User's Guide for OpenModelica Python API

Bernt Lie
University College of Southeast Norway

Version of August 18, 2016

Contents

List of Figures

List of Tables

Preface

Modelica is a modern, equation based, acausal language for encoding models of dynamic systems in the form of differential algebraic equations (DAEs). *OpenModelica* is a mature, freely available toolset that includes *OpenModelica Connection Editor* (flow sheeting, textual editor with debugging facilities, and simulation environment) and the *OMShell* (command line execution, script based execution). OpenModelica Shell supports commands for simulation of Modelica models, for use of the Modelica extension *Optimica*, for carrying out analytic linearization via the Modelica package *Modelica_LinearSystem2*, and for converting Modelica models into Functional Mock-Up Units (FMUs) as well as for converting FMUs back to Modelica models. A tool *OMPython* has been developed and communicates with OpenModelica via CORBA. Essentially, OMPython is a Python package which makes it possible to pass OpenModelica Shell commands as strings to a Python function, and then receive the results back into Python. This possibility does, however, require good knowledge of OpenModelica Shell commands and syntax.

Modelica and OpenModelica Shell in themselves have relatively little support for advanced analysis of models. Examples of such desirable analysis capabilities could be (i) study of model sensitivity, (ii) random number generation and statistical analysis, (iii) Monte Carlo simulation, (iv) advanced plotting capabilities, (v) general optimization capabilities, (vi) linear analysis and control synthesis, etc. Scripting languages such as MATLAB and Python hold most of these desirable analysis capabilities, and it is of interest to integrate Modelica models with such script languages. The free *JModelica.org* tool includes a Python package for converting Modelica models to FMUs, and then for importing the FMU as a Python object. This way, Modelica models can essentially be simulated from Python — Optimica is also supported. It is possible to do more advanced analysis with `JModelica.org` via *CasADi*, but these possibilities are not always pursued by the developer of `JModelica.org`. It would be more ideal if such possibilities were supported by the tool developer.

It is thus of interest to develop an extension of OMPython which enables simulation and analysis of Modelica models with a better integration with the Python language, and in particular that such an extension is provided by the OpenModelica developers. A Python API[1] for controlling Modelica simulation and analysis from Python was proposed in February 2015[2]. Based on this proposal, a first version of a Python API was implemented by Sudeep Bajracharya in his MSc thesis at Linköping University (2015-2016). [3]. This work has been further refined at Linköping University during the spring and early summer of 2016, principally by Mr. Bajracharya, in an interaction with the author of this document and many developers at the PETEK group of Linköping University.

[1] API = Application Programming Interface

[2] *Python API for Accessing OpenModelica Models*, by B. Lie, February 20, 2015, communicated to P. Fritzson of Linköping University.

Bernt Lie, Porsgrunn, August 18, 2016

Chapter 1

Introduction

1.1 Dynamic systems

System is a basic concept in analysis. A system may be *defined* as in [5]:

- A system $\mathcal{S}$ is an object or collection of objects whose properties we want to study.

We could alternatively say that a system $\mathcal{S}$ is a well defined subset of the *world* $\mathcal{W}$ which we want to study, where we know precisely what is part of the system and what is not part of the system. The parts of the world that is exterior to the system, $\mathcal{E} = \complement_{\mathcal{W}}\mathcal{S}$, is denoted the *environment*.

Information exchange $\mathcal{I}$ may take place between the system and the environment, such as interaction of mass, energy, force, sensor signal, etc.

The idea of *causality* is important in physics. From experience, reality is *causal*, which implies that there is a *cause* that leads to an *effect*. In other words: the cause must come first, and the effect is a result of the cause; information exchange has a direction. By *inputs* to the system is meant flow of information from the environment into the system which *affects* the behavior of the *system*, in another word: causes. The common symbol used for inputs in system science is u. Similarly, *outputs* from the system is flow of information from the system into the environment which *affects* the behavior of the *environment*, in another word: effects. The symbol used for outputs is y. Outputs include information about the properties that we seek in the system.

All systems are *dynamic*, in the sense that they exhibit some inertia against change: change takes time. Our observation of this inertia depends on the time scale considered. Some systems may appear to have instantaneous response in a certain time scale, but appear to have considerable inertia against change in another time scale.

The *purpose* of defining a system is to find out something about the properties of interest in the system through observations. Thus, we can observe the value of outputs from a system. It is reasonable to think that we can find out more about a system by manipulating (if possible) the inputs to the system. An experiment can be defined as [5]:

- An *experiment* is the process of manipulating inputs to a system in such a way as to be able to extract information about the system.

In system science, both inputs u to the system and outputs y from the system are variables of time. Thus, carrying out experiments imply varying inputs to the system, and observing the response in the outputs. However, it is also of interest to carry out experiments where the *design* of the system is changed, e.g. the geometry, fluid properties, topology of subsystems, etc. These quantities are strictly speaking not inputs to the system.

1.2 Model types

A model is a *representation* of a system, including the inputs that influence the behavior of the system and the outputs that influence the environment. A model can be defined as [5]:

- A *model* of a system is anything an "experiment" can be applied to in order to answer questions about the system in question.

A real problem in modeling is that we may not be aware of all inputs that influence the system behavior. And even if we do know of an input variable that influences the system, we may not know its precise value — sometimes, we only know that it varies randomly with realizations from a given probability distribution.

Inputs to a system that can be manipulated at will and in fact are manipulated, are known as control inputs or manipulated variables. Inputs to a system that are known but can not be manipulated at will or which we choose to not manipulate, or inputs that we do not even know of, are known as *disturbances*.

Realistically, there is an infinite number of outputs (influences) from a system to the environment. We will, however, only include in output variable y those outputs that we measure and/or are properties of interest in the system.

In summary, this implies that in a model description, our sets of input and output variables u, y will be a subset of the information exchange $\mathcal{I}$ between the system and the environment.

Many types of models/system representations are useful. However, here we will consider *quantitative* mathematical models, i.e., mathematical models that can be used to quantify forecasts numerically. We will restrict the discussion to models of finite order where time is allowed to vary continuously, and where those inputs that vary randomly can only change at a countable number of time instances.[1]

The most general model for such systems can be written as the DAE[2] system

$$0 = f\left(\frac{dx}{dt}, x, u; \theta\right) \tag{1.1}$$

$$0 = g\left(y, x, u; \theta\right) \tag{1.2}$$

where $f(\cdot)$ is some function, $\frac{dx}{dt}$ is the rate of change of some hidden quantity x,[3] while u is the input variable to the system. Furthermore, $g(\cdot)$ is some function and y is the output variable from the system. Quantity θ represents model *parameters*, which are constants describing system geometry (volume, etc.), material constants (fluid density, etc.), etc. — in other words, parameters may vary from experiment to experiment.[4] System inputs u and outputs y are in principle invariants of the model, i.e. they are the same no matter how detailed the model is. However, quantities x and parameters θ depend on the model complexity.

[1] If inputs that vary randomly are allowed to vary at any time $t \in \mathbb{R}$, the model description is much more complicated than if the models are restricted to only vary at a countable number of times $t \in \{t_i \mid i \in \mathbb{I}\}$.

[2] DAE = Differential and Algebraic Equation

[3] $\frac{dx}{dt}$ is related to the change and inertia of the system

[4] Models may also contain constants of type *natural constants* (e.g., π, Boltzmann's constant, etc.) which have the same value in any experiment; these natural constants are not contained in parameters θ.

Many real models can be written in the structured DAE form of Eqs. 1.3–1.5:

$$\frac{dx}{dt} = f(x, z, u; \theta) \tag{1.3}$$

$$0 = h(x, z, u; \theta) \tag{1.4}$$

$$y = g(x, z, u; \theta). \tag{1.5}$$

In addition to system inputs u and outputs y, the structured DAE form includes differential variables x, algebraic variables z, and parameters θ.

When formulating models based on a mechanistic understanding of the world, it is common to introduce conservation/balance laws (mass conservation, species balances, momentum balances, energy balance), and complete the model with constitutive equations (Equations of State, transport laws, friction model, etc.). Such models fit directly into Eqs. 1.3–1.4 where the conserved/balanced quantities are collected in the differential variables x, the balance laws are differential equations of form $\frac{dx}{dt} = f(x, z, u; \theta)$, while the additional constitutive laws are algebraic equations of form $0 = h(x, z, u; \theta)$.

The structured DAE form can be simplified to an ODE[5] form model as in Eqs. 1.6–1.7 by the elimination of algebraic variables z:

$$\frac{d\tilde{x}}{dt} = \tilde{f}\left(\tilde{x}, u; \tilde{\theta}\right) \tag{1.6}$$

$$y = \tilde{g}\left(\tilde{x}, u; \tilde{\theta}\right). \tag{1.7}$$

If the structured DAE is of index 0 or 1 [4], then $\dim x = \dim \tilde{x}$, but in principle x and $\tilde{x}$ may contain different quantities. Furthermore, normally $\dim \tilde{\theta} \leq \dim \theta$ — elimination of algebraic variables may render some parameter unnecessary. For higher index models, it is necessary to introduce additional variables in $\tilde{x}$, e.g., using Pantelides' method, leading to $\dim \tilde{x} > \dim x$.

In order to solve models, the DAE or ODE must be known, together with the time evolution of inputs $u(t)$, and model parameters $\theta/\tilde{\theta}$. However, this information is not sufficient to make the solution unique: the minimum *additional* information to make the solution unique, is known as the initial *state* of the system. For models of form Eqs. 1.6–1.7, quantity $\tilde{x}$ — or a nonsingular transformation of $\tilde{x}$ — can serve as the system state. Thus, it is common to refer to $\tilde{x}$ in Eqs. 1.6–1.7 as the system state, hence the initial value of $\tilde{x}$ must be known in order to make the model solution unique.

For a structured DAE model as in Eqs. 1.3–1.5 of index 0 or 1, the transformation between x and $\tilde{x}$ of the ODE model is unique, hence we can also refer to x in Eqs. 1.3–1.5 as the state.

A system is *state controllable* if there exists an input sequence $u(t)$ such that the system state (x, $\tilde{x}$) can be changed from an arbitrary initial value to an arbitrary final value in finite time. To be controllable, this definition requires that *every state* can be moved in the specified way. A system is *observable* if by observing a sequence of outputs $y(t)$ over a finite time, it is possible to infer the value of the initial state.

Controllability and observability of a system is assessed from a model of the system. Systems can exhibit any of four combinations of (controllable, non-controllable) and (observable, non-observable), and systems with any combination can be modeled/simulated.

Because non-observable modes of the system state can not be observed through y, these modes are redundant and may just as well be removed from the model. Likewise,

[5]ODE = Ordinary Differential Equation

non-controllable modes can not be affected through u, hence are of no use in operational studies. It follows that a model should normally be both controllable and observable. A model which is both controllable and observable, is known as a *minimal realization* of the system.

