization.csv_lookup_table_mixin.CSVLookupTableMixin attribute), 16

has_target_min (rtctools.optimization.csv_lookup_table_mixin.CSVLookupTableMixin attribute), 16

history() (rtctools.optimization.csv_mixin.CSVMixin method), 14

max_timeseries_id() (rtctools.optimization.csv_mixin.CSVMixin method), 14

min_timeseries_id() (rtctools.optimization.csv_mixin.CSVMixin method), 14

min_timeseries_id() (rtctools.optimization.pi_mixin.PIMixin method), 15

min_timeseries_id() (rtctools.optimization.pi_mixin.PIMixin method), 15

ModelicaMixin (class in rtctools.optimization.modelica_mixin), 14

objective() (rtctools.optimization.optimization_problem.OptimizationProblem method), 9

OptimizationProblem (class in rtctools.optimization.optimization_problem), 6

OptimizationProblem (class in rtctools.optimization.optimization_problem), 6

parameters() (rtctools.optimization.optimization_problem.OptimizationProblem method), 9

path_constraints() (rtctools.optimization.optimization_problem.OptimizationProblem method), 10

path_goals() (rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 22

path_objective (rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem attribute), 10

PIMixin (class in `rtctools.optimization.pi_mixin`), 14
PIMixin (class in `rtctools.simulation.pi_mixin`), 25
`post()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 10
`post()` (`rtctools.simulation.simulation_problem.SimulationProblem` method), 24
`pre()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 10
`pre()` (`rtctools.simulation.simulation_problem.SimulationProblem` method), 24
`priority_completed()` (`rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin` method), 22
`priority_started()` (`rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin` method), 22
`times()` (`rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem` method), 13
`timeseries_at()` (`rtctools.optimization.timeseries.Timeseries` attribute), 5
`timeseries_at()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 12
`timeseries_at()` (`rtctools.simulation.csv_mixin.CSVMixin` method), 25
`timeseries_at()` (`rtctools.simulation.pi_mixin.PIMixin` method), 25
`timeseries_export()` (`rtctools.optimization.pi_mixin.PIMixin` attribute), 15
`timeseries_import()` (`rtctools.optimization.pi_mixin.PIMixin` attribute), 15
`timeseries_in()` (`rtctools.optimization.pi_mixin.PIMixin` attribute), 15

R

`reset()` (`rtctools.simulation.simulation_problem.SimulationProblem` method), 24
`run_optimization_problem()` (in module `rtctools.util`), 12
`run_simulation_problem()` (in module `rtctools.util`), 24

U

`update()` (`rtctools.simulation.simulation_problem.SimulationProblem` method), 24

V

`values` (`rtctools.optimization.timeseries.Timeseries` attribute), 6

S

`seed()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 10
`set_timeseries()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 10
`setup_experiment()` (`rtctools.simulation.simulation_problem.SimulationProblem` method), 24
`simulate()` (`rtctools.simulation.simulation_problem.SimulationProblem` method), 24
`SimulationProblem` (class in `rtctools.simulation.simulation_problem`), 23
`solver_options()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 10
`solver_success()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 11
`state()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 11
`state_at()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 11
`StateGoal` (class in `rtctools.optimization.goal_programming_mixin`), 20
`states_in()` (`rtctools.optimization.optimization_problem.OptimizationProblem` method), 11

T

`theta` (`rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem` attribute), 13
`times` (`rtctools.optimization.timeseries.Timeseries` attribute), 6