Contents

1 Contents 3
1.1 Getting Started ................................................................. 3
1.2 Optimization .................................................................. 5
1.3 Simulation ...................................................................... 25
1.4 Examples ....................................................................... 28

2 Indices and tables 95

Index 97
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 successful, 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 successful, you should see that the solver succeeds:

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:
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](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

#### 1.2.1 Basics

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

Bases: `object`
Time series object, bundling time stamps with values.

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

**Parameters**

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

```python
times
values
```
Array of time stamps.
Array of values.

class rtctools.optimization.optimization_problem.OptimizationProblem(**kwargs)
```python
Bases: rtctools.data.storage.DataStoreAccessor
```
Base class for all optimization problems.

```python
```
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:

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

class rtctools.optimization.optimization_problem.OptimizationProblem(**kwargs)
```python
Bases: rtctools.data.storage.DataStoreAccessor
```
Base class for all optimization problems.

```python
constant_inputs(ensemble_member: int) → rtctools._internal.alias_tools.AliasDict[str, rtctools.optimization.timeseries.Timeseries][str, rtctools.optimization.timeseries.Timeseries]
```
Returns a dictionary of constant inputs.

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

**Returns** A dictionary of constant input names and time series.

```python
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:

```python
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),  
    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 \((x, y, \tau)\), 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) → rtc_tools._internal.alias_tools.AliasDict[str, rtc_tools.optimization.timeseries.Timeseries][str, rtc_tools.optimization.timeseries.Timeseries]  
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) → rtc_tools._internal.alias_tools.AliasDict[str, float][str, float]  
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[str, ForwardRef('LookupTable')][str, LookupTable]`

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:

```python
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[str, typing.Union[bool, int, float, casadi.casadi.MX]][str, Union[bool, int, float, casadi.casadi.MX]]`

Returns a dictionary of parameters.
**Parameters** `ensemble_member` – The ensemble member index.

**Returns** A dictionary of parameter names and values.

### path_constraints

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

```python
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

`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:

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

### post

`post() → None`

Postprocessing logic is performed here.

### pre

`pre() → None`

Preprocessing logic is performed here.

### seed

`seed(ensemble_member: int) → rtctools._internal.alias_tools.AliasDict[str, typing.Union[Float, rtctools.optimization.timeseries.Timeseries]][str, Union[float, rtctools.optimization.timeseries.Timeseries]]`

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.

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, bool]]

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 NotEnoughDegreesOfFreedom, 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`

```python
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`

```python
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`

```python
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`

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

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.
RTC-Tools Documentation, Release 2.4.0a2+18.g49ab02a

- **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

```python
class rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem
    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 \(\theta\)-method

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

  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.

1.2. Optimization 13
Warning: This is an experimental feature for $0 < \theta < 1$.

```python
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*.

```python
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

```python
class rtctools.optimization.csv_mixin.CSVMixin(**kwargs)
Bases: rtctools.optimization.io_mixin.IOMixin
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) \(\rightarrow\) str
Returns the name of the upper bound timeseries for the specified variable.
Parameters variable -- Variable name.
min_timeseries_id(variable: str) \(\rightarrow\) str
Returns the name of the lower bound timeseries for the specified variable.
Parameters variable -- Variable name.
```
1.2.5 Delft-FEWS I/O

PI-XML

class rtctools.optimization.pi_mixin.PIMixin(**kwargs)
    Bases: rtctools.optimization.io_mixin.IOMixin

    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.

NetCDF

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.

```python
linearization_parameters() → Dict[str, str]
```

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

### 1.2.7 Lookup tables

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

Bases: object

Lookup table.

```python
__call__(*args) → Union[float, numpy.ndarray, 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:

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

```python
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.

```python
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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>delta_theta_0</td>
<td>float</td>
<td>1.0</td>
</tr>
<tr>
<td>delta_theta_min</td>
<td>float</td>
<td>0.01</td>
</tr>
<tr>
<td>homotopy_parameter</td>
<td>string</td>
<td>theta</td>
</tr>
</tbody>
</table>

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 (1DV AR, 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 constrains 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.

**A goal can be written in vector form. In a vector goal:**

- The goal size determines how many goals there are.
- The goal function has shape *(goal size, 1).*
• The function is either minimized or has, possibly various, targets.
• Function nominal can either be an array with as many entries as the goal size or have a single value.
• Function ranges can either be an array with as many entries as the goal size or have a single value.
• In a goal, the target can either be an array with as many entries as the goal size or have a single value.
• In a path goal, the target can also be a Timeseries whose values are either a 1-dimensional vector or have as many columns as the goal size.

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

```python
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:

```python
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.

**critical = False**

Critical goals must always be fully satisfied.

**function** (optimization_problem: rtctools.optimization.optimization_problem.OptimizationProblem, ensemble_member: int) \rightarrow casadi.casadi.MX

This method returns a CasADi MX object describing the goal function.

**Returns** A CasADi MX object.

**function_nominal = 1.0**

Nominal value of function (used for scaling)

**function_range = (nan, nan)**

Range of goal function

**function_value_timeseries_id = None**

Timeseries ID for function value data (optional)

**get_function_key** (optimization_problem: rtctools.optimization.optimization_problem.OptimizationProblem, ensemble_member: int) \rightarrow 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.

order = 2
    The goal violation value is taken to the order’th power in the objective function.

priority = 1
    Lower priority goals take precedence over higher priority goals.

relaxation = 0.0
    Absolute relaxation applied to the optimized values of this goal

size = 1
    The size of the goal if it’s a vector goal.

target_max = nan
    Desired upper bound for goal function

target_min = nan
    Desired lower bound for goal function

violation_timeseries_id = None
    Timeseries ID for goal violation data (optional)

weight = 1.0
    Goals with the same priority are weighted off against each other in a single objective function.

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:

```python
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.

```python
__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.

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>violation_relaxation</td>
<td>float</td>
<td>0.0</td>
</tr>
<tr>
<td>constraint_relaxation</td>
<td>float</td>
<td>0.0</td>
</tr>
<tr>
<td>mu_reinit</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>fix_minimized_values</td>
<td>bool</td>
<td>True/False</td>
</tr>
<tr>
<td>check_monotonicity</td>
<td>bool</td>
<td>True</td>
</tr>
<tr>
<td>equality_threshold</td>
<td>float</td>
<td>1e-8</td>
</tr>
<tr>
<td>interior_distance</td>
<td>float</td>
<td>1e-6</td>
</tr>
<tr>
<td>scale_by_problem_size</td>
<td>bool</td>
<td>False</td>
</tr>
<tr>
<td>keep_soft_constraints</td>
<td>bool</td>
<td>False</td>
</tr>
</tbody>
</table>

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 active time steps (per goal). 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.

```py
goals() → List[rtctools.optimization.goal_programming_mixin.Goal]
```
User problem returns list of `Goal` objects.

**Returns** A list of goals.

```py
path_goals() → List[rtctools.optimization.goal_programming_mixin.Goal]
```
User problem returns list of path `Goal` objects.

**Returns** A list of path goals.

```py
priority_completed(priority: int) → None
```
Called after optimization for goals of certain priority is completed.

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

```py
priority_started(priority: int) → None
```
Called when optimization for goals of certain priority is started.

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

### Minimize absolute value

```py
class rtctools.optimization.min_abs_goal_programming_mixin.MinAbsGoalProgrammingMixin(*args, **kwargs)
```

**Bases:** `rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin`

Similar behavior to `GoalProgrammingMixin`, but any `MinAbsGoal` passed to `min_abs_goals()` or `min_abs_path_goals()` will be automatically converted to:

1. An auxiliary minimization variable
2. Two additional linear constraints relating the auxiliary variable to the goal function
3. A new goal (of a different type) minimizing the auxiliary variable
min_abs_goals() \rightarrow \text{List}[\text{rtctools.optimization.min_abs_goal_programming_mixin.MinAbsGoal}]

User problem returns list of MinAbsGoal objects.

\textbf{Returns} A list of goals.

min_abs_path_goals() \rightarrow \text{List}[\text{rtctools.optimization.min_abs_goal_programming_mixin.MinAbsGoal}]

User problem returns list of MinAbsGoal objects.

\textbf{Returns} A list of goals.

class rtctools.optimization.min_abs_goal_programming_mixin.MinAbsGoal

\textbf{Bases:} rtctools.optimization.goal_programming_mixin.Goal

Absolute minimization goal class which can be used to minimize the absolute value of the goal’s (linear) goal function. Contrary to its super class, the default order is 1 as absolute minimization is typically desired for fully linear problems.

class rtctools.optimization.min_abs_goal_programming_mixin.MinAbsStateGoal(optimization_problem)

\textbf{Bases:} rtctools.optimization.goal_programming_mixin.StateGoal, rtctools.optimization.min_abs_goal_programming_mixin.MinAbsGoal

\textbf{__init__}(optimization_problem)

Initialize the state goal object.

\textbf{Parameters} optimization_problem -- OptimizationProblem instance.

Linearized order

class rtctools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderGoalProgrammingMixin

\textbf{Bases:} rtctools.optimization.goal_programming_mixin.GoalProgrammingMixin

Adds support for linearization of the goal objective functions, i.e. the violation variables to a certain power. This can be used to keep a problem fully linear and/or make sure that no quadratic constraints appear when using the goal programming option \texttt{keep_soft_constraints}.

\textbf{goal_programming_options()}

If \texttt{linearize_goal_order} is set to True, the goal’s order will be approximated linearly for any goals where order > 1. Note that this option does not work with minimization goals of higher order. Instead, it is suggested to transform these minimization goals into goals with a target (and function range) when using this option. Note that this option can be overridden on the level of a goal by using a \texttt{LinearizedOrderGoal} (see \texttt{LinearizedOrderGoal.linearize_order}).

class rtctools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderGoal

\textbf{Bases:} rtctools.optimization.goal_programming_mixin.Goal

\textbf{linearize_order = None}

Override linearization of goal order. Related global goal programming option is \texttt{linearize_goal_order} (see \texttt{LinearizedOrderGoalProgrammingMixin.goal_programming_options()}). The default value of None defers to the global option, but the user can explicitly override it per goal by setting this value to True or False.

class rtctools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderStateGoal(optimization_problem)

\textbf{Bases:} rtctools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderGoal, rtctools.optimization.goal_programming_mixin.StateGoal

Convenience class definition for linearized order state goals. Note that it is possible to just inherit from \texttt{LinearizedOrderGoal} to get the needed functionality for control of the linearization at goal level.

\textbf{__init__}(optimization_problem)

Initialize the state goal object.
Parameters \texttt{optimization\_problem} – OptimizationProblem instance.

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.

    \texttt{control\_tree\_options()} \rightarrow \text{Dict[str, Union[List[str], List[float], int]]}

    Returns a dictionary of options controlling the creation of a \textit{k}-ary stochastic tree.

    \begin{tabular}{|l|l|l|}
    \hline
    \textbf{Option} & \textbf{Type} & \textbf{Default value} \\
    \hline
    forecast\_variables & list of strings & All constant inputs \\
    branching\_times   & list of floats & self.times() \\
    \textit{k}         & int           & 2 \\
    \hline
    \end{tabular}

    A \textit{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.

    \textbf{Returns} A dictionary of control tree generation options.

1.3 Simulation

\textbf{Note:} For a simulation example, see \textit{Simulation examples}

Contents:

1.3.1 Basics

class rtctools.simulation.simulation\_problem.SimulationProblem(**kwargs)
    Bases: rtctools.data.storage.DataStoreAccessor

    Implements the BMI Interface.

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

    \textbf{Variables} \texttt{modelica\_library\_folders} – Folders containing any referenced Modelica libraries. Default is an empty list.

    \texttt{get\_current\_time()}  
    Return current time of simulation.

    \textbf{Returns} The current simulation time.

    \texttt{get\_end\_time()}  
    Return end time of experiment.

    \textbf{Returns} The end time of the experiment.

    \texttt{get\_start\_time()}  
    Return start time of experiment.

    \textbf{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.

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

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.io_mixin.IOMixin

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`.