1.3 Simulation and Model use

1.3.1 Simulation

Models are often discussed in tandem with *simulation*[6]. Simulation can be defined as follows [5]:

- A *simulation* is an experiment performed on a model.

Simulation can thus include both varying input variables, and varying model parameters. Simulation is used for both model prediction and inverse problems.

1.3.2 Model prediction

With model prediction, model causality is used to:

- predict how evolution of input variables influence the behavior of the system through the outputs (operational behavior),
- study how similar the model is to the real system by comparing model and system outputs when using the same input variables (model deviation),
- predict how variation in model parameters (design parameters, material parameters, etc.) influence the behavior of the system (design studies),
- predict how variation in model parameters influence how similar the model is to the real system (model sensitivity, parameter identifiability, model uncertainty),
- predict how different model realizations compare wrt. representing the real system (model simplification),
- predict how explicit control laws are able to suppress disturbances and achieve operational requirements,
- predict how explicit state observers are able to compute hidden states.

1.3.3 Inverse problems

It is often of interest to use the models non-causally, which essentially implies to invert the model. Inverting the model is numerically challenging; related problems are denoted inverse problems. Examples of inverse problems are:

- if a desired output is specified, what input evolution is necessary in order to achieve this output?

[6]Simulation: from Latin *simulare*, meaning to *pretend*.

- if a desired output is specified, what model parameters are needed in order to achieve this output?
- if a model output is to fit a real system output in some optimal sense (e.g., least squares sense), what is the best choice of model parameters? (Parameter estimation, optimal design.)
- if a model output is to fit a real system output in some optimal sense, what is the most likely initial state of the system? (State estimation; can be considered a special case of parameter estimation where the initial states are considered model parameters.)
- if a model output is to fit a specified system output in some optimal sense, what is the best choice of input variables? (Optimal control, etc.),
- study the importance of model accuracy in inverse problems.

1.4 Modelica and OpenModelica

Modelica is a modern, equation based, acausal language for encoding models of dynamic systems in the form of differential algebraic equations (DAEs) as in Eqs. 1.1–1.2 or Eqs. 1.3–1.5, see e.g. [5] on Modelica and e.g. [4] on DAEs. How is *acausality* related to the *causal* nature of the world? Causality is useful for developing and understanding mechanistic models. However, when a model has been developed, it may be of interest to forget the causality of the system, e.g. in inverse problems. Modelica allows for this, which is why it is described as acausal.

OpenModelica[7] is a mature, freely available toolset that includes *OpenModelica Connection Editor* (flow sheeting, textual editor with debugging facilities, and simulation environment) and the *OMShell* (command line execution, script based execution). OpenModelica Shell supports commands for simulation of Modelica models, for use of the Modelica extension *Optimica*, for carrying out analytic linearization via the Modelica package *Modelica_LinearSystem2*, and for converting Modelica models into Functional Mock-Up Units (FMUs) as well as for converting FMUs back to Modelica models. A tool *OMPython* has been developed and communicates with OpenModelica via CORBA. Essentially, OMPython is a Python package which makes it possible to pass OpenModelica Shell commands as strings to a Python function, and then receive the results back into Python. This possibility does, however, require good knowledge of OpenModelica Shell commands and syntax.

Modelica and OpenModelica Shell in themselves have relatively little support for advanced analysis of models. Examples of such desirable analysis capabilities could be (i) study of model sensitivity, (ii) random number generation and statistical analysis, (iii) Monte Carlo simulation, (iv) advanced plotting capabilities, (v) general optimization capabilities, (vi) linear analysis and control synthesis, etc.

Scripting languages such as MATLAB and Python hold most of these desirable analysis capabilities, and it is of interest to integrate Modelica models with such script languages. The free *JModelica.org* tool includes a Python package for converting Modelica models to FMUs, and then for importing the FMU as a Python object. This way, Modelica models

[7] `www.openmodelica.org`

can essentially be simulated from Python — Optimica is also supported. It is possible to do more advanced analysis with JModelica.org[8] via *CasADi*, see e.g. [7] and [8]. However, the possibilities in the work of Perera et al. use an old version of JModelica.org. It would be more ideal if the possible extensions were supported by the tool developer.

1.5 Goal and status

It is of interest to develop an extension to OMPython which enables simulation and analysis of Modelica models with a better integration with the Python language, and in particular that such an extension is provided by the OpenModelica developers.

Modeling and the use of Modelica with Python is of interest to a wide range of engineering disciplines. The computer science threshold of using Modelica with Python should be low. Ideally, the OMPython extension should work with simple one-click Python installations such as Anaconda[9] and Canopy[10]. Furthermore, the extension should support both 32 bit and 64 bit OpenModelica, work with both 32 bit and 64 bit Python, with Python 2.7 and Python 3.X, and on platforms Windows, OSX and Linux. These requirements imply that results should be returned as standard Python structures. However, it is reasonable that the OMPython extension depends on the NumPy package. Because Python has excellent plotting capabilities e.g. via Matplotlib, the OpenModelica Shell facility for plotting results should not be implemented — this is more naturally handled directly in Python.

Currently, the Python API is under development and has been tested with 32 bit Python 2.7 from the Anaconda installation in tandem with 32 bit OpenModelica v. 1.9.4 under Windows 8.1 and OpenModelica v. 1.9.6 under Windows 10, using a modified `__init__.py` file. OpenModelica uses CORBA for communication, and CORBA compatibility needs some refinement. The code is somewhat unstable when run from the Spyder IDE in the Anaconda installation, but runs fine from Jupyter notebooks.

1.6 Structure of User's Guide

This User's Guide is organized as follows. Chapter 2 introduces as simple dynamic model of water level in a tank, and illustrates how OpenModelica can be used to simulate this system. This presentation is mainly motivational, but also serves to highlight the possibilities of the Python API, which is described in Chapter 3. In Chapter 4, the Python API is used to simulate and analyze the simple dynamic model of water level in a tank. In Chapter 5, a somewhat more complex case study is given based on a model from [9]. Finally, the new API is discussed in Chapter 6, and some conclusions are drawn.

Appendices hold (i) details of the model from [9], and (ii) Jupyter notebooks in pdf format. The Jupyter notebooks are provided separately. Finally, a bibliography list is given.

[8] `www.JModelica.org`
[9] `www.continuum.io/downloads`
[10] `www.enthought.com/products/canopy`

Chapter 2

OpenModelica case study

2.1 Case study: simple tank filled with water

We consider the tank in Fig. 2.1 filled with water.

Water with initial mass $m(0)$ is emptied by gravity through a hole in the bottom at effluent mass flow rate $\dot{m}_{\mathrm{e}}$, while at the same time water is filled into the tank at influent mass flow rate $\dot{m}_{\mathrm{i}}$.

Our *modeling objective* is to find the liquid level h. This objective is illustrated by the *functional diagram* in Fig. 2.2.

The functional diagram depicts the causality of the *system* ("Tank with influent and effluent mass flow"), where *inputs* (green arrow) cause a change in the system and is observed at *outputs* (orange arrow). Here, the input variable is the influent mass flow rate $\dot{m}_{\mathrm{i}}$, while the output variable is the quantity we are interested in, h.

2.2 Model summary

The model can be summarized in a form suitable for implementation in Modelica as

$$\begin{aligned} \frac{dm}{dt} &= \dot{m}_{\mathrm{i}} - \dot{m}_{\mathrm{e}} \\ m &= \rho V \\ V &= Ah \\ \dot{m}_{\mathrm{e}} &= K\sqrt{\frac{h}{h_{\mathrm{s}}}}. \end{aligned}$$

To complete the model description, we need to specify model constants and parameters, and operating conditions. Model constants and parameters are given in Table 2.1.

Table 2.1: Constants and parameters for water tank with fixed cross sectional area.

Parameter	**Value**	**Unit**	**Comment**
ρ	1	kg/L	Density of liquid; considered a *constant*
A	5	dm^2	Fixed cross sectional area
K	5	kg/s	Valve constant
h_{s}	3	dm	Level scaling

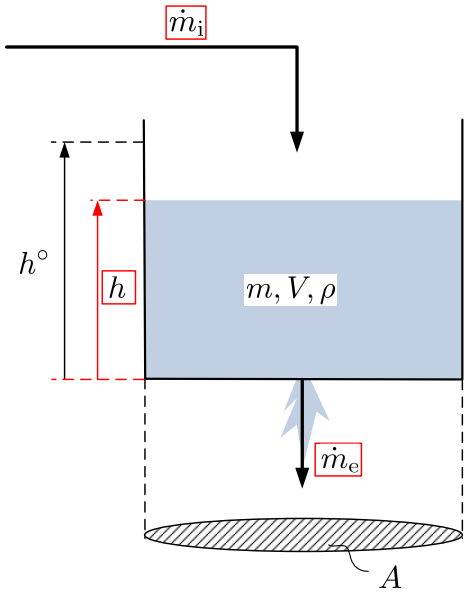


Figure 2.1: Water tank, with externally available quantities framed in red: initial mass is emptied through bottom at rate $\dot{m}_\mathrm{e}$, while at the same time water enters the tank at rate $\dot{m}_\mathrm{i}$.

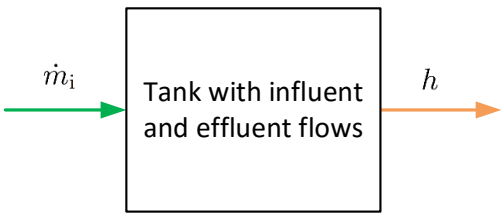


Figure 2.2: Functional diagram of tank with influent and effluent flow.

Table 2.2: Operating condition for water tank with fixed cross sectional area.

Quantity	Value	Unit	Comment
$h(0)$	1.5	dm	Initial level
$m(0)$	$\rho h(0) A$	kg	Initial mass (alternative initial condition)
$\dot{m}_{\mathrm{i}}$	2	kg/s	Nominal influent mass flow rate; may be varied

One comment on Table 2.1: the liquid density ρ would normally be categorized as a parameter — it is not a natural constant. However, for illustrative purposes, it is useful to include a constant in this first model. The operating conditions are given in Table 2.2.

2.3 Modelica encoding of model

The Modelica code describes the core model of the tank, `ModWaterTank`, and consists of a *first section* where constants and variables are specified, and a *second section* where the model equations are specified.

```
1  model ModWaterTank
2      // Main water tank model
3      // author:    Bernt Lie
4      //            University College of Southeast Norway
5      //            June 3, 2016
6      //
7      // Parameters
8      constant Real rho = 1 "Density of liquid, kg/L";
9      parameter Real A = 5 "Cross sectional area of tank, dm2";
10     parameter Real K = 5 "Valve constant, kg/s";
11     parameter Real h_s = 3 "Scaling level, dm";
12     // Initial state parameters
13     parameter Real h_0 = 1.5 "Initial tank level, dm";
14     parameter Real m_0 = rho*h_0*A "Initial tank mass, kg";
15     // Declaring variables
16     // -- states
17     Real m(start = m_0, fixed = true) "Mass in tank, kg";
18     // -- auxiliary variables
19     Real V "Tank liquid volume, L";
20     Real md_e "Effluent mass flow rate from tank, kg/s";
21     // -- input variables
22     input Real md_i "Influent mass flow rate to tank, kg/s";
23     // -- outut variables
24     output Real h "Tank liquid level, dm";
25 // Equations constituting the model
26 equation
27     // Differential equation
28     der(m) = md_i - md_e;
29     // Algebraic equations
30     m = rho*V;
31     V = A*h;
```

```
32      md_e = K*sqrt(h/h_s);
33  end ModWaterTank;
```

As seen from the *first part* of model `ModWaterTank`, the model has 1 "natural constant" `rho`,[1] and 3 parameters (`A`, `K`, `h_s`), compare this to Table 2.1. Furthermore, the model contains 2 "initial state" parameters, where one of them can be chosen at liberty, `h_0`, while the other one, `m_0`, is computed automatically from `h_0`, see Table 2.2. The purpose of using the "free parameter" `h_0` is that it is easier for the user to specify level than mass. Also, free "initial state" parameters makes it possible for the user to change the initial states from outside of model `ModWaterTank`, e.g., from Python.

Next, 1 variable is given with initial value — the state `m` — is initialized with the "initial state" parameter `m_0`. Then, 2 variables are defined as auxiliary variables (algebraic variables), `V` and `md_e`.[2]

One input variable is defined — `md_i` — this is the influent mass flow rate $\dot{m}_{\mathrm{i}}$, see Table 2.2. Inputs are characterized by that their values are not specified in model the core model — here `ModWaterTank`. Instead, their values must be given in an external model/code — we will specify this input in a simulation model and later in Python. Finally, 1 output is given — `h`.

In the *second part* of model `ModWaterTank`, the Model equations exactly map the mathematical model given in Section 2.2.

Next, we want to carry out *experiments* on the model, thus we need to specify the input $\dot{m}_{\mathrm{i}}$ — notice that $\dot{m}_{\mathrm{i}}$ is left indetermined in model `ModWaterTank`. To specify the experiment and prepare for simulating the system, we thus develop a second model which is meant for simulating the system — named `SimWaterTank`.

```
1  model SimWaterTank
2      // Experiment on water tank model
3      // author:    Bernt Lie
4      //            University College of Southeast Norway
5      //            June 3, 2016
6      //
7      // Instantiating model
8      ModWaterTank wt;        // Instantiating water tank
9      //
10     // Defining external input function
11     Real _md_i "External input, passed on to instantiated
           model, kg/s";
12     //
13 // Equations specifying the experiment
14 equation
15     _md_i = 2;              // Input as a function of time
16     //
17     wt.md_i = _md_i;        // Injecting input function into
           water tank model
18 end SimWaterTank;
```

[1] Normally, ρ (or `rho`) would be defined as a *parameter* — indicating that the model is valid for other fluids properties than $\rho = 1$. But to illustrate how constants are handled with the Python API, ρ is defined as a *constant* for illustrative purposes. The consequence is that the value of ρ can not be changed from Python.

[2] `md` is notation for m with a dot, $\dot{m}$, i.e., a mass flow rate.

In this simulation model, we *instantiate* a copy of model `ModWaterTank` — which we name `wt`. Next, we define an experimental function for $\dot{m}_\mathrm{i}$ named `_md_i` — just to emphasize that in principle this function is different from variable `md_i` within model `ModWaterTank`. Finally, we inject the experimental input function `_md_i` into variable `md_i` of model `wt`. Since input variable `md_i` of model `wt` must be referred to as `wt.md_i`, the equation for injecting `_md_i` into `wt.md_i` thus becomes `wt.md_i = _md_i`.

Since we have two models for the water tank, it is most convenient to put them in the same file. But if we have more than a single model in a file, we need to wrap them in a *package*. To this end, we wrap models `ModWaterTank` and `SimWaterTank` as defined above into a package named `WaterTank`, which must be stored in file `WaterTank.mo`:

```
1  package WaterTank
2      // Package for simulating driven water tank
3      // author:    Bernt Lie
4      //            University College of Southeast Norway
5      //            June 3, 2016
6      //
7      //
8      model SimWaterTank
9          // Experiment on water tank model
10         // ....
11         // ....
12     end SimWaterTank;
13     //
14     model ModWaterTank
15         // Main water tank model
16         // ....
17         // ....
18     end ModWaterTank;
19     // End package
20 end WaterTank;
```

2.4 OpenModelica simulation

We are now ready to simulate the system using OpenModelica. In OpenModelica, open the `WaterTank.mo` file using `File/Open Modelica Library File(s)` — or via clicking on the symbol in the menu line. In the OpenModelica Libraries Browser, we then observe the `WaterTank` behind package icon and the two models `SimWaterTank` and `ModWaterTank` behind model icon, Fig. 2.3.

By choosing the simulation model (click on it in the Library Browser; Fig. 2.3), we can first check that the model is correct. This checking is done from menu item `Simulation /Check Model` or by clicking on menu icon. In our case, the response is as in Fig. 2.4.

Next, we need to set up the simulation, menu item `Simulation/Simulation Setup` — or by clicking on menu icon. This opens up a menu as in Fig. 2.5.

In Fig. 2.5, the only change we introduce, is to change the Stop Time to 10. Finally, we click on the Simulate button in this menu. The result, after closing an information window and selecting some quantities for plotting, is shown in Fig. 2.6.

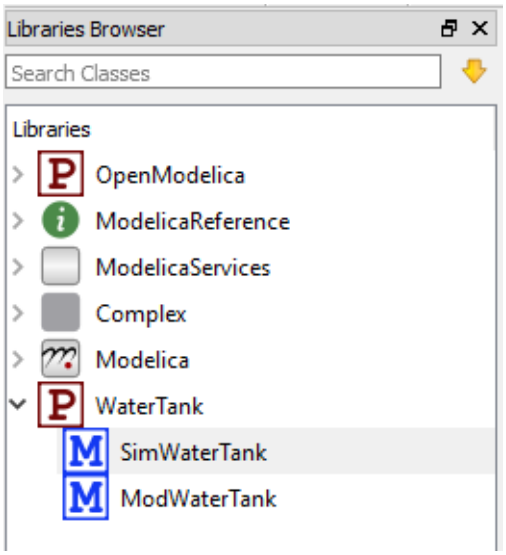


Figure 2.3: Library Browser of OpenModelica, with model `SimWaterTank` selected.

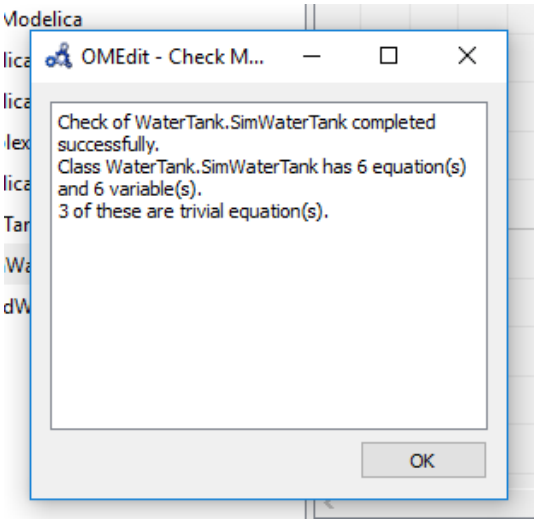


Figure 2.4: Checked model `SimWaterTank`.

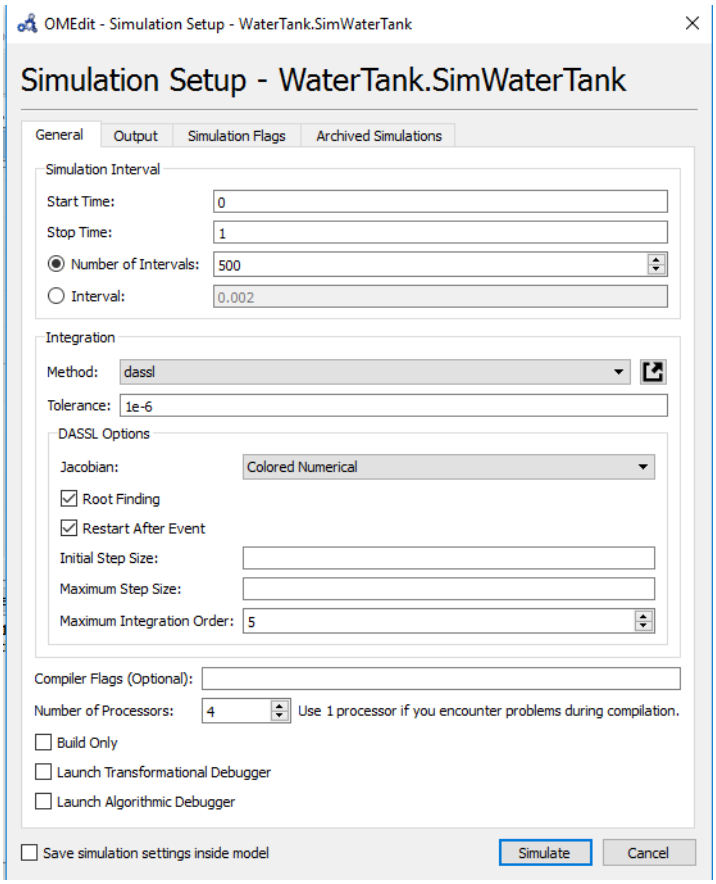


Figure 2.5: Menu for setting up simulation conditions for `SimWaterTank`. We only change the *Stop Time* from 1 to 10.