```python
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

### 1.3.3 Delft-FEWS I/O

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

Bases: rtctools.simulation.io_mixin.IOMixin

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:
  Storage,
  Inflow,
  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):

```modelica
model Example
    Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(V(nominal=4e5, min=2e5, max=6e5));
    input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
    input Modelica.SIunits.VolumeFlowRate Q_release(fixed = false, min = 0.0, max = 6.);
```
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 QSISO which contains the connectors QIn and QOut. With QSISO, 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 form the rtctools.util package, and any extra packages we want to use. For this example, the import block looks like:

```python
from rtctools.optimization.collocated_integrated_optimization_problem \
  import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
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.

```python
class Example(C SVMixin, ModelicaM i xin, CollocatedIntegratedOptimizationProblem):
```

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

```python
def objective(self, ensemble_member):
    # Minimize water pumped. The total water pumped is the integral of the
    # water pumped from the starting time until the stoping time. In
    # practice, self.integral() is a summation of all the discrete states.
    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.

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

```python
run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```python
from rtctools.optimization.collocated_integrated_optimization_problem \
import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

class Example(C SVMixin, ModelicaM i xin, CollocatedIntegratedOptimizationProblem):
    """
    A basic example for introducing users to RTC-Tools 2
    """
    def objective(self, ensemble_member):
```
# Minimize water pumped. The total water pumped is the integral of the
# water pumped from the starting time until the stoping time. In
# practice, self.integral() is a summation of all the discrete states.
return self.integral('Q_release', ensemble_member)

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

# Run
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 im-
port it, but this is what the results look like when plotted with the python library matplotlib:

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 nlpsol 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 pump \texttt{Deltares.ChannelFlow.Hydraulic.Structures.Pump}
• an orifice modeled as a pump \texttt{Deltares.ChannelFlow.Hydraulic.Structures.Pump}

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

```modelica
model Example

// Declare Model Elements
Deltares.ChannelFlow.Hydraulic.Storage.Linear storage(A=1.0e6, H_b=0.0, HQ.H(min=0.0, max=0.5));

// Define Input/Output Variables and set them equal to model variables
input Modelica.SIunits.VolumeFlowRate Q_pump(fixed=false, min=0.0, max=7.0) = pump.Q;
inout Boolean is_downhill;
inout Modelica.SIunits.VolumeFlowRate Q_in(fixed=true) = discharge.Q;
in Modelica.SIunits.Position H_sea(fixed=true) = level.H;
input Modelica.SIunits.VolumeFlowRate Q_orifice(fixed=false, min=0.0, max=10.0) = orifice.Q;
output Modelica.SIunits.Position storage_level = storage.HQ.H;
output Modelica.SIunits.Position sea_level = level.H;

// Connect Model Elements
connect(orifice.HQDown, level.HQ);
connect(storage.HQ, orifice.HQUp);
connect(storage.HQ, pump.HQUp);
connect(discharge.HQ, storage.HQ);
connect(pump.HQDown, level.HQ);
end Example;
```

1.4. Examples
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 \( Q_{\text{in}}, H_{\text{sea}}, Q_{\text{pump}}, \) and \( Q_{\text{orifice}} \) 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:

```
import numpy as np
from rtctools.optimization.collocated_integrated_optimization_problem \\n    import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
```

Note that we are also importing \( \text{inf} \) 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.

```
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:
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.

```
def path_constraints(self, ensemble_member):
    # Call super to get default constraints
    constraints = super().path_constraints(ensemble_member)
    M = 2  # The so-called "big-M"

    # Release through orifice downhill only. This constraint enforces the
    # fact that water only flows downhill.
    constraints.append((self.state('Q_orifice') + (1 - self.state('is_downhill')) * 10,
                        0.0, 10.0))

    # Make sure is_downhill is true only when the sea is lower than the
    # water level in the storage.
    constraints.append((self.state('H_sea') - self.state('storage.HQ.H') -
                        (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
    constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
                        self.state('is_downhill') * M, 0.0, np.inf))

    # Orifice flow constraint. Uses the equation:
    # \( Q(\text{HUp}, \text{HDown}, d) = w \times C \times d \times (2 \times g \times (\text{HUp} - \text{HDown}))^{0.5} \)
    # Note that this equation is only valid for orifices that are submerged
    # units: description:
    # \( w = 3.0 \) m width of orifice
    # \( d = 0.8 \) m height of orifice
    # \( C = 1.0 \) none orifice constant
    # \( g = 9.8 \) m/s^2 gravitational acceleration
    constraints.append(((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) +
                        self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -
                        M * (1 - self.state('is_downhill')),
                        -np.inf, 0.0))

    return constraints
```

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

```
def solver_options(self):
    options = super().solver_options()
    # Restrict solver output
    solver = options['solver']
    options[solver]['print_level'] = 1
    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.

```python
run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```python
import numpy as np

from rtctools.optimization.collocated_integrated_optimization_problem import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

class Example(CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
    ""
    This class is the optimization problem for the Example. Within this class,
    the objective, constraints and other options are defined.
    ""

    # This is a method that returns an expression for the objective function.
    # RTC-Tools always minimizes the objective.
    def objective(self, ensemble_member):
        # Minimize water pumped. The total water pumped is the integral of the
        # water pumped from the starting time until the stoping time. In
        # practice, self.integral() is a summation of all the discrete states.
        return self.integral('Q_pump', ensemble_member)

    # A path constraint is a constraint where the values in the constraint are a
    # Timeseries rather than a single number.
    def path_constraints(self, ensemble_member):
        # Call super to get default constraints
        constraints = super().path_constraints(ensemble_member)
        M = 2  # The so-called "big-M"

        # Release through orifice downhill only. This constraint enforces the
        # fact that water only flows downhill.
        constraints.append((self.state('Q_orifice') + (1 - self.state('is_downhill')) * 10,
                           0.0, 10.0))

        # Make sure is_downhill is true only when the sea is lower than the
        # water level in the storage.
        constraints.append((self.state('H_sea') - self.state('storage.HQ.H') -
                            (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
        constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
                            self.state('is_downhill') * M, 0.0, np.inf))

        # Orifice flow constraint. Uses the equation:
        # Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ 0.5
```

(continues on next page)
# Note that this equation is only valid for orifices that are submerged
# units: description:

\[
\begin{align*}
    w &= 3.0 \text{ m width of orifice} \\
    d &= 0.8 \text{ m height of orifice} \\
    C &= 1.0 \text{ orifice constant} \\
    g &= 9.8 \text{ m/s}^2 \text{ gravitational acceleration}
\end{align*}
\]

```
c = np.array([((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) +
               self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -
               M * (1 - self.state('is_downhill')),
               -np.inf, 0.0))
```

```python
def solver_options(self):
    options = super().solver_options()
    # Restrict solver output
    solver = options['solver']
    options[solver]['print_level'] = 1
    return options
```

```python
# Run
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:

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

# Import Data
data_path = "../../../examples/mixed_integer/output/timeseries_export.csv"
results = np.recfromcsv(data_path, encoding=None)

# Get times as datetime objects
times = list(map(lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S"), results["time"]))
```