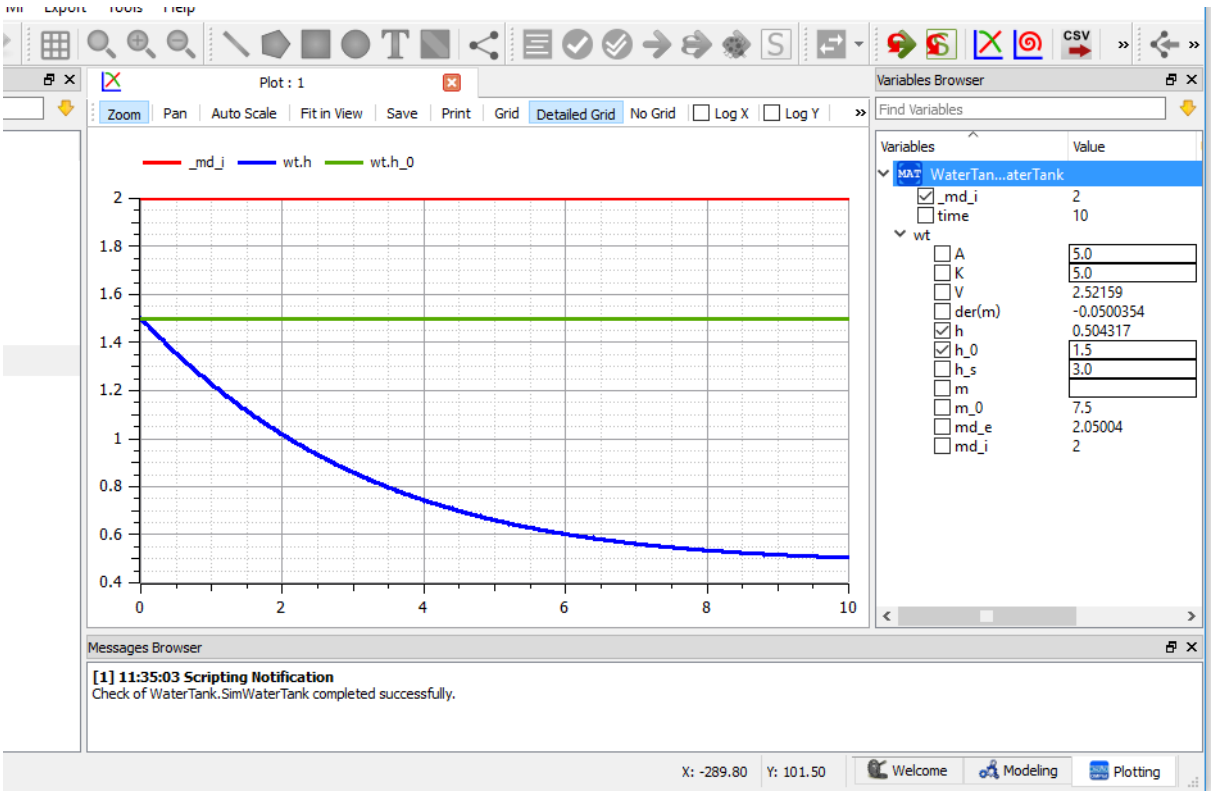
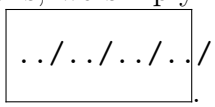


Figure 2.6: Results from simulating the water tank.

In Fig. 2.6, the line width/curve thickness has been changed. This can be done in menu `Tools/Options/Plotting`. If we want to re-do the simulation, we have to go back to the `Modeling` window, see the three tabs in the lower right corner of Fig. 2.6. If we want to change the simulation settings, we have to go back into the Simulation Setup menu. However, if we want to keep the simulation settings as-is, we simply run the command `Simulation/Simulate` — or click on the simulate icon 

.

Chapter 3

Python API

3.1 Installing the OMPython extension

3.1.1 Installation under Windows 10*

Under Windows, the new OMPython extension will be automatically installed in a file `__init__.py` in directory `shareomcscriptsPythonInterfaceOMPython` in the OpenModelica directory when OpenModelica is downloaded and installed. In order to activate the extension, the user must next run the command `python setup.py install` from the command line in the directory of the `setup.py` file, which is in the `PythonInterface` subdirectory. It follows that in order to activate the extension, the user must first install Python on the relevant computer. Under Linux/OSX, OMPython is part of `pip` (`pypi`) and is not shipped with the OpenModelica installer.

3.1.2 Installation under OSX*

...

3.1.3 Installation under Linux*

...

3.2 Description of the API

The API is described in the subsections below, where input argument `sj` indicates *string* no. *j*, input argument `S` indicates a *list of strings*, input argument `n` indicates a *scalar number*, input argument `N` indicates a *list of numbers*, and input argument `T` indicates a *list of tuples*.

3.2.1 Python Class and Constructor

The name of the Python *class* which is used for operation on Modelica models, is *ModelicaSystem*. This class is equipped with an object constructor of the same name as the class. In addition, the class is equipped with a number of methods for manipulating the instantiated objects.

In this subsection, we discuss how to import the class, and how to use the constructor to instantiate an object.

The object is imported from package *OMPython*, i.e. with Python commands[1]:

```
>>> from OMPython import ModelicaSystem
```

Other Python packages to be used such as `numpy`, `matplotlib`, `pandas`, etc. must be imported in a similar manner.

The object constructor requires a minimum of 2 input arguments which are strings, and may need a third string input argument.

- The *first input argument* must be a string with the file name of the Modelica code, with Modelica file extension `.mo`. If the Modelica file is not in the current directory of Python, then the file path must also be included.
- The *second input argument* must be a string with the name of the Modelica model, including the namespace if the model is wrapped within a Modelica package.
- *A third input argument* is used if the Modelica model builds on other Modelica code, e.g. the Modelica Standard Library.

The result of using the object constructor is a Python object.

Example 1 *Use of constructor.*

Suppose we have a Modelica model with name *CSTR* wrapped in a Modelica package *Reactors* — stored in file `Reactor.mo`:

```
package Reactors
    // ...
    model CSTR
        /// ...
    end CSTR;
    //
end Reactors;
```

If this model does not use any external Modelica code and the file is located in the current Python directory, the following Python code instantiates a Python object `mod`:

```
>>> mod = ModelicaSystem('Reactors.mo', 'Reactors.CSTR')
```

The user is free to choose any valid Python label name for the Python object.

All methods of class *ModelicaSystem* refers to the instantiated object, in standard Python fashion. Thus, method `simulate()` is invoked with the Python command:

```
>>> mod.simulate()
```

In the subsequent overview of methods, the object name is not included. In practice, of course, it must be included in order to operate on the object in question.

Methods may have no input arguments, one, or several input arguments. Methods may or may not return results — if the methods do not return results, the results are stored within the object.

[1]The Python prompt >>> is not typed, and does not appear in script files, in iPython or in Jupyter notebooks.

3.2.2 Utility routines, converting Modelica ↔ FMU

Two utility methods convert files between Modelica files with file extension `.mo` and Functional Mock-up Unit (FMU) files with file extension `.fmu`.

1. `convertMo2Fmu()` — method for converting the Modelica model of the object, say `ModelName`, into FMU file.
   - Required input arguments: none, operates on the Modelica file associated with the object.
   - Optional input arguments:[2]
     - `className`: string with the class name that should be translated,
     - `version`: string with FMU version, "1.0" or "2.0"; the default is "1.0".
     - `fmuType`: string with FMU type, "me" (model exchange) or "cs" (co-simulation); the default is "me".
     - `fileNamePrefix`: string; the default is \"className\".
     - `generatedFileName`: string, returns the full path of the generated FMU.
   - Result: file `ModelName.fmu` in the current directory
2. `convertFmu2Mo(s)` — method for converting an FMU file into a Modelica file.
   - Required input arguments: string `s`, where `s` is name of FMU file, including extension `.fmu`.
   - Optional input arguments: a number of optional input arguments, e.g. the possibility to change working directory for the imported FMU files.
   - Result: Assume the name of the file is `fmuName.fmu`. Then file `fmuName_me_FMU.mo` is generated in the current Python directory.

3.2.3 Getting and setting information

Quite a few methods are dedicated to getting and setting information about objects. With two exceptions — `getQuantities()` and `getSolutions()` — the *get* methods have identical use of input arguments and results, while all the set methods have identical use of input arguments, with results stored in the object.

3.2.3.1 Getting quantity information

Method `getQuantities()` does not accept input arguments, and returns a *list of dictionaries*, one dictionary for each quantity. Each dictionary has the following keys — with values being strings, too.

- `Changeable` — value `'true'` or `'false'`,
- `Description` — the string used in Modelica to describe the quantity, e.g. `'Mass in tank, kg'`,
- `Name` — the name of the quantity, e.g. `'T'`, `'der(T)'`, `'n[1]'`, `'mod1.T'`, etc.,

[2]The optional input arguments have not been implemented yet; they are set to their default value.

- `Value` — the value of the quantity, e.g. `'None'`, `'5.0'`, etc.,
- `Variability` — `'continuous'`, `'parameter'`.

When applying the *Pandas* method *DataFrame* to the returned list of dictionaries, the result is a conveniently typeset table in Jupyter notebooks. Modelica `constants` are not included in the returned quantities.

3.2.3.2 Standard get methods

We consider methods `getXXXs()`, where `XXXs` is in `{Continuous, Parameters, Inputs, Outputs, SimulationOptions, OptimizationOptions, LinearizationOptions}`. Thus, methods `getContinuous()`, `getParameters()`, etc.

Two calling possibilities are accepted.

- `getXXXs()`, i.e. without input argument, returns a *dictionary* with names as keys and values as ... values.
- `getXXXs(S)`, where `S` is a *sequence* of strings of names, returns a *tuple* of values for the specified names.

3.2.3.3 Getting solutions

We consider method `getSolutions()`. Two calling possibilities are accepted.

- `getSolutions()`, i.e. without input arguments, returns a *list of strings of names* of quantities for which there is a *solution = time series*.[3]
- `getSolutions(S)`, where `S` is a *sequence* of strings of names, returns a *tuple* of values = 1D numpy arrays = time series for the specified names.

3.2.3.4 Setting methods

The information that can be set is a subset of the information that can be set. Thus, we consider methods `setXXXs()`, where `XXXs` is in `{Parameters, Inputs, SimulationOptions, OptimizationOptions, LinearizationOptions}`, thus methods `setParameters()`, `setInputs()`, etc. Two calling possibilities are accepted.

- `setXXXs(K)`, with `K` being a sequence of keyword assignments of type `quantity name = value`. Here, the quantity name could be a parameter name (i.e., not a string), an input name, etc.
  - For parameters and simulation/optimization/linearization options, the value should be a numerical value or a string (e.g. a string of ODE solver name such as `'dassl'`, etc.).
  - For inputs, the value could be a numerical value if the input is constant in the time range of the simulation,

[3] The reason why a dictionary with every name as key and time series as values is not returned, is that the amount of data would be exhaustive.

  - For inputs, the value could alternatively be a *list* of tuples (t_j, u_j), i.e.,

$$[(t_1, u_1), (t_2, u_2), \ldots, (t_N, u_N)]$$

    where the input varies linearly between (t_j, u_j) and (t_{j+1}, u_{j+1}), where $t_j \leq t_{j+1}$, and where at most two subsequent time indices t_j, t_{j+1} can have the same value. As an example, `[..., (1,10), (1,20), ...]` describes a perfect jump in input value from value 10 to value 20 at time instance 1.
  - This type of sequence of input arguments does not work for certain quantity names, e.g. `'der(T)'`, `'n[1]'`, `'mod1.T'`, because Python does not allow for label names `der(T)`, `n[1]`, `mod1.T`, etc.

- `setXXXs(**D)`, with `D` being a *dictionary* with quantity names as keywords and values as described with the alternative input argument `K`.

3.2.4 Operating on Python object: simulation, optimization

The following methods operate on the object, and have no *input arguments*. The methods have no return values, instead the results are stored within the object.

- `simulate()` — simulates the system with the given simulation options
- `optimize()` — optimizes the Optimica problem with the given optimization options

To retrieve the results, method `getSolutions()` is used as described previously.

3.2.5 Operating on Python object: linearization

The following methods are proposed for linearization[4]:

- `linearize()` — with no input argument, returns a tuple of 2D numpy arrays (matrices) A, B, C and D.
- `getLinearInputs()` — with no input argument, returns a list of strings of names of inputs used when forming matrices B and D.
- `getLinearOutputs()` — with no input argument, returns a list of strings of names of outputs used when forming matrices C and D.
- `getLinearStates()` — with no input argument, returns a list of strings of names of states used when forming matrices A, B, C and D.

[4]This part of the API is not completed at the moment, and may change.

Chapter 4

Python API for simulation

4.1 Use of Python API

Because of problems with running the code from the Spyder API as of this writing[1], the code is run from a Jupyter notebook[2]. In *all cases*, the following Python/iPython statements are run initially in the notebook.

```
from OMPython import ModelicaSystem
import numpy as np
import numpy.random as nr
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
LW1 = 2.5
LW2 = LW1/2
Cb1 = (0.3,0.3,1)
Cb2 = (0.7,0.7,1)
Cg1 = (0,0.6,0)
Cg2 = (0.5,0.8,0.5)
Cr1 = "Red"
Cr2 = (1,0.5,0.5)
LS1 = "solid"
LS2 = "dotted"
figpath = "C:/Users/berntl/.../figs/"
```

Here, we use **NumPy** to handle simulation results, etc. The **random** number package will be used in a sensitivity/Monte Carlo study. The *magic* function `%matplotlib inline` is used to *embed* **Matplotlib** plots in the Jupyter notebook; it is possible to save these plots to file by right-clicking on the plot — the result is really ugly, so instead we use the `savefig` method after each plot has been designed. **Pandas** are used to present data in tables in the Jupyter notebook. Finally, labels `LW1` and `LW2` are used to give a conform line widths in plots, labels `Cb1`, `Cb2`, `Cg1`, `Cg2`, `Cr1` and `Cr2` specify colors,[3] labels `LS1`

[1]Possibly some problem for Spyder to handle CORBA.

[2]Some virus programs may flag a possible problem due to the CORBA connection activity between Python and the executable file. As an example, F-Secure flags a possible problem the first time a Python object is simulated after it has been instantiated. To continue, just choose that the activity should be accepted. Other virus programs, such as Windows Defender, does not flag this activity.

[3]The standard blue color appears very dark — close to black – on printed paper. Thus, blue with

and `LS2` may be used to give conform line styles, while label `figpath` sets the path where figures are stored.

4.2 Water tank without external inputs

4.2.1 Learning goals

The *learning goals* in this section are basic instantiation of a Python object, checking quantities, running the simulation, and extracting/plotting results. We also consider how to set quantities with valid Modelica names which are illegal as Python labels.

4.2.2 Instantiation of Python object

We consider the model of a water tank as described in Chapter 2, i.e. without external inputs. In other words: we consider the model named `SimWaterTank`.

In the sequel, Python prompt `>>>` is used when Jupyter notebook actually uses `In[*]` — where `*` is some number, while the response is given without any prompt; in Jupyter notebook the response is prepended with `Out[*]`. We *instantiate* object `tnk` with the following command:

```
>>> tnk = ModelicaSystem("WaterTank.mo", "WaterTank.
    SimWaterTank")
```

whereupon Python/Jupyter notebook responds that the OMC Server is up and running the file,

```
2016-06-13 15:51:20,242 - OMCSession - INFO - OMC Server is
    up and running at file:///c:\users\berntl\appdata\local\
    temp\openmodelica.objid.a107f76206e040378a69393719dcc490
```

4.2.3 Quantities of Python object

Next, we are interested in which *quantities* are available in the model.

```
>>> q = tnk.getQuantities()
>>> type(q)
list
>>> len(q)
12
>>> q[0]
{'Changeable': 'true',
 'Description': 'Mass in tank, kg',
 'Name': 'wt.m',
 'Value': None,
 'Variability': 'continuous'}
>>> pd.DataFrame(q)
```

The last command leads Jupyter notebook to typeset a tabular presentation of the quantities, Fig. 4.1. The results in Fig. 4.1 should be compared to the Modelica model in Section 2.3. Observe that:

increased transparency is chosen, etc. Increased line width also helps to make the colors stand out.

In [9]:
```
pd.DataFrame(q)
```

Out[9]:

	Changeable	Description	Name	Value	Variability
0	true	Mass in tank, kg	wt.m	None	continuous
1	false	Mass in tank, kg	der(wt.m)	None	continuous
2	false	External input, passed on to instantiated mode...	_md_i	None	continuous
3	false	Tank liquid level, dm	wt.h	None	continuous
4	false	Effluent mass flow rate from tank, kg/s	wt.md_e	None	continuous
5	true	Cross sectional area of tank, dm2	wt.A	5.0	parameter
6	true	Valve constant, kg/s	wt.K	5.0	parameter
7	true	Initial tank level, dm	wt.h_0	1.5	parameter
8	true	Scaling level, dm	wt.h_s	3.0	parameter
9	false	Initial tank mass, kg	wt.m_0	None	parameter
10	false	Tank liquid volume, L	wt.V	None	continuous
11	false	Influent mass flow rate to tank, kg/s	wt.md_i	None	continuous

Figure 4.1: Typesetting of Data Frame of quantity list in Jupyter notebook, case of no external inputs.

1. Modelica *constants* (e.g. `rho`) are not included in the quantity list,
2. Since the quantities of object `tnk` originate from model `SimWaterTank`, the quantity names of `ModWaterTank` are *prepended* with the name of the instantiated model in `SimWaterTank`: `"wt."`, e.g. `wt.m`, `der(wt.m)`, etc.

4.2.4 Inputs

Because model `SimWaterTank` doesn't have external inputs, we should not find any inputs. We can check this:

```
>>> tnk.getInputs()
{}
```

As there are no inputs, an empty Python dictionary (`{}`) is returned.

4.2.5 Simulation options and simulation

Next, we check the simulation options:

```
>>> sopt = tnk.getSimulationOptions()
>>> type(sopt)
dict
>>> sopt
{'solver': 'dassl',
 'startTime': 0.0,
 'stepSize': 0.002,
 'stopTime': 1.0,
```

```
 'tolerance': 1e-06}
```

It should be observed that the `stepSize` is the frequency at which solutions are *stored*, and is *not* the step size of the *solver*. The number of data points stored, is thus `(stopTime-startTime)/stepSize` with due rounding. This means that if we *increase* the `stopTime` to a large number, we should also *increase* the `stepSize` to avoid storing a large number of data points.

Here, we want to simulate the system with a stop time of 10 s:

```
>>> tnk.setSimulationOptions(startTime = 0, stopTime = 10)
```

Finally, we simulate the system:

```
>>> tnk.simulate()
```

4.2.6 Extracting and plotting simulation results

If we do not specify for which variable we want to extract the solution, the result is a *list* of names of possible "variables" for which there is a solution:

```
>>> sol = tnk.getSolutions()
>>> sol
['time',
 'wt.m',
 'der(wt.m)',
 '_md_i',
 'wt.h',
 'wt.md_e',
 'wt.V',
 'wt.md_i',
 'wt.A',
 'wt.K',
 'wt.h_0',
 'wt.h_s',
 'wt.m_0']
```

Observe that this list includes variable `time` which holds the time instances for which the other variables have a value. Also observe that the "solution" of parameters (e.g., `wt.A`, `wt.K`, etc.) is available — these should be constant, and are normally not very interesting. Finally: the reason why the time evolution for the variables is not returned when no variable is specified is simply that the amount of data is prohibitive except for extremely small models.

Suppose now that we want to check the time evolution of level h. To do this, we need to extract both the `time` variable and the level `wt.h`:

```
>>> sol = tnk.getSolutions("time", "wt.h")
>>> type(sol)
tuple
>>> sol
(array([  0.00000000e+00,   2.00000000e-03,   4.00000000e-03, ...,
          9.99800000e+00,   1.00000000e+01,   1.00000000e+01]),
 array([ 1.5       ,  1.49938593,  1.49877215, ...,  0.50433763,
         0.5043176 ,  0.5043176 ]))
```

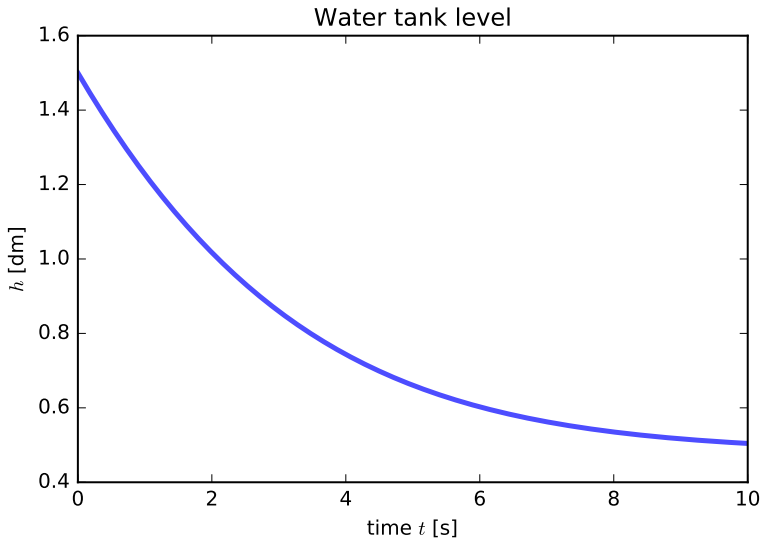


Figure 4.2: Level h as a function of time t based on model `SimWaterTank`.

We can now use the Python method for *unpacking* tuple `sol`:

```
>>> tm, h = sol
>>> type(tm)
numpy.ndarray
>>> np.shape(tm)
(5002,)
>>> tm.shape
(5002,)
```

However, we could just as well have unpacked the solution directly:

```
>>> tm, h = tnk.getSolutions("time", "wt.h")
```

We are now ready to plot the result. This can be done as follows:

```
>>> plt.plot(tm,h,linewidth=LW,color=Cb1,label=r"$h$")
>>> plt.title("Water␣tank␣level")
>>> plt.xlabel(r"time␣$t$␣[s]")
>>> plt.ylabel(r"$h$␣[dm]")
>>> figfile = "levelSimWaterTank.pdf"
>>> plt.savefig(figpath+figfile)
```

The result is as in Fig. 4.2.