(continues on next page)
Computational Example

```python
fig, 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(times, results["storage_level"], label="Storage", linewidth=2, color="b")
axarr[0].plot(times, results["sea_level"], label="Sea", linewidth=2, color="m")
axarr[0].plot(tiles, 0.5 * np.ones_like(tiles), label="Storage Max", linewidth=2, color="r", linestyle="--")

# Lower Subplot
axarr[1].set_ylabel("Flow Rate [m$^3$/s]")
axarr[1].plot(times, results["q_orifice"], label="Orifice", linewidth=2, color="g")
axarr[1].plot(times, results["q_pump"], label="Pump", linewidth=2, color="r")

# Format bottom axis label
axarr[-1].xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))

# Shrink margins
fig.tight_layout()

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):
model Example
  // Declare Model Elements
  Deltares.ChannelFlow.Hydraulic.Storage.Linear storage(A=1.0e6, H_b=0.0, HQ.H(min=0.0, max=0.5));
  
  // Define Input/Output Variables and set them equal to model variables
  input Modelica.SIunits.VolumeFlowRate Q_pump(fixed=false, min=0.0, max=7.0) = pump.Q;
  input Boolean is_downhill;
  input Modelica.SIunits.VolumeFlowRate Q_in(fixed=true) = discharge.Q;
  input Modelica.SIunits.Position H_sea(fixed=true) = level.H;
  input Modelica.SIunits.VolumeFlowRate Q_orifice(fixed=false, min=0.0, max=10.0) = orifice.Q;
  output Modelica.SIunits.Position storage_level = storage.HQ.H;
  output Modelica.SIunits.Position sea_level = level.H;
  equation
    // Connect Model Elements
    connect (orifice.HQDown, level.HQ);
    connect (storage.HQ, orifice.HQUp);
    connect (storage.HQ, pump.HQUp);
    connect (discharge.HQ, storage.HQ);
    connect (pump.HQDown, level.HQ);
  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:
import numpy as np

from rtctools.optimization.collocated_integrated_optimization_problem \ import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin \ import Goal, GoalProgrammingMixin, StateGoal
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:

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

class MinimizeQpumpGoal(Goal):
    # This goal does not use a helper class, so we have to define the function
    # method, range and nominal explicitly. We do not specify a target_min or
    # target_max in this class, so the goal programming mixin will try to
    # minimize the expression returned by the function method.
    def function(self, optimization_problem, ensemble_member):
        return optimization_problem.integral('Q_pump')

    # The nominal is used to scale the value returned by
    # the function method so that the value is on the order of 1.
    function_nominal = 100.0
    # The lower the number returned by this function, the higher the priority.
    priority = 2
    # The penalty variable is taken to the order'th power.
    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.

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

**Optimization Problem**

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

```python
class Example(GoalProgrammingMixin, CSVMixin, ModelicaMixin,
              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.

```python
def path_constraints(self, ensemble_member):
    # We want to add a few hard constraints to our problem. The goal
    # programming mixin however also generates constraints (and objectives)
    # from on our goals, so we have to call super() here.
    constraints = super().path_constraints(ensemble_member)

    # Release through orifice downhill only. This constraint enforces the
    # fact that water only flows downhill
    constraints.append((self.state('Q_orifice') +
                       (1 - self.state('is_downhill')) * 10, 0.0, 10.0))

    # Make sure is_downhill is true only when the sea is lower than the
    # water level in the storage.
    M = 2  # The so-called "big-M"
    constraints.append(((self.state('H_sea') - self.state('storage.HQ.H') -
                        (1 - self.state('is_downhill'))) * M, -np.inf, 0.0))
    constraints.append((self.state('H_sea') - self.state('storage.HQ.H') +
                        self.state('is_downhill') * M, 0.0, np.inf))

    # Orifice flow constraint. Uses the equation:
    # Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ 0.5
    # Note that this equation is only valid for orifices that are submerged
    # units: description:
    w = 3.0  # m width of orifice
    d = 0.8  # m height of orifice
    C = 1.0  # none orifice constant
    g = 9.8  # m/s^2 gravitational acceleration
```

(continues on next page)
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.

```
def goals(self):
    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 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 and must always be initialized with the parameter self.

```
def path_goals(self):
    # Sorting goals on priority is done in the goal programming mixin. We
do not have to worry about order here.
    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:

```
def pre(self):
    # Call super() class to not overwrite default behaviour
    super().pre()
    # We keep track of our intermediate results, so that we can print some
    # information about the progress of goals at the end of our run.
    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.

```
def priority_completed(self, priority):
    # We want to show that the results of our highest priority goal (water
    # level) are remembered. The other information we want to see is how our
    # lower priority goal (Q_pump) progresses. We can write some code that
    # sumerizes the results and stores it.

    # A little bit of tolerance when checking for acceptance, because
    # strictly speaking 0.4299... is smaller than 0.43.
    _min = 0.43 - 1e-4
    _max = 0.44 + 1e-4
```
results = self.extract_results()

n_level_satisfied = sum(
    1 for x in results['storage.HQ.H'] if _min <= x <= _max)

q_pump_integral = sum(results['Q_pump'])

q_pump_sum_changes = np.sum(np.diff(results['Q_pump'])**2)

self.intermediate_results.append(
    (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.

```python
def post(self):
    # Call super() class to not overwrite default behaviour
    super().post()

    for priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes in self.intermediate_results:
        print('After finishing goals of priority {}:
            Level goal satisfied at {} of {} time steps'.format(
                priority, n_level_satisfied, len(self.times())))
        print('Integral of Q_pump = {:.2f}'.format(q_pump_integral))
        print('Sum of squares of changes in Q_pump: {:.2f}'.format(q_pump_sum_changes))
```

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

```python
def solver_options(self):
    options = super().solver_options()
    solver = options['solver']

    options[solver]['print_level'] = 1

    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.

```python
run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```python
import numpy as np

from rtctools.optimization.collocated_integrated_optimization_problem \   import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin \   import Goal, GoalProgrammingMixin, StateGoal
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem
```
class WaterLevelRangeGoal(StateGoal):
    # Applying a state goal to every time step is easily done by defining a goal
    # that inherits StateGoal. StateGoal is a helper class that uses the state
    # to determine the function, function range, and function nominal
    # automatically.
    state = 'storage.HQ.H'
    # One goal can introduce a single or two constraints (min and/or max). Our
    # target water level range is 0.43 - 0.44. We might not always be able to
    # realize this, but we want to try.
    target_min = 0.43
    target_max = 0.44
    # Because we want to satisfy our water level target first, this has a
    # higher priority (=lower number).
    priority = 1

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

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

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

    ""
    An introductory example to goal programming in RTC-Tools
    ""
    def path_constraints(self, ensemble_member):
        # We want to add a few hard constraints to our problem. The goal
        # programming mixin however also generates constraints (and objectives)
        # from on our goals, so we have to call super() here.
constraints = super().path_constraints(ensemble_member)

# Release through orifice downhill only. This constraint enforces the 
# fact that water only flows downhill
constraints.append((self.state('Q_orifice') + 
    (1 - self.state('is_downhill')) * 10, 0.0, 10.0))

# Make sure is_downhill is true only when the sea is lower than the 
# water level in the storage.
M = 2 # The so-called "big-M"
constraints.append((self.state('H_sea') - self.state('storage.HQ.H') - 
    (1 - self.state('is_downhill')) * M, -np.inf, 0.0))
constraints.append((self.state('H_sea') - self.state('storage.HQ.H') + 
    self.state('is_downhill') * M, 0.0, np.inf))

# Orifice flow constraint. Uses the equation:
# Q(HUp, HDown, d) = width * C * d * (2 * g * (HUp - HDown)) ^ 0.5
# Note that this equation is only valid for orifices that are submerged
# units: description:
w = 3.0 # m width of orifice
d = 0.8 # m height of orifice
C = 1.0 # none orifice constant
g = 9.8 # m/s^2 gravitational acceleration
constraints.append(
    (((self.state('Q_orifice') / (w * C * d)) ** 2) / (2 * g) + 
    self.state('orifice.HQDown.H') - self.state('orifice.HQUp.H') -
    M * (1 - self.state('is_downhill'))),
    -np.inf, 0.0))

return constraints

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

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

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

def priority_completed(self, priority):
    # We want to show that the results of our highest priority goal (water
    # level) are remembered. The other information we want to see is how our
    # lower priority goal (Q_pump) progresses. We can write some code that
    # summarizes the results and stores it.

    # A little bit of tolerance when checking for acceptance, because
    # strictly speaking 0.4299... is smaller than 0.43.
    _min = 0.43 - 1e-4
    _max = 0.44 + 1e-4
results = self.extract_results()

n_level_satisfied = sum(
    1 for x in results["storage.HQ.H"] if _min <= x <= _max)

q_pump_integral = sum(results["Q_pump"])

q_pump_sum_changes = np.sum(np.diff(results["Q_pump"])**2)

self.intermediate_results.append(
    (priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes))

def post(self):
    # Call super() class to not overwrite default behaviour
    super().post()

    for priority, n_level_satisfied, q_pump_integral, q_pump_sum_changes:
        print('After finishing goals of priority {}:'.format(priority))
        print('Level goal satisfied at {} of {} time steps'.format(
            n_level_satisfied, len(self.times())))
        print('Integral of Q_pump = {:.2f}'.format(q_pump_integral))
        print('Sum of squares of changes in Q_pump: {:.2f}'.format(q_pump_sum_ changes))

    # Any solver options can be set here
    def solver_options(self):
        options = super().solver_options()
        solver = options["solver"]
        options[solver]["print_level"] = 1
        return options

    # Run
    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.

1.4. Examples
After optimizing the priority 3 goal, the integral of $Q_{\text{pump}}$ 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:

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

# Import Data
data_path = "../../../examples/goal_programming/output/timeseries_export.csv"
results = np.recfromcsv(data_path, encoding=None)

# Get times as datetime objects
times = list(map(lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S"), results["time "]

# Generate Plot
n_subplots = 3
fig, 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(times, results["storage_level"], label="Storage", linewidth=2, color="b 

# Middle subplot
axarr[1].set_ylabel("Water Level [m]")
axarr[1].plot(times, results["storage_level"], label="Storage", linewidth=2, color="b 

# Lower Subplot
axarr[2].plot(
times,
0.44 * np.ones_like(times),
label="Storage Max",
linestyle="--",

axarr[2].plot(
times,
0.43 * np.ones_like(times),
label="Storage Min",
linestyle="--",

axarr[2].plot(
...)
```

(continues on next page)
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):

```modelica
model Example
  Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(V(nominal=4e5, min=2e5, max=6e5));
  input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
  input Modelica.SIunits.VolumeFlowRate Q_release(fixed = false, min = 0.0, max = 10.0);
  equation
    connect(inflow.QOut, storage.QIn);
    connect(storage.QOut, outfall.QIn);
    storage.Q_release = Q_release;
    inflow.Q = Q_in;
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:

```python
import numpy as np
from rtctools.optimization.collocated_integrated_optimization_problem \
    import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixIn
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin \
    import GoalProgrammingMixin, StateGoal
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.

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

We also want to save energy, so we define a goal to minimize \( Q_{\text{release}} \). This goal has a lower priority.

```python
class MinimizeQreleaseGoal(StateGoal):
    # GoalProgrammingMixin will try to minimize the following state:
    state = 'Q_release'
    # The lower the number returned by this function, the higher the priority.
    priority = 2
    # The penalty variable is taken to the order'th power.
    order = 1
```
Optimization Problem

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

```python
37 class Example (GoalProgrammingMixin, CSVLookupTableMixin, CSVMixin,
                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 range 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.

```python
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')))
```

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.

```python
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.
```python
def priority_completed(self, priority):
    results = self.extract_results()
    self.set_timeseries('storage_V', results['storage.V'])
    _max = self.get_timeseries('V_max').values
    _min = self.get_timeseries('V_min').values
    storage_V = self.get_timeseries('storage_V').values

    # A little bit of tolerance when checking for acceptance.
    tol = 10
    _max += tol
    _min -= tol
    n_level_satisfied = sum(np.logical_and(_min <= storage_V, storage_V <= _max))
    q_release_integral = sum(results['Q_release'])
    self.intermediate_results.append((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.

```python
def post(self):
    # Call super() class to not overwrite default behaviour
    super().post()
    for priority, n_level_satisfied, q_release_integral in self.intermediate_results:
        print("After finishing goals of priority {}:
            Volume goal satisfied at {} of {} time steps
            Integral of Q_release = {:.2f}".format(priority, n_level_satisfied, len(self.times()), q_release_integral))
```

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

```python
def solver_options(self):
    options = super().solver_options()
    solver = options['solver']
    options[solver]['print_level'] = 1
    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.

```python
run_optimization_problem(Example)
```

**The Whole Script**

All together, the whole example script is as follows:

```python
import numpy as np
from rtctools.optimization.collocated_integrated_optimization_problem
```

(continues on next page)
import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin, StateGoal
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

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

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

class Example(GoalProgrammingMixin, CSVLookupTableMixin, CSVMixin,
               ModelicaMixin, CollocatedIntegratedOptimizationProblem):
    ""
    An extention of the goal programming example that shows how to incorporate
    non-linear storage elements in the model.
    ""
    def pre(self):
        super().pre()
        # Empty list for storing intermediate_results
        self.intermediate_results = []

        # Cache lookup tables for convenience and 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 like this:
        self.set_timeseries('H_min', np.ones_like(self.times()) * 0.44, output=False)

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

        # Convert our water level constraints into volume constraints
(continues on next page)
self.set_timeseries('V_max',
    self.lookup_storage_V(self.get_timeseries('H_max')))
self.set_timeseries('V_min',
    self.lookup_storage_V(self.get_timeseries('H_min')))

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

# We want to print some information about our goal programming problem. We # store the useful numbers temporarily, and print information at the end of # our run (see post() method below).
def priority_completed(self, priority):
    results = self.extract_results()
    self.set_timeseries('storage_V', results['storage.V'])

    _max = self.get_timeseries('V_max').values
    _min = self.get_timeseries('V_min').values
    storage_V = self.get_timeseries('storage_V').values

    # A little bit of tolerance when checking for acceptance.
tol = 10
    _max += tol
    _min -= tol
    n_level_satisfied = sum(np.logical_and(_min <= storage_V, storage_V <= _max))
    q_release_integral = sum(results['Q_release'])
    self.intermediate_results.append((priority, n_level_satisfied, q_release_integral))

def post(self):
    # Call super() class to not overwrite default behaviour
    super().post()
    for priority, n_level_satisfied, q_release_integral in self.intermediate_results:
        print("\nAfter finishing goals of priority {0}: ".format(priority))
        print("Volume goal satisfied at {} of {} time steps ".format( n_level_satisfied, len(self.times())))
        print("Integral of Q_release = {:.2f}").format(q_release_integral))

    # Any solver options can be set here
    def solver_options(self):
        options = super().solver_options()
        solver = options['solver']
        options[solver]['print_level'] = 1
        return options

    # Run
    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 $Q_{\text{release}} = 42.69$

After finishing goals of priority 2:
Volume goal satisfied at 12 of 12 time steps
Integral of $Q_{\text{release}} = 42.58$

As the output indicates, while optimizing for the priority 1 goal, no attempt was made to minimize the integral of $Q_{\text{release}}$. 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 $Q_{\text{release}}$ 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:

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

# Import Data
data_path = "../../../examples/lookup_table/output/timeseries_export.csv"
results = np.recfromcsv(data_path, encoding=None)

# Get times as datetime objects
times = list(map(
    lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S"),
    results["time"]
))

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

# Upper subplot
axarr[0].set_ylabel("Water Volume \[m\^3\]"
axarr[0].ticklabel_format(style="sci", axis="y", scilimits=(0, 0))
axarr[0].plot(times, results["storage_v"], label="Storage", linewidth=2, color="b")
axarr[0].plot(  
    times, results["v_max"], label="Storage Max", linewidth=2, color="r", linestyle="--"  
)
axarr[0].plot(  
    times, results["v_min"], label="Storage Min", linewidth=2, color="g", linestyle="--"  
)

# Lower Subplot
axarr[1].set_ylabel("Flow Rate \[m^3/s\]"
axarr[1].plot(times, results["q_in"], label="Inflow", linewidth=2, color="g")
axarr[1].plot(times, results["q_release"], label="Release", linewidth=2, color="r")
```

(continues on next page)
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**
    - **timeseries_import.csv**
    - **initial_state.csv**
  - **forecast2**
    - **timeseries_import.csv**
    - **initial_state.csv**
  - ...
- **output**: The folder where the output is saved:
  - **forecast1**
    - **timeseries_export.csv**
  - **forecast2**
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:

![OpenModelica Connection Editor](image)

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

```modelica
model Example
  Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(V(nominal=4e5, min=2e5, max=6e5));
  input Modelica.SIunits.VolumeFlowRate Q_in(fixed = true);
  input Modelica.SIunits.VolumeFlowRate Q_release(fixed = false, min = 0.0, max = 6.0);
  equation
    connect(inflow.QOut, storage.QIn);
    connect(storage.QOut, outfall.QIn);
    storage.Q_release = Q_release;
    inflow.Q = Q_in;
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:

```python
import numpy as np
from rtctools.optimization.collocated_integrated_optimization_problem \ 
    import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.control_tree_mixin import ControlTreeMixin
from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin \ 
    import GoalProgrammingMixin, StateGoal
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.

```python
class WaterVolumeRangeGoal(StateGoal):
    def __init__(self, optimization_problem):
        # Assign V_min and V_max the the target range
        self.target_min = optimization_problem.get_timeseries('V_min')
        self.target_max = optimization_problem.get_timeseries('V_max')
        super().__init__(optimization_problem)
        state = 'storage.V'
        priority = 1
```

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

```python
class MinimizeQreleaseGoal(StateGoal):
    # GoalProgrammingMixin will try to minimize the following state
    state = 'Q_release'
    # The lower the number returned by this function, the higher the priority.
    priority = 2
    # The penalty variable is taken to the order'th power.
    order = 1
```
Optimization Problem

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

```python
class Example(GoalProgrammingMixin, CSVMixin, CSVLookupTableMixin, ModelicaMixin,
              ControlTreeMixin, CollocatedIntegratedOptimizationProblem):
```

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

```python
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.

```python
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.

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

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

def priority_completed(self, priority):
    # We want to print some information about our goal programming problem.
    # We store the useful numbers temporarily, and print information at the
    # end of our run.
    for e_m in range(self.ensemble_size):
        results = self.extract_results(e_m)
        self.set_timeseries('V_storage', results['storage.V'], ensemble_member=e_m)
        _max = self.get_timeseries('V_max', ensemble_member=e_m).values
        _min = self.get_timeseries('V_min', ensemble_member=e_m).values
        V_storage = self.get_timeseries('V_storage', ensemble_member=e_m).values

        # A little bit of tolerance when checking for acceptance. This
        # tolerance must be set greater than the tolerance of the solver.
        tol = 10
        _max = _max + tol
        _min = _min - tol
        n_level_satisfied = sum(np.logical_and(_min <= V_storage, V_storage <= _max))
        q_release_integral = sum(results['Q_release'])
        self.intermediate_results[e_m].append((priority, n_level_satisfied, q_release_integral))

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

def post(self):
    super().post()
    for e_m in range(self.ensemble_size):
        print(f'\n\nResults for Ensemble Member {e_m}:')
        for priority, n_level_satisfied, q_release_integral in self.intermediate_results[e_m]:
            print(f'\nAfter finishing goals of priority {priority}:')
            print(f'Level goal satisfied at {n_level_satisfied} of \n{len(self.times())} time steps')
            print(f'Integral of Q_release = {q_release_integral:.2f}')
Finally, we want to apply some additional configuration, reducing the amount of information the solver outputs:

```python
def solver_options(self):
    options = super().solver_options()
    # When mumps_scaling is not zero, errors occur. RTC-Tools does its own
    # scaling, so mumps scaling is not critical. Proprietary HSL solvers
    # do not exhibit this error.
    solver = options['solver']
    options[solver]['mumps_scaling'] = 0
    options[solver]['print_level'] = 1
    return options
```

Run the Optimization Problem

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

```
run_optimization_problem(Example)
```