We can also study other variables in the list `sol`, e.g. $\frac{dm}{dt}$ (or: `wt.der(m)`):

```
>>> tm, dmdt = tnk.getSolutions("time","der(wt.m)")
>>> plt.plot(tm,dmdt,linewidth=LW,color=Cg1,label=r"$\frac{dm
   }{dt}$")
>>> plt.title("Change␣of␣accumulated␣mass")
>>> plt.xlabel(r"time␣$t$␣[s]")
>>> plt.legend(loc=0)
>>> figfile = "massChangeSimWaterTank.pdf"
>>> plt.savefig(figpath+figfile)
```

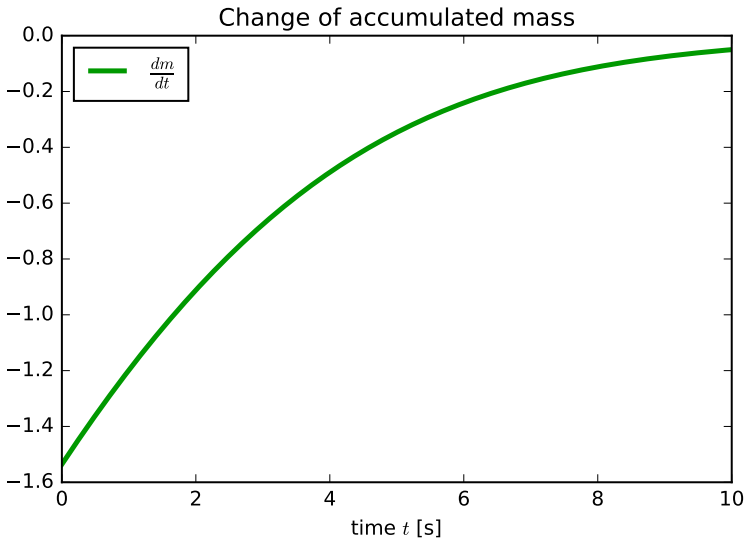


Figure 4.3: Change of accumulated mass, $\frac{dm}{dt}$, as a function of time t.

The result is shown in Fig. 4.3.

4.2.7 Quantities with names which are illegal Python labels

Modelica allows for names of quantities which are illegal as Python labels, e.g., Fig. 4.1. This creates a problem when we want to set their values.

```
>>> tnk.getParameters()
{'wt.A': 5.0, 'wt.K': 5.0, 'wt.h_0': 1.5, 'wt.h_s': 3.0, 'wt.
   m_0': None}
>>> tnk.setParameters(wt.A=6,wt.K=3)
  File "<ipython-input-93-2433dcbe5a08>", line 1
    tnk.setParameters(wt.A=6,wt.K=3)
SyntaxError: keyword can't be an expression
```

The reason for this problem is that in Python, symbol "." indicates that what follows, is a method/function or an attribute. Thus, `wt.A` would be interpreted as applying attribute `A` to label `wt` — in other words, an expression. In Modelica, on the other hand, `wt.A` implies quantity `A` in object `wt`, thus in Modelica `wt.A` is a valid variable name. Similarly, in Modelica, `der(wt.A)` is a valid variable name. In Python, on the other hand, `der(wt.A)` would be interpreted as using function `der` on attribute `A` of label `wt`.

To get around this syntax conflict, we have to pose the setting of values as a dictionary:

```
>>> d = d = {"wt.A": 6, "wt.K": 3}
>>> tnk.setParameters(**d)
>>> tnk.getParameters()
{'wt.A': 6.0, 'wt.K': 3.0, 'wt.h_0': 1.5, 'wt.h_s': 3.0, 'wt.
   m_0': None}
```

Note that setting values via a dictionary and using the `**d` argument is a more general method in Python. For the case when Modelica quantity names are valid Python label names, we can use the simpler argument form.

```
In [30]: pd.DataFrame(tnk.getQuantities())
```

Out[30]:

	Changeable	Description	Name	Value	Variability
0	true	Mass in tank, kg	m	None	continuous
1	false	Mass in tank, kg	der(m)	None	continuous
2	false	Tank liquid level, dm	h	None	continuous
3	false	Effluent mass flow rate from tank, kg/s	md_e	None	continuous
4	false	Influent mass flow rate to tank, kg/s	md_i	None	continuous
5	true	Cross sectional area of tank, dm2	A	5.0	parameter
6	true	Valve constant, kg/s	K	5.0	parameter
7	true	Initial tank level, dm	h_0	1.5	parameter
8	true	Scaling level, dm	h_s	3.0	parameter
9	false	Initial tank mass, kg	m_0	None	parameter
10	false	Tank liquid volume, L	V	None	continuous

Figure 4.4: Typesetting of Data Frame of quantity list in Jupyter notebook, case with external inputs.

4.3 Water tank with external input

4.3.1 Learning goals

The *learning goals* in this section are to get and set inputs, including how to start the model in steady state.

To this end, we consider the more interesting problem of studying the model with input, i.e., we instantiate model `ModWaterTank`:

```
>>> tnk = ModelicaSystem("WaterTank.mo", "WaterTank.
    ModWaterTank")
```

4.3.2 Quantities of Python object

It is of interest to constrast the *quantities* available in this model (`ModWaterTank`) with those of model `SimWaterTank`:

```
>>> pd.DataFrame(tnk.getQuantities())
```

The last command leads Jupyter notebook to typeset a tabular presentation of the quantities, Fig. 4.4. The results in Fig. 4.4 for model `ModWaterTank` should be compared to the Modelica model in Section 2.3, and also with the results in Fig. 4.1 for model `SimWaterTank`. Observe that:

1. Modelica *constants* (e.g. `rho`) are not included in the quantity list — which is the same as in Fig. 4.1,

2. The quantities of object `tnk` originate from model `ModWaterTank`, and do not contain the object name `wt` that appears in the quantities in Fig. 4.1.

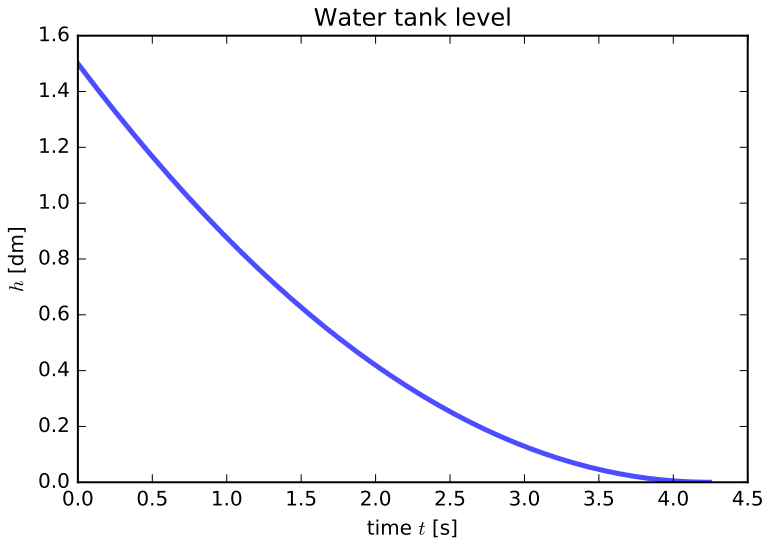


Figure 4.5: Level h as a function of time t based on model `ModWaterTank`, when no input has been specified.

If I'm interested in the details of a *particular* quantity, and can query for that quantity, e.g.:

```
>>> tnk.getQuantities("wt.m")
[{'Changeable': 'true',
  'Description': 'Mass in tank, kg',
  'Name': 'wt.m',
  'Value': None,
  'Variability': 'continuous'}]
```

4.3.3 Simulation without caring about the input

First, we forget that this new object should have inputs, and simply observe what happens.

```
>>> tnk.setSimulationOptions(stopTime = 10)
>>> tnk.simulate()
```

We are now ready to extract and plot the results:

```
>>> tm, h = tnk.getSolutions("time", "h")
>>> plt.plot(tm,h,linewidth=LW, color=Cb1, label=r"$h$")
>>> plt.title("Water tank level")
>>> plt.xlabel(r"time $t$ [s]")
>>> plt.ylabel(r"$h$ [dm]")
>>> figfile = "levelModWaterTank0.pdf"
>>> plt.savefig(figpath+figfile)
```

The result is as in Fig. 4.5.

As Fig. 4.5 is different from Fig. 4.2, a different input must have been used. We can check the input:

```
>>> u = tnk.getInputs()
>>> type(u)
```

```
dict
>>> u.keys()
['md_i']
>>> u.values()
[None]
>>> u
{'md_i': None}
```

To see which value has actually been used for input $\dot{m}_\mathrm{i}$, we have to extract the solution for `md_i` from the simulation:

```
>>> tnk.getSolutions('md_i')
array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

This indicates that an unspecified input defaults to zero.

4.3.4 Setting the input for the simulation

In model `SimWaterTank`, the input is set to $\dot{m}_\mathrm{i} = 2\,\mathrm{kg/s}$. We can achieve this for model `ModWaterTank` as follows:

```
>>> tnk.setInputs(md_i=2)
```

The simulation result should now be comparable to that in Fig. 4.2:

```
>>> tnk.simulate()
>>> tm, h = tnk.getSolutions("time","h")
>>> plt.plot(tm,h,linewidth=LW,color=Cb1,label=r"$h$")
>>> plt.title("Water tank level")
>>> plt.xlabel(r"time $t$ [s]")
>>> plt.ylabel(r"$h$ [dm]")
>>> figfile = "levelModWaterTank1.pdf"
>>> plt.savefig(figpath+figfile)
```

See the result in Fig. 4.6.

Next, we want to set a time varying input:

```
>>> u_md_i = [(0,2),(3,2),(3,4),(7,5),(10,4)]
>>> tnk.setInputs(md_i = u_md_i)
>>> tnk.getInputs()
{'md_i': [(0, 2), (3, 2), (3, 4), (7, 5), (10, 4)]}
```

The interpretation of the input specification is a straight line between the points (t_j, u_j). Observe that we can specify a jump at t_j by letting two subsequent points have $t_{j+1} = t_j$, e.g., as the pints `(3,2)` and `(3,4)`.

Next, we simulate the system with the new input, and show the results.