The Whole Script

All together, the whole example script is as follows:

```python
import numpy as np

from rtctools.optimization.collocated_integrated_optimization_problem \
    import CollocatedIntegratedOptimizationProblem
from rtctools.optimization.control_tree_mixin import ControlTreeMixin
from rtctools.optimization.csv_lookup_table_mixin import CSVLookupTableMixin
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin \
    import GoalProgrammingMixin, StateGoal
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

class WaterVolumeRangeGoal(StateGoal):
    def __init__(self, optimization_problem):
        # Assign V_min and V_max the the target range
        self.target_min = optimization_problem.get_timeseries('V_min')
        self.target_max = optimization_problem.get_timeseries('V_max')
        super().__init__(optimization_problem)
        state = 'storage.V'
        priority = 1

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

(continues on next page)
class Example(GoalProgrammingMixin, CSVMixin, CSVLookupTableMixin, ModelicaMixin, ControlTreeMixin, CollocatedIntegratedOptimizationProblem):
    
    An extension of the goal programming and lookuptable examples that 
    demonstrates how to work with ensembles.
    
    # Override default csv_ensemble_mode = False from CSVMixin before calling pre()
    csv_ensemble_mode = True

    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)

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

    def control_tree_options(self):
        # We want to modify the control tree options, so we override the default 
        # control_tree_options method. We call super() to get the default options
        options = super().control_tree_options()
        # Change the branching_times list to only contain the fifth timestep
        options['branching_times'] = [self.times()[5]]
        return options
```python
def priority_completed(self, priority):
    # We want to print some information about our goal programming problem.
    # We store the useful numbers temporarily, and print information at the
    # end of our run.
    for e_m in range(self.ensemble_size):
        results = self.extract_results(e_m)
        self.set_timeseries('V_storage', results['storage.V'], ensemble_member=e_m)

        _max = self.get_timeseries('V_max', ensemble_member=e_m).values
        _min = self.get_timeseries('V_min', ensemble_member=e_m).values
        V_storage = self.get_timeseries('V_storage', ensemble_member=e_m).values

        # A little bit of tolerance when checking for acceptance. This
        # tolerance must be set greater than the tolerance of the solver.
        tol = 10
        _max += tol
        _min -= tol
        n_level_satisfied = sum(
            np.logical_and(_min <= V_storage, V_storage <= _max)
        )
        q_release_integral = sum(results['Q_release'])
        self.intermediate_results[e_m].append((priority, n_level_satisfied, q_release_integral))

def post(self):
    super().post()
    for e_m in range(self.ensemble_size):
        print('

Results for Ensemble Member {}:
'.format(e_m))
        for priority, n_level_satisfied, q_release_integral in 
            self.intermediate_results[e_m]:
            print("After finishing goals of priority {}:
".format(priority))
            print("Level goal satisfied at {} of {} time steps".format(
                n_level_satisfied, len(self.times())))
            print("Integral of Q_release = {:.2f}".format(q_release_integral))

    # Any solver options can be set here
    def solver_options(self):
        options = super().solver_options()
        # When mumps_scaling is not zero, errors occur. RTC-Tools does its own
        # scaling, so mumps scaling is not critical. Proprietary HSL solvers
        # do not exhibit this error.
        solver = options['solver']
        options[solver]['mumps_scaling'] = 0
        options[solver]['print_level'] = 1
        return options

    # Run
run_optimization_problem(Example)
```

### 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:
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:

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

forecast_names = ['forecast1', 'forecast2']
dir_template = '/path/to/examples/ensemble/output/{}/timeseries_export.csv'

# Import Data
forcasts = {}
for forecast in forecast_names:
    data_path = dir_template.format(forecast)
    forcasts[forecast] = np.recfromcsv(data_path, encoding=None)

# Get times as datetime objects
times = list(map(lambda x: datetime.strptime(x, '%Y-%m-%d %H:%M:%S'), forcasts[forecast_names[0]]['time'],
                 forcasts[forecast_names[0]]))
```

(continues on next page)
n_subplots = 2
fig, 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]

# Upper Subplot
axarr[0].set_ylabel("Water Volume in Storage \[m^3\]")
axarr[0].ticklabel_format(style="sci", axis="y", scilimits=(0, 0))

# Lower Subplot
axarr[1].set_ylabel("Flow Rate \[m^3/s\]")

# Plot Ensemble Members
for idx, forecast in enumerate(forecast_names):
    # Upper Subplot
    results = forcasts[forecast]
    if idx == 0:
        axarr[0].plot(
            times, results["v_max"], label="Max", linewidth=2, color="r", linestyle="--")
        axarr[0].plot(
            times, results["v_min"], label="Min", linewidth=2, color="g", linestyle="--")
        axarr[0].plot(
            times, results["v_storage"],
            label=forecast + ":Volume",
            linewidth=2,
            color=get_cmap(cmaps[idx])(shades[1]),
        )
    # Lower Subplot
    axarr[1].plot(
        times,
        results["q_in"],
        label="{}:Inflow".format(forecast),
        linewidth=2,
        color=get_cmap(cmaps[idx])(shades[0]),
    )
    axarr[1].plot(
        times,
        results["q_release"],
        label="{}:Release".format(forecast),
        linewidth=2,
        color=get_cmap(cmaps[idx])(shades[1]),
    )

# Format bottom axis label
axarr[-1].xaxis.set_major_formatter(mdates.DateFormatter("%m/%d"))

# Shrink margins
fig.tight_layout()

# Shrink each axis and put a legend to the right of the axis
(continues on next page)
for i in range(len(axarr)):
    box = axarr[i].get_position()
    axarr[i].set_position([box.x0, box.y0, box.width * 0.75, 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:

![OpenModelica Connection Editor screenshot](image)

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

```modelica
model Example
    // Model Elements
    Deltares.ChannelFlow.Hydraulic.BoundaryConditions.Level Level(H = 0.);
        H(each max = 1.0),
        H_b_down = -2.0,
        H_b_up = -1.5,
        friction_coefficient = 10.,
        length = 2000.,
        theta = theta,
        uniform_nominal_depth = 1.75,
        width_down = 10.,
        width_up = 10.,
        semi_implicit_step_size = step_size,
        Q_nominal = 1.0
    );
```

(continues on next page)
    H(each max = 1.5),
    H_b_down = -1.5,
    H_b_up = -1.0,
    friction_coefficient = 10.,
    length = 2000.,
    theta = theta,
    uniform_nominal_depth = 1.75,
    width_down = 10.,
    width_up = 10.,
    semi_implicit_step_size = step_size,
    Q_nominal = 1.0
);

    H(each max = 2.0),
    H_b_down = -1.0,
    H_b_up = -0.5,
    friction_coefficient = 10.,
    length = 2000.,
    theta = theta,
    uniform_nominal_depth = 1.75,
    width_down = 10.,
    width_up = 10.,
    semi_implicit_step_size = step_size,
    Q_nominal = 1.0
);


// Parameters
parameter Real theta;
parameter Real step_size;

// Inputs
input Real Inflow_Q(fixed = true) = Inflow.Q;
input Real UpperControlStructure_Q(fixed = false, min = 0., max = 4.) =
    →UpperControlStructure.Q;
input Real LowerControlStructure_Q(fixed = false, min = 0., max = 4.) =
    →LowerControlStructure.Q;
input Real DrinkingWaterExtractionPump_Q(fixed = false, min = 0., max = 2.) =
    →DrinkingWaterExtractionPump.Q;

initial equation
    der(LowerChannel.Q) = 0;
    der(MiddleChannel.Q) = 0;
    der(UpperChannel.Q) = 0;

equation
    connect(DrinkingWaterExtractionPump.HQDown, DrinkingWaterPlant.HQ);
    connect(Inflow.HQ, UpperChannel.HQUp);
    connect(LowerChannel.HQDown, Level.HQ);
    connect(LowerControlStructure.HQDown, LowerChannel.HQUp);
    connect(MiddleChannel.HQDown, DrinkingWaterExtractionPump.HQUp);
    connect(MiddleChannel.HQDown, LowerControlStructure.HQUp);
    connect(UpperChannel.HQDown, UpperControlStructure.HQUp);
    connect(UpperChannel.HQDown, MiddleChannel.HQUp);
end Example;
Important: Modellers should take care to set proper values for the initial derivatives, in order to avoid spurious waves at the start of the optimization run. In this example we assume a steady state initial condition for all states except those acted upon by the PID controllers.

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 parameters() method
  - Implementation of path_goals() method
- A run statement

Goals

In this model, we define two generic StateGoal subclasses:

```python
class RangeGoal(StateGoal):
    def __init__(self, opt_prob, state, priority):
        self.state = state
        self.target_min = opt_prob.get_timeseries(state + "_min")
        self.target_max = opt_prob.get_timeseries(state + "_max")
        self.violation_timeseries_id = state + "_target_violation"
        self.function_value_timeseries_id = state
        self.priority = priority
        super().__init__(opt_prob)
```

```python
class TargetGoal(StateGoal):
    def __init__(self, opt_prob, state, priority):
        self.state = state
        self.target_min = opt_prob.get_timeseries(state + "_target")
        self.target_max = self.target_min
        self.violation_timeseries_id = state + "_target_violation"
        self.function_value_timeseries_id = state
        self.priority = priority
        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.

```python
class Example(
    HomotopyMixin,
    GoalProgrammingMixin,
    CSVMixin,
    ModelicaMixin,
    CollocatedIntegratedOptimizationProblem,
):
```

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.

```python
def pre(self):
    super().pre()
    # Generate handy tuples to iterate over
    self.channel_node_indices = tuple(range(1, self.channel_n_level_nodes + 1))
    self.channel_level_nodes = tuple("{}.H[{}]".format(c, n)
                                    for c, n in itertools.product(self.channels, self.channel_node_indices))
    # Expand channel water level goals to all nodes
    for channel in self.channels:
        channel_max = self.get_timeseries(channel + "_max")
        channel_min = self.get_timeseries(channel + "_min")
        for i in self.channel_node_indices:
            self.set_timeseries("{}.H[{}]_max".format(channel, i), channel_max)
            self.set_timeseries("{}.H[{}]_min".format(channel, i), channel_min)
    # Make input series appear in output csv
    self.set_timeseries("Inflow_Q", self.get_timeseries("Inflow_Q"))
    self.set_timeseries("DrinkingWaterExtractionPump_Q_target",
                        self.get_timeseries("DrinkingWaterExtractionPump_Q_target"),
                        )
```

Next, we implement the `parameters()` method. This method passes parameter values down to the model. The model uses the step size parameter to perform a semi-implicit discretization of the hydraulic equations. We set the `step_size` parameter value to match the time step size in the input time series.

```python
def parameters(self, ensemble_member):
    p = super().parameters(ensemble_member)
    times = self.times()
    p["step_size"] = times[1] - times[0]
    return p
```

Finally, 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.

```python
def path_goals(self):
    g = super().path_goals()
    # Add RangeGoal on water level states with a priority of 1
    for node in self.channel_level_nodes:
        g.append(RangeGoal(self, node, 1))
```

(continues on next page)
# Add TargetGoal on Extraction Pump with a priority of 2
```
g.append(TargetGoal(self, "DrinkingWaterExtractionPump_Q", 2))
```

```
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.

```
run_optimization_problem(Example)
```

## The Whole Script

All together, the whole example script is as follows:

```python
import itertools

from rtctools.optimization.collocated_integrated_optimization_problem import (  
    CollocatedIntegratedOptimizationProblem
)
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin,
    StateGoal
from rtctools.optimization.homotopy_mixin import HomotopyMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

class RangeGoal(StateGoal):
    def __init__(self, opt_prob, state, priority):
        self.state = state
        self.target_min = opt_prob.get_timeseries(state + "_min")
        self.target_max = opt_prob.get_timeseries(state + "_max")
        self.violation_timeseries_id = state + "_target_violation"
        self.function_value_timeseries_id = state
        self.priority = priority
        super().__init__(opt_prob)

class TargetGoal(StateGoal):
    def __init__(self, opt_prob, state, priority):
        self.state = state
        self.target_min = opt_prob.get_timeseries(state + "_target")
        self.target_max = self.target_min
        self.violation_timeseries_id = state + "_target_violation"
        self.function_value_timeseries_id = state
```

(continues on next page)
self.priority = priority
super().__init__(opt_prob)

class Example(HomotopyMixin, GoalProgrammingMixin, CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
    channels = "LowerChannel", "MiddleChannel", "UpperChannel"
    channel_n_level_nodes = 2

    def pre(self):
        super().pre()
        # Generate handy tuples to iterate over
        self.channel_node_indices = tuple(range(1, self.channel_n_level_nodes + 1))
        self.channel_level_nodes = tuple("{}.H[{}]".format(c, n)
            for c, n in itertools.product(self.channels, self.channel_node_indices)
        )
        # Expand channel water level goals to all nodes
        for channel in self.channels:
            channel_max = self.get_timeseries(channel + "_max")
            channel_min = self.get_timeseries(channel + "_min")
            for i in self.channel_node_indices:
                self.set_timeseries("{}.H[{}]"_max".format(channel, i), channel_max)
                self.set_timeseries("{}.H[{}]"_min".format(channel, i), channel_min)
        # Make input series appear in output csv
        self.set_timeseries("Inflow_Q", self.get_timeseries("Inflow_Q"))
        self.set_timeseries("DrinkingWaterExtractionPump_Q_target",
            self.get_timeseries("DrinkingWaterExtractionPump_Q_target"),
        )

    def parameters(self, ensemble_member):
        p = super().parameters(ensemble_member)
        times = self.times()
        p["step_size"] = times[1] - times[0]
        return p

    def path_goals(self):
        g = super().path_goals()
        # Add RangeGoal on water level states with a priority of 1
        for node in self.channel_level_nodes:
            g.append(RangeGoal(self, node, 1))

        # Add TargetGoal on Extraction Pump with a priority of 2
        g.append(TargetGoal(self, "DrinkingWaterExtractionPump_Q", 2))
        return g

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

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:

Modeling Waves in Rivers and Canals

Note: This is a more advanced example that implements advanced channel hydraulics 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*.

The RTC-Tools is capable of handling non-linear hydraulics. In this example, we model a river channel that receives a sudden pulse of higher-than-usual water volumes. We compare the results to those of an identical model built in HEC-RAS.

The Model

In this example, water is flowing through a single channel. There is an inflow at the upstream end and a water level bound at the downstream end.

In OpenModelica Connection Editor, the model looks like this in plan view:
In text mode, the Modelica model looks as follows (with annotation statements removed):

```modelica
model Example
  // Elements
    Q_nominal = 100.0,
    H_b_down = -5.0,
    H_b_up = -5.0,
    friction_coefficient = 0.045,
    use_manning = true,
    length = 10000,
    theta = theta,
    use_inertia = true,
    use_convective_acceleration = false,
    use_upwind = false,
    n_level_nodes = 11,
    uniform_nominal_depth = 5.0,
    bottom_width_down = 30,
    bottom_width_up = 30,
    left_slope_angle_up = 45,
    left_slope_angle_down = 45,
    right_slope_angle_up = 45,
    right_slope_angle_down = 45,
    semi_implicit_step_size = step_size
  );
  // Inputs
  input Real Inflow_Q(fixed=true) = Discharge.Q;
  input Real Level_H(fixed=true) = Level.H;
  parameter Real theta;
  parameter Real step_size;
  // Output Channel states
  output Real Channel_Q_up = Discharge.Q;
  output Real Channel_Q_dn = Level.HQ.Q;
  output Real Channel_H_up = Discharge.HQ.H;
  output Real Channel_H_dn = Level.H;
  equation
```

(continues on next page)
37 connect (Channel.HQDown, Level.HQ);
38 connect (Discharge.HQ, Channel.HQUp);
39 initial equation
40 Channel.Q = fill(Inflow_Q, Channel.n_level_nodes + 1);
41 end Example;

The plan view of the model looks like this in HEC-RAS:

![Plan View of Model](image)

The channel cross-section is a simple trapezoidal shape. As rendered by HEC-RAS, here is a cross-section view of the channel being modeled:

![Channel Cross-section](image)

The model was built with HEC-RAS version 5.0.6. In case you wish to verify the HEC-RAS model yourself, a zip of the HEC-RAS model used in this comparison is available: HEC-RAS.zip

The Python File

To keep this example simple and to allow for a 1:1 comparison with HEC-RAS, we will not have any decision variables in this model.

```python
from rtctools.optimization.collocated_integrated_optimization_problem import (CollocatedIntegratedOptimizationProblem,
)
from rtctools.optimization.csv_mixin import CSVMixin
from rtctools.optimization.homotopy_mixin import HomotopyMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

class Example(HomotopyMixin, CSVMixin, ModelicaMixin, CollocatedIntegratedOptimizationProblem):
```
def parameters(self, ensemble_member):
    p = super().parameters(ensemble_member)
    times = self.times()
    if self.use_semi_implicit:
        p['step_size'] = times[1] - times[0]
    else:
        p['step_size'] = 0.0
    p['Channel.use_convective_acceleration'] = self.use_convective_acceleration
    p['Channel.use_upwind'] = self.use_upwind
    return p

def constraints(self, ensemble_member):
    constraints = super().constraints(ensemble_member)
    times = self.times()
    # Extract the number of nodes in the channel
    parameters = self.parameters(ensemble_member)
    n_level_nodes = int(parameters['Channel.n_level_nodes'])

    # To Mimic HEC-RAS behaviour, enforce steady state both at t0 and at t1.
    for i in range(n_level_nodes):
        state = "Channel.H[{}]".format(i + 1)
        constraints.append((self.state_at(state, times[0]) -
                            self.state_at(state, times[1]), 0, 0, 0))
    return constraints

class ExampleInertialWave(Example):
    """Inertial wave equation (no convective acceleration)""
    model_name = 'Example'
    use_semi_implicit = False
    use_convective_acceleration = False
    use_upwind = False
    timeseries_export_basename = "timeseries_export_inertial_wave"

class ExampleInertialWaveSemiImplicit(Example):
    """Inertial wave equation (no convective acceleration)""
    model_name = 'Example'
    use_semi_implicit = True
    use_convective_acceleration = False
    use_upwind = False
    timeseries_export_basename = "timeseries_export_inertial_wave_semi_implicit"

class ExampleSaintVenant(Example):
    """Saint Venant equation. Convective acceleration discretized with central differences"""
    model_name = 'Example'
    use_semi_implicit = False
    use_convective_acceleration = True
    use_upwind = False
    timeseries_export_basename = "timeseries_export_saint_venant"
As you can see, this model is as simple as it gets. We only add a constraint to keep the initialization states consistent with the HEC-RAS initialization.

**Comparison of Discretizations and Numerical Schemes**

HEC-RAS and RTC-Tools use different discretizations and numerical schemes, but also solve different equations. RTC-Tools solves the original nonlinear equations, whereas HEC-RAS solves a linearized momentum equation.

<table>
<thead>
<tr>
<th></th>
<th>RTC-Tools 2</th>
<th>HEC-RAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Momentum equation</td>
<td>Saint-Venant / inertial wave (default)</td>
<td>Linearized Saint-Venant</td>
</tr>
<tr>
<td>Spatial discretization</td>
<td>Staggered</td>
<td>Collocated</td>
</tr>
<tr>
<td>Numerical scheme (temporal)</td>
<td>Semi-implicit / implicit (default)</td>
<td>Centered Preissmann box scheme</td>
</tr>
<tr>
<td>Numerical scheme (spatial)</td>
<td>Central differences, upwind convective acceleration (optional)</td>
<td>Centered Preissmann box scheme</td>
</tr>
</tbody>
</table>

**Note:** For optimization, the recommended momentum equation and temporal scheme for RTC-Tools is semi-implicit inertial wave. Consult Baayen and Piovesan, *A continuation approach to nonlinear model predictive control of open channel systems*, 2018, for details. A preprint is available online as arXiv:1801.06507.
Comparison of Results

The results from the RTC-Tools run are found in the output directory with the name `timeseries_export.csv`, and the results generated by HEC-RAS have been exported into the same directory under the name `HEC-RAS_results.csv`. We can compare the results using the Python library `matplotlib`:

Both HEC-RAS and RTC-Tools were run with a spatial step size of 1000 m and a temporal step size of 15 min.

Controlling a Cascade of Weirs: Local PID Control vs. Optimization

![Cascade of Weirs](image)

**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*.

One of the advantages of using RTC-Tools for control is that it is capable of making decisions that are optimal for the whole network and for all future timesteps within the model time horizon. This is in contrast to local control algorithms such as PID controllers, where the control decision must be made on past states and local information alone. Furthermore, unlike a PID-style controller, RTC-Tools does not have gain parameters that need to be tuned.

This example models a cascading channel system, and compares a local control scheme using PID controllers with the RTC-Tools approach that uses Goal Programming.