```
>>> tnk.simulate()
>>> tm, h = tnk.getSolutions("time","h")
>>> plt.plot(tm,h,linewidth=LW,color=Cb1,label=r"$h$")
>>> plt.title("Water tank level")
>>> plt.xlabel(r"time $t$ [s]")
>>> plt.ylabel(r"$h$ [dm]")
```

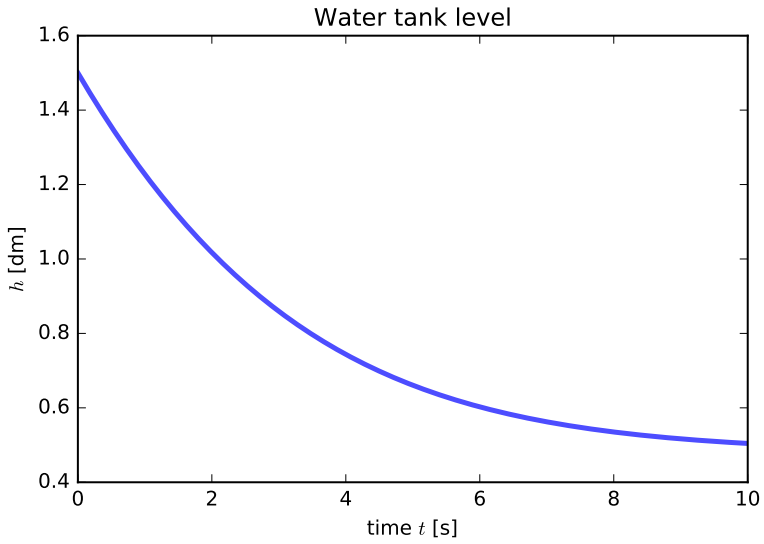


Figure 4.6: Level h as a function of time t based on model `ModWaterTank`, when the input $\dot{m}_\mathrm{i}$ has been specified by `md_i=2`. The result is identical to that in Fig. 4.2.

```
>>> figfile = "levelModWaterTank2.pdf"
>>> plt.savefig(figpath+figfile)
```

The result is as in Fig. 4.7.

The input is now found to be:

```
>>> tm, md_i = tnk.getSolutions("time","md_i")
>>> plt.plot(tm,md_i,linewidth=LW,color=Cr1,label=r"$\dot{m}_
   {\mathrm{i}}$")
>>> plt.title("Water tank level")
>>> plt.xlabel(r"time $t$ [s]")
>>> plt.ylabel(r"$\dot{m}_{\mathrm{i}}$ [kg/s]")
>>> plt.ylim((1.5,5.5))
>>> figfile = "inputModWaterTank2.pdf"
>>> plt.savefig(figpath+figfile)
```

see Fig. 4.8.

Here, it should be observed that we *could* have extracted both variables at the same time:

```
>>> tm, h, md_i = tnk.getSolutions("time","h","md_i")
```

or alternatively without re-extracting the time vector the second time:

```
>>> tm, h = tnk.getSolutions("time","h")
>>> md_i = tnk.getSolutions("md_i")
```

4.3.5 Starting the simulation in steady state

Quite often, we are interested in the transient based on a given input — when we start the simulation in steady state. Modelica does have a construct for enforcing initiation

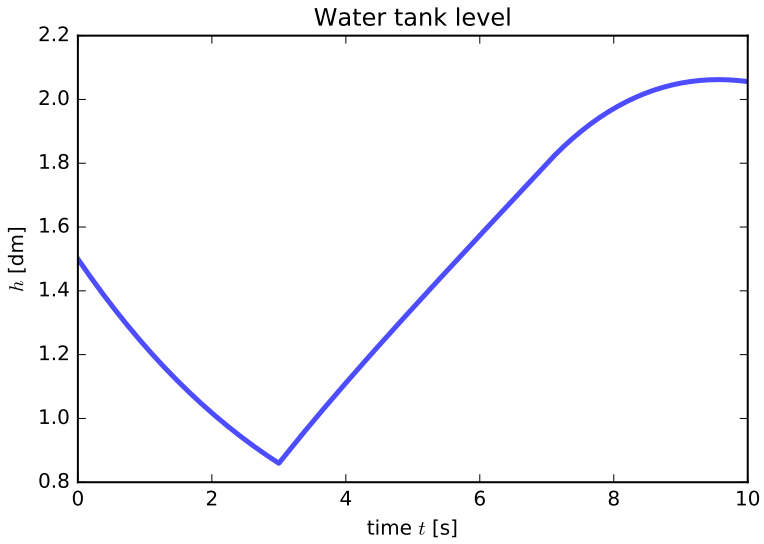


Figure 4.7: Level h as a function of time t based on model `ModWaterTank`, when the input $\dot{m}_\mathrm{i}$ has been specified as straight lines going through the points `[(0,2),(3,2),(3,4),(7,5),(10,4)]`.

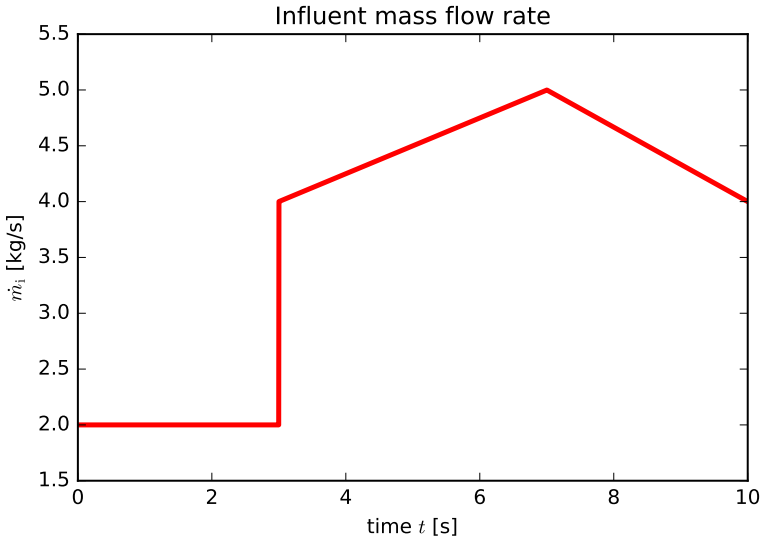


Figure 4.8: Level $\dot{m}_\mathrm{i}$ as a function of time t: the input goes through the points `[(0,2),(3,2),(3,4),(7,5),(10,4)]`.

in steady state: we can specify an `initial equation` block. However, with an `initial equation` block, Modelica attempts to solve a set of nonlinear algebraic equations, and this doesn't always work well for large models.[4]

An alternative for finding the steady state is to simulate the system with *stop time* equal to infinity (or: a very large vaue), with a fixed input. When using a large stop time, it is important to also choose a large *step size* to avoid excessive amount of data.

The following sequence of Python commands can be used to simulate the system for a long time, extract the steady state, and initiate the steady state. Observe that we can specify the initial *level* given by `h_0` (instead of the initial mass `m_0`; the mass is the actual state).

First, we set the input which we want to start with, $\dot{m}_{\mathrm{i}} = 3\,\mathrm{kg/s}$ and simulate the system for a long time:

```
>>> tnk.setInputs(md_i=3)
>>> tnk.setSimulationOptions(stopTime=1e4,stepSize=10)
>>> tnk.simulate()
```

Next, we extract the level (not the state m, but the level h — since we indirectly set initial state $m\,(0)$ via setting the initial level $h\,(0)$):

```
>>> h = tnk.getSolutions("h")
>>> h[-1]
2.055960822758669
>>> tnk.setParameters(h_0=h[-1])
```

Observe that we set parameter `h_0` to be equal to the last value of $h\,(t)$ in the solution `h`; the last value is given by `h[-1]` in Python/NumPy.

Observe that we could have found the end-time value in other ways. We can ask for the (current values) of the continuous quantities:

```
>>> tnk.getContinuous()
{'V': 10.27980411379334,
 'der(m)': -0.1392036500179881,
 'h': 2.055960822758669,
 'm': 10.27980411379334,
 'md_e': 4.1392036500179881,
 'md_i': 4.0}
```

or we can restrict the request to the continuous quantity we are interested in:

```
>>> tnk.getContinuous("h")
2.055960822758669
```

Since we are interested in variable `h`, which also happens to be an *output*, we can also find the current value as follows:

```
>>> tnk.getOutputs()
{'h': 2.055960822758669}
```

Next, we specify the simulation options we want to use, specify the input which is to start at $\dot{m}_{\mathrm{i}}\,(0) = 3$, and simulate the system:

```
>>> tnk.setInputs(md_i=[(0,3),(2,3),(2,4),(6,4),(6,2),(10,2)
   ])
```

[4] It works for *small* models.

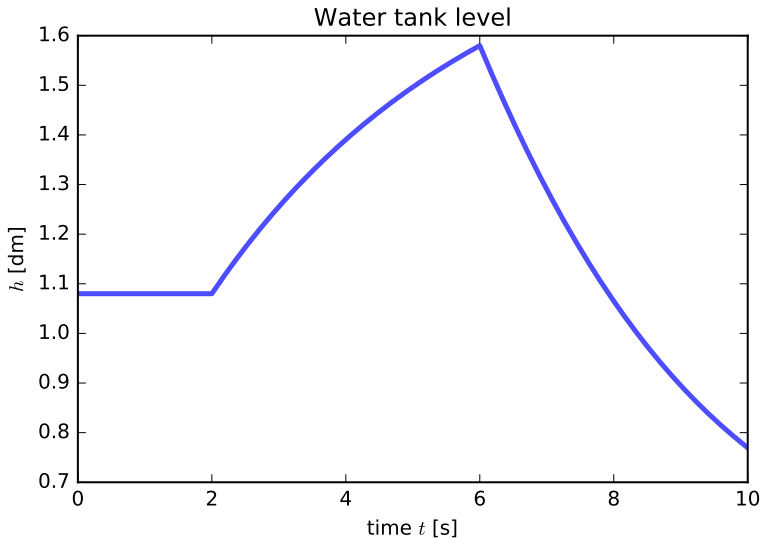


Figure 4.9: Level h as a function of time t based on model `ModWaterTank`, when the input $\dot{m}_\mathrm{i}$ has been specified as straight lines going through the points `[(0,3),(2,3),(2,4),(6,4),(6,2),(10,2)]`. Observe that (i) we start in steady state, and (ii) the input is different from the one in Fig. 4.7.

```
>>> tnk.setSimulationOptions(stopTime=10,stepSize=0.02)
>>> tnk.simulate()
```

Finally, we extract the solution and plot it:

```
>>> tm, h = tnk.getSolutions("time","h")
>>> plt.plot(tm,h,linewidth=LW,color=Cb1,label=r"$h$")
>>> plt.title("Water tank level")
>>> plt.xlabel(r"time $t$ [s]")
>>> plt.ylabel(r"$h$ [dm]")
>>> figfile = "levelModWaterTank3.pdf"
>>> plt.savefig(figpath+figfile)
```

The resulting level $h\,(t)$ is as indicated in Fig. 4.9.

4.4 Water tank parameter sensitivity

The *learning goal* in this section is how to combine the Modelica model with simple analysis using a Python package.

With outputs $y \in \mathbb{R}^{n_y}$ and parameters $\theta \in \mathbb{R}^{n_\theta}$, we could consider sensitivities defined as $s_\theta = \frac{\partial y}{\partial \theta} \in \mathbb{R}^{n_y \times n_\theta}$. Here, we instead consider Monte Carlo simulation, i.e. how outputs (or states) vary with uncertain parameters θ. In particular, we will study how the water level varies with uncertain effluent valve constant K. This can be done as follows. We first check possible parameters in the model:

```
>>> par = tnk.getParameters())
>>> type(par)
dict
```

```
>>> par.keys()
['A', 'K', 'm_0', 'h_s', 'h_0']
>>> par.values()
[5.0, 5.0, None, 3.0, 1.08]
```

Next, we extract the parameter value of parameter `K` (on the Modelica side) and set it as a nominal parameter `K` on the Python side:

```
>>> K = par["K"]
```

We then use the random number generator in Python/Numpy to generate an array of 50 random values for the effluent valve constant K and store these in array `KK` in Python. These numbers are uniformly distributed in the interval `K`$\times \left(1 \pm \frac{0.5}{10}\right)$.