**The Model**

For this example, water is flowing through a multilevel channel system. The model has three channel sections. There is an inflow forcing at the upstream boundary and a water level at the downstream boundary. The decision variables are the flow rates (and by extension, the weir settings) between the channels.

In OpenModelica Connection Editor, the model looks like this:
In text mode, the Modelica model looks as follows (with annotation statements removed):

```modelica
model Example

// Structures
    theta = theta,
    semi_implicit_step_size = step_size,
    H_b_up = 15,
    H_b_down = 15,
    bottom_width_up = 50,
    bottom_width_down = 50,
    length = 20000,
    uniform_nominal_depth = 5,
    friction_coefficient = 35,
    n_level_nodes = 4,
    Q_nominal = 100.0
);
    theta = theta,
    semi_implicit_step_size = step_size,
    H_b_up = 10,
    H_b_down = 10,
    bottom_width_up = 50,
    bottom_width_down = 50,
    length = 20000,
    uniform_nominal_depth = 5,
    friction_coefficient = 35,
    n_level_nodes = 4,
    Q_nominal = 100.0
);
    theta = theta,
    semi_implicit_step_size = step_size,
    H_b_up = 5,
    H_b_down = 5,
```
Note: In order to simulate and show how the PID controllers activate only once the incoming wave has propagated downstream, we will discretize the model in time with a resolution of 5 minutes. With our spatial resolution of 4 level nodes per 20 km reach, this produces a CFL number of approximately 0.4.

For optimization-based control, such a fine temporal resolution is not needed, as the system is able to look ahead and plan corrective measures ahead of time. In this case, CFL numbers of up to 1 or even higher are typically used.

Nevertheless, in order to present a consistent comparison, a 5 minute time step is also used for the optimization example. It is easy to explore the effect of the time step size on the optimization results by changing the value of the step_size class variable.

To run the model with the local control scheme, we make a second model that constrains the flow rates to the weir settings as determined by local PID controller elements:

```model ExampleLocalControl
  extends Example;
  // Add PID Controllers that apply local control
  PIDController upstream_pid(
    state = dam_upstream.HQUp.H,
  )
end Example;
```
The local control model makes use of a PID controller class:

```plaintext
model PIDController
  input Real state;
  parameter Real target_value;
  parameter Real P = 1.0;
  parameter Real I = 0.0;
  parameter Real D = 0.0;
  parameter Real feed_forward = 0.0;
  output Real control_action;
  Real _error;
  Real error_integral(nominal = 3600);
  equation
    _error = target_value - state;
    der(error_integral) = _error;
    control_action = feed_forward + P * _error + I * error_integral + D * der(_error);
  initial equation
    error_integral = 0.0;
end PIDController;
```

**Important:** Modellers should take care to set proper values for the initial derivatives, in order to avoid spurious waves at the start of the optimization run. In this example we assume a steady state initial condition, as indicated and enforced by the `SteadyStateInitializationMixin` in the Python code.

The Optimization Problem

Goals

In this model, we define a TargetLevelGoal to find a requested target level:

```plaintext
class TargetLevelGoal(Goal):
  """Really Simple Target Level Goal""
```

(continues on next page)
**RTC-Tools Documentation, Release 2.4.0a2+18.g49ab02a**

(continued from previous page)

```python
def __init__(self, state, target_level):
    self.function_range = target_level - 5.0, target_level + 5.0
    self.function_nominal = target_level
    self.target_min = target_level
    self.target_max = target_level
    self.state = state

def function(self, optimization_problem, ensemble_member):
    return optimization_problem.state(self.state)

priority = 1
```

We will later apply this goal to the upstream and middle channels.

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.

```python
class ExampleOptimization(
    StepSizeParameterMixin,
    SteadyStateInitializationMixin,
    HomotopyMixin,
    GoalProgrammingMixin,
    CSVMixin,
    ModelicaMixin,
    CollocatedIntegratedOptimizationProblem,
):
```

The `StepSizeParameterMixin` defines the step size parameter and sets the optimization time steps, while the `SteadyStateInitializationMixin` constrains the initial conditions to be steady-state.

Next, we instantiate the goals. There are two water level goals, applied at the upper and middle channels. The goals are very simple—they just target a specific water level.

```python
def path_goals(self):
    # Add water level goals
    return [
        TargetLevelGoal("dam_upstream.HQUp.H", 20.0),
        TargetLevelGoal("dam_middle.HQUp.H", 15.0),
    ]
```

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.

For comparison, we also define an optimization problem that uses a local control scheme. This example does not use any goals, as the flow rate is regulated by the PID Controller.

```python
class ExampleLocalControl(
    StepSizeParameterMixin,
    SteadyStateInitializationMixin,
    HomotopyMixin,
    CSVMixin,
)
```

(continues on next page)
```
ModelicaMixin,
CollocatedIntegratedOptimizationProblem,
);

"""Local Control Approach"
"

timeseries_export_basename = "timeseries_export_local_control"
```

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 classes we just created. We do this for both the local control model and the goal programming model.

```
runtime_optimization = run_optimization_problem(ExampleOptimization)
runtime_local_control = run_optimization_problem(ExampleLocalControl)
```

The Whole Script

All together, all the scripts are as as follows:

**step_size_parameter_mixin.py**:

```
import numpy as np
from rtctools.optimization.optimization_problem import OptimizationProblem

class StepSizeParameterMixin(OptimizationProblem):
    step_size = 5 * 60  # 5 minutes

    def times(self, variable=None):
        times = super().times(variable)
        return np.arange(times[0], times[-1], self.step_size)

    def parameters(self, ensemble_member):
        p = super().parameters(ensemble_member)
        p['step_size'] = self.step_size
        return p
```

**steady_stateInitialization_mixin.py**:

```
from rtctools.optimization.optimization_problem import OptimizationProblem

class SteadyStateInitializationMixin(OptimizationProblem):
    def constraints(self, ensemble_member):
        c = super().constraints(ensemble_member)
        times = self.times()
        parameters = self.parameters(ensemble_member)
        # Force steady-state initialization at t0 and at t1.
        for reach in ['upstream', 'middle', 'downstream']:
            for i in range(int(parameters[f'{reach}.n_level_nodes']) + 1):
                state = f'{reach}.Q[{i + 1}]'
(continues on next page)```
c.append((self.der_at(state, times[0]), 0, 0))
return c

timeseries_export_basename = "timeseries_export_local_control"

def __name__ == "__main__":
    run_optimization_problem(ExampleLocalControl)

class TargetLevelGoal(Goal):
    """Really Simple Target Level Goal""

from example_local_control import ExampleLocalControl
from rtctools.optimization.collocated_integrated_optimization_problem import (CollocatedIntegratedOptimizationProblem,
    CSVMixin,
).
from rtctools.optimization.homotopy_mixin import HomotopyMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

from steady_state_initialization_mixin import SteadyStateInitializationMixin

from step_size_parameter_mixin import StepSizeParameterMixin

class TargetLevelGoal(Goal):
    """Really Simple Target Level Goal""

from example_local_control import ExampleLocalControl
from rtctools.optimization.collocated_integrated_optimization_problem import (CollocatedIntegratedOptimizationProblem,
    CSVMixin,
).
from rtctools.optimization.goal_programming_mixin import Goal, GoalProgrammingMixin 
from rtctools.optimization.homotopy_mixin import HomotopyMixin
from rtctools.optimization.modelica_mixin import ModelicaMixin
from rtctools.util import run_optimization_problem

from steady_state_initialization_mixin import SteadyStateInitializationMixin

from step_size_parameter_mixin import StepSizeParameterMixin

class TargetLevelGoal(Goal):
    """Really Simple Target Level Goal""

1.4. Examples
```python
def __init__(self, state, target_level):
    self.function_range = target_level - 5.0, target_level + 5.0
    self.function_nominal = target_level
    self.target_min = target_level
    self.target_max = target_level
    self.state = state

    def function(self, optimization_problem, ensemble_member):
        return optimization_problem.state(self.state)

    priority = 1

class ExampleOptimization(
    StepSizeParameterMixin,
    SteadyStateInitializationMixin,
    HomotopyMixin,
    GoalProgrammingMixin,
    CSVMixin,
    ModelicaMixin,
    CollocatedIntegratedOptimizationProblem,
):
    """Goal Programming Approach""

    def path_goals(self):
        # Add water level goals
        return [
            TargetLevelGoal("dam_upstream.HQUp.H", 20.0),
            TargetLevelGoal("dam_middle.HQUp.H", 15.0),
        ]

    # Run
    run_optimization_problem(ExampleOptimization)
    run_optimization_problem(ExampleLocalControl)
```

### Extracting Results

The results from the run are found in output/timeseries_export.csv and output/timeseries_export_local_control.csv. Here are the results when plotted using the python library matplotlib:

In this example, the PID controller is tuned poorly and ends up amplifying the incoming wave as it propagates downstream. The optimizing controller, in contrast, does not amplify the wave and maintains the target water level throughout the wave event.

#### 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
2. 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:

  Storage,
  Inflow,
  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):

```modelica
model Example
  // Elements
  Deltares.ChannelFlow.SimpleRouting.BoundaryConditions.Inflow inflow(Q = Q_in);
  Deltares.ChannelFlow.SimpleRouting.Storage.Storage storage(Q_release = P_control,
    →V(start=storage_V_init, fixed=true, nominal=4e5));
  // Initial States
```
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 QSISO which contains the connectors QIn and QOut. With QSISO, 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 form the rtctools.util package, and any extra packages we want to use. For this example, the import block looks like:

```python
import logging
from rtctools.simulation.csv_mixin import CSVMixin
from rtctools.simulation.simulation_problem import SimulationProblem
from rtctools.util import run_simulation_problem
```
logger = logging.getLogger("rtctools")

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.

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.

def update(self, dt):
    # Get the time step
    if dt < 0:
        dt = self.get_time_step()

    # Get relevant model variables
    volume = self.get_var('storage.V')
    target = self.get_var('storage_V_target')

    # Calculate error in storage.V
    error = target - volume

    # Calculate the desired control
    control = -error / dt

    # Get the closest feasible setting.
    bounded_control = min(max(control, self.min_release), self.max_release)

    # Set the control variable as the control for the next step of the simulation
    self.set_var('P_control', bounded_control)

    # Call the super class so that everything else continues as normal
    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.

run_simulation_problem(Example, log_level=logging.DEBUG)

The Whole Script

All together, the whole example script is as follows:
import logging

from rtctools.simulation.csv_mixin import CSVMixin
from rtctools.simulation.simulation_problem import SimulationProblem
from rtctools.util import run_simulation_problem

logger = logging.getLogger("rtctools")

class Example(CSVMixin, SimulationProblem):
    
    
    def initialize(self):
        
        self.set_var('P_control', 0.0)
        super().initialize()

        # Min and Max flow rate that the storage is capable of releasing
        min_release, max_release = 0.0, 8.0  # m^3/s

        # Here is an example of overriding the update() method to show how control
        # can be build into the python script
        def update(self, dt):

            # Get the time step
            if dt < 0:
                dt = self.get_time_step()

            # Get relevant model variables
            volume = self.get_var('storage.V')
            target = self.get_var('storage_V_target')

            # Calculate error in storage.V
            error = target - volume

            # Calculate the desired control
            control = -error / dt

            # Get the closest feasible setting.
            bounded_control = min(max(control, self.min_release), self.max_release)

            # Set the control variable as the control for the next step of the simulation
            self.set_var('P_control', bounded_control)

            # Call the super class so that everything else continues as normal
            super().update(dt)

            # Run
            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), 22
__init__()  (rtctools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderStateGoal method), 24
__init__()  (rtctools.optimization.min_abs_goal_programming_mixin.MinAbsStateGoal method), 24
__init__()  (rtctools.optimization.timeseries.Timeseries method), 6
bounds()  (rtctools.optimization.optimization_problem.OptimizationProblem method), 6

CollocatedIntegratedOptimizationProblem (class in rtctools.optimization.collocated_integrated_optimization_problem), 13
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), 25
ControlTreeMixin (class in rtctools.optimization.control_tree_mixin), 25
critical (rtctools.optimization.goal_programming_mixin.Goal attribute), 20
CSVMixin (class in rtctools.optimization.csv_mixin), 14
CSVMixin (class in rtctools.simulation.csv_mixin), 27
CSVLookupTableMixin (class in rtctools.optimization.csv_lookup_table_mixin), 16
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
ensemble_member_probability()  (rtctools.optimization.optimization_problem.OptimizationProblem method), 8
ensemble_size  (rtctools.optimization.optimization_problem.OptimizationProblem attribute), 8
function()  (rtctools.optimization.goal_programming_mixin.Goal method), 20
function_nominal  (rtctools.optimization.goal_programming_mixin.Goal attribute), 20
function_range  (rtctools.optimization.goal_programming_mixin.Goal attribute), 20
function_value_timeseries_id  (rtctools.optimization.goal_programming_mixin.Goal attribute), 20
get_current_time()  (rtctools.simulation.simulation_problem.SimulationProblem method), 25

97
get_end_time() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 25
get_function_key() (rtc-tools.optimization.goal_programming_mixin.Goal method), 20
get_start_time() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 25
get_timeseries() (rtc-tools.optimization.optimization_problem.OptimizationProblem method), 8
get_var() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 25
get_var_count() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
get_var_name() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
get_var_rank() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
get_var_shape() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
get_var_type() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
get_variables() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
Goal (class in rtc-tools.optimization.goal_programming_mixin), 18
goal_programming_options() (rtc-tools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 22
goal_programming_options() (rtc-tools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderGoalProgrammingMixin method), 24
GoalProgrammingMixin (class in rtc-tools.optimization.goal_programming_mixin), 22
goals() (rtc-tools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 23
has_target_bounds (rtc-tools.optimization.goal_programming_mixin.Goal attribute), 20
has_target_max (rtc-tools.optimization.goal_programming_mixin.Goal attribute), 21
has_target_min (rtc-tools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderGoal attribute), 24
history() (rtc-tools.optimization.optimization_problem.OptimizationProblem method), 8
homotopy_options() (rtc-tools.optimization.homotopy_mixin.HomotopyMixin method), 17
HomotopyMixin (class in rtc-tools.optimization.homotopy_mixin), 17
initial_state() (rtc-tools.optimization.optimization_problem.OptimizationProblem method), 8
initial_state_measurements() (rtc-tools.optimization.initial_state_estimation_mixin.InitialStateEstimationMixin method), 18
initial_state_smoothing_pairs() (rtc-tools.optimization.initial_state_estimation_mixin.InitialStateEstimationMixin method), 18
initial_time (rtc-tools.optimization.optimization_problem.OptimizationProblem attribute), 8
initialize() (rtc-tools.simulation.simulation_problem.SimulationProblem method), 26
integrate() (rtc-tools.optimization.optimization_problem.OptimizationProblem method), 8
integrated_states (rtc-tools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem attribute), 13
integrate_state() (rtc-tools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem method), 13
LinearizationMixin (class in rtc-tools.optimization.linearization_mixin), 15
linearization_parameters() (rtc-tools.optimization.linearization_mixin.LinearizationMixin method), 16
linearize_order (rtc-tools.optimization.linearized_order_goal_programming_mixin.LinearizedOrderGoal attribute), 24
LinearizedOrderGoal (class in rtc-tools.optimization.linearized_order_goal_programming_mixin), 24

Index
LinearizedOrderGoalProgrammingMixin (class in rtc-
tools.optimization.linearized_order_goal_programming_mixin), 24

LinearizedOrderStateGoal (class in rtc-
tools.optimization.linearized_order_goal_programming_mixin), 24

lookup_tables() (rtc-
tools.optimization.optimization_problem.OptimizationProblem method), 9

lookup_tables() (rtc-
tools.optimization.optimization_problem.OptimizationProblem method), 10

lookup_tables() (rtc-
tools.optimization.optimization_problem.OptimizationProblem method), 17

lookup_tables() (rtc-
tools.optimization.optimization_problem.OptimizationProblem method), 23

LookupTable (class in rtc-
tools.optimization.csv_lookup_table_mixin), 16

max_timeseries_id() (rtc-
tools.optimization.csv_mixin.CSVMixin method), 14

max_timeseries_id() (rtc-
tools.optimization.pi_mixin.PIMixin method), 15

min_abs_goals() (rtc-
tools.optimization.min_abs_goal_programming_mixin.MinAbsGoalProgrammingMixin method), 23

min_abs_path_goals() (rtc-
tools.optimization.min_abs_goal_programming_mixin.MinAbsGoalProgrammingMixin method), 24

min_timeseries_id() (rtc-
tools.optimization.csv_lookup_table_mixin), 14

min_timeseries_id() (rtc-
tools.optimization.pi_mixin.PIMixin method), 15

MinAbsGoal (class in rtc-
tools.optimization.min_abs_goal_programming_mixin), 24

MinAbsGoalProgrammingMixin (class in rtc-
tools.optimization.min_abs_goal_programming_mixin), 23

MinAbsStateGoal (class in rtc-
tools.optimization.min_abs_goal_programming_mixin), 24

ModelicaMixin (class in rtc-
tools.optimization.modelica_mixin), 14

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

OptimizationProblem (class in rtc-
tools.optimization.optimization_problem), 6

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

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

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

path_objective() (rtc-
tools.optimization.optimization_problem.OptimizationProblem method), 10

PIMixin (class in rtctools.optimization.pi_mixin), 15

PIMixin (class in rtctools.simulation.pi_mixin), 27

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

post() (rtctools.simulation.optimization_problem.SimulationProblem method), 26

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

pre() (rtctools.simulation.optimization_problem.SimulationProblem method), 26

priority (rtc-
tools.optimization.goal_programming_mixin.GoalProgrammingMixin), 21

priority_completed() (rtc-
tools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 23

priority_started() (rtc-
tools.optimization.goal_programming_mixin.GoalProgrammingMixin method), 23

relaxation (rtc-
tools.optimization.goal_programming_mixin.Goal attribute), 21

reset() (rtctools.simulation.optimization_problem.SimulationProblem method), 26

reset() (rtctools.optimization.optimization_problem.OptimizationProblem method), 26

run_optimization_problem() (in module rtc-
tools.util), 12

run_simulation_problem() (in module rtc-
tools.util), 27

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

set_timeseries() (rtc-
tools.optimization.optimization_problem.OptimizationProblem method), 10

setup_experiment() (rtc-
tools.simulation.optimization_problem.SimulationProblem method), 26

Index 99
simulate() (rtctools.simulation.simulation_problem.SimulationProblem method), 26
update() (rtctools.simulation.simulation_problem.SimulationProblem method), 26
SimulationProblem (class in rtctools.simulation.simulation_problem), 25
Reflection() (rtctools.optimization.optimization_problem.OptimizationProblem method), 11
solver_options() (rtctools.optimization.optimization_problem.OptimizationProblem method), 11
dimensions() (rtctools.optimization.optimization_problem.OptimizationProblem method), 11
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), 12
StateGoal (class in rtctools.optimization.goal_programming_mixin), 21
states_in() (rtctools.optimization.optimization_problem.OptimizationProblem method), 12

T

target_max (rtctools.optimization.goal_programming_mixin.Goal attribute), 21
target_min (rtctools.optimization.goal_programming_mixin.Goal attribute), 21
theta (rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem attribute), 13
times (rtctools.optimization.timeseries.Timeseries attribute), 6
times() (rtctools.optimization.collocated_integrated_optimization_problem.CollocatedIntegratedOptimizationProblem method), 14
Timeseries (class in rtctools.optimization.timeseries), 6
timeseries_at() (rtctools.optimization.optimization_problem.OptimizationProblem method), 12
timeseries_at() (rtctools.simulation.csv_mixin.CSVMixin method), 27
timeseries_at() (rtctools.simulation.pi_mixin.PIMixin method), 28
timeseries_export (rtctools.optimization.pi_mixin.PIMixin attribute), 15
timeseries_import (rtctools.optimization.pi_mixin.PIMixin attribute), 15
timeseries_import_times (rtctools.optimization.pi_mixin.PIMixin attribute), 15