```
>>> KK = K*(1 + (nr.randn(50)-0.5)/10)
```

First we simulate and plot the system using nominal value `K`:

```
>>> tnk.simulate()
>>> tm, h = tnk.getSolutions("time","h")
```

We delay plotting this nominal solution, to make sure that it is plotted on top of the other lines. Thus, next, we set up a Python for loop and find the solution for each valve constant in array `KK`, as well as plot the result:

```
>>> for k in KK:
        tnk.setParameters(K=k)
        tnk.simulate()
        tm1, h1 = tnk.getSolutions("time","h")
        plt.plot(tm1,h1,linewidth=LW/2,color=Cb2,linestyle="
           dotted",label="_nolabel_")
>>> plt.plot(tm,h,linewidth=LW,color=Cb1,label=r"$h$")
>>> plt.title("Tank level sensitivity")
>>> plt.xlabel(r"time $t$ [s]")
>>> plt.ylabel(r"$h$ [dm]")
>>> plt.legend()
>>> figfile = "levelModWaterTank4.pdf"
>>> plt.savefig(figpath+figfile)
```

The result is as shown in Fig. 4.10

Observe that we set the value of parameter `K` using the syntax:

```
>>> tnk.setParameters(K=k)
```

This works because the Modelica quantity name `K` (see Fig. 4.4) is a legal Python label. We could, of course, have use the more general (and less elegant) form

```
>>> tnk.setParameters(**{"K": k})
```

which would give the same result.

4.5 Nonlinear chemical reactor

4.5.1 Learning goals

The *learning goals* in this section are how to study a slightly more complex model with two states, which exhibits strong nonlinear effects and starts in an unsteady operating

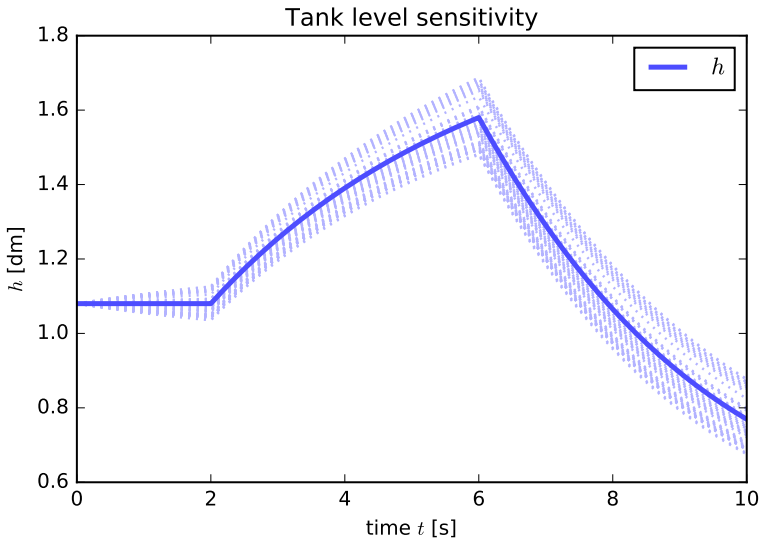


Figure 4.10: Uncertainty in tank level with a 10% uncertainty in valve constant K (uncertainty: $\pm 5\%$).

condition. To this end, we use the model of a nonlinear continuous stirred tank reactor from [9].

4.5.2 Model of CSTR

Consider the Continuous Stirred Tank Reactor (CSTR) in Example 2.5 of [9], which can be described with the model[5]

$$\begin{aligned}\frac{dn_{\mathrm{A}}}{dt} &= \dot{V}\left(c_{\mathrm{Ai}} - \frac{n_{\mathrm{A}}}{V}\right) - rV \\ \rho V \hat{c}_p \frac{dT}{dt} &= \rho \hat{c}_p \dot{V}\left(T_{\mathrm{i}} - T\right) + \left(-\Delta \tilde{H}_{\mathrm{r}}\right) rV + \dot{Q},\end{aligned}$$

where

$$\begin{aligned}r &= k_0 \exp\left(-\frac{E/R}{T}\right)\frac{n_{\mathrm{A}}}{V} \\ \dot{Q} &= UA\left(T_{\mathrm{ci}} - T\right).\end{aligned}$$

The standard state space form of such models is

$$\begin{aligned}\frac{dx}{dt} &= f\left(x, u; \theta\right) \\ y &= g\left(x, u; \theta\right)\end{aligned}$$

where $x \in \mathbb{R}^{n_x}$, $u \in \mathbb{R}^{n_u}$, $y \in \mathbb{R}^{n_y}$, and $\theta \in \mathbb{R}^{n_\theta}$. Here, x are the state variables of the system, u the input variables which are determined outside of the system, and y are the output variables from the system; θ are the parameters (constants) of the system. For

[5]The model notation is changed from that of [9], and the model formulation is slightly different (different states), but the model here is equivalent to that in the reference.

this reactor system:

$$x = (n_{\mathrm{A}}, T)\,, \qquad u = \left(\dot{V}, c_{\mathrm{Ai}}, T_{\mathrm{i}}, T_{\mathrm{ci}}\right)$$
$$y = (c_{\mathrm{A}})\,, \qquad \theta = \left(V, \Delta\tilde{H}_{\mathrm{r}}, \rho, \hat{c}_p, k_0, \frac{E}{R}, UA\right).$$

Here, we could in principle choose to make E (activation energy) and R (gas constant) two independent parameters, but in the model, it is the fraction E/R which is of importance. Likewise, we could choose to consider U (overall heat transfer coefficient), A (heat transfer area) two independent parameters, but in the model it is their product that is important. Furthermore, we have chosen to set the output equal to c_{A}. We could have chosen to expand y with various measured/sensor variables.

The following model parameters are given, Table 4.1.

Table 4.1: Numerical values for parameters and operating points for the CSTR. *As the original publication uses c_{A} as state, $n_{\mathrm{A}}\,(t=0)$ is not specified there.

Quantity	**Symbol**	**Nominal value**
Vessel volume	V	$100\,\mathrm{L}$
Preexponential factor	k_0	$\exp\left(\frac{8750}{350}\right) \approx 7.2 \times 10^{10}\,\mathrm{min}^{-1}$
Activation "temperature"	E/R	$8750\,\mathrm{K}$
Reaction enthalpy	$-\Delta\tilde{H}_{\mathrm{r}}$	$5 \times 10^4\,\mathrm{J/mol}$
Density of reactor fluid	ρ	$1000\,\mathrm{g/L}$
Specific heat capacity of reactor fluid	$\hat{c}_p$	$0.239\,\mathrm{J/(g\,K)}$
Heat transfer parameter	UA	$5 \times 10^4\,\mathrm{J/(min\,K)}$
Initial value, concentration of A	$c_{\mathrm{A}}\,(t=0)$	$0.5\,\mathrm{mol/L}$
Initial mole number, A*	$n_{\mathrm{A}}\,(t=0)$	$= c_{\mathrm{A}}(t=0) \cdot V\,\mathrm{mol}$
Initial value, temperature	$T\,(t=0)$	$350\,\mathrm{K}$
Volumetric feed flow, nominal	$\dot{V}$	$100\,\mathrm{L/min}$
Influent concentration, nominal	c_{Ai}	$1\,\mathrm{mol/L}$
Influent temperature, nominal	T_{i}	$350\,\mathrm{K}$
Coolant feed temperature, nominal	T_{ci}	$300\,\mathrm{K}$

In Example 2.5 of [9], the preexponential factor is in fact given as $k_0 = 7.2 \times 10^{10}\,\mathrm{min}^{-1}$. However, because the system starts in an unsteady operating point, it is important to choose $k_0 = \exp\left(\frac{8750}{350}\right)\,\mathrm{min}^{-1}$ in order to maintain the system in the operating point.

4.5.3 Modelica encoding of model

The Modelica code describes the core model of the tank, `ModReactor`, which is embedded in package `Reactor` in file `Reactor.mo`. Model `ModReactor` consists of a *first section* where constants and variables are specified, and a *second section* where the model equations are specified.

```
1 package Reactor
2     // Package for study of nonlinear reactor adapted from
         Seborg et al.
3     // author:     Bernt Lie
4     //             University College of Southeast Norway
```

```
5       //              June 16, 2016
6       //
7       model ModReactor
8           // Reactor model in Ex 2.5 of Seborg et al. (2011),
9           // Process Dynamics and Control
10          //
11          // author: Bernt Lie
12          //         University College of Southeast Norway
13          //         June 16, 2016
14          //
15          // Parameters
16          parameter Real V = 100 "Reactor vessel volume; L";
17          parameter Real k0 = exp(8750/350) "Preexponential
                factor,
18                                                reaction; 1/min";
19          parameter Real EdR = 8750 "Activation 'temperature';
                K";
20          parameter Real dHr = -5e4 "Specific reaction enthalpy
                ; J/mol";
21          parameter Real rho = 1e3 "Density, reactor fluid; g/L
                ";
22          parameter Real cp = 0.239 "Specific heat capacity,
                reactor fluid; J/(g.K)";
23          parameter Real UA = 5e4 "Heat transfer parameter; J/(
                min.K)";
24          // Initial state parameter
25          parameter Real cA0 = 0.5 "Initial cA; mol/L";
26          parameter Real nA0 = cA0*V "Initial 'real' state nA;
                mol";
27          parameter Real T0 = 350 "Initial T; K";
28          // Declaring variables
29          // -- states
30          Real nA(start = nA0, fixed = true); // initial #
                moles of A in vessel; mol
31          Real T(start = T0, fixed = true);   // initial
                temperature in vessel; K
32          // -- auxiliary variables
33          Real ndAi "Influent of A; mol/min";
34          Real ndAe "Effluent of A; mol/min";
35          Real ndAg "Generation rate of A; mol/min";
36          Real r "Rate of reaction; mol/(L.min)";
37          Real Qd "Heat flow from cooling fluid to reactor
                vessel; J/min";
38          // Inputs
39          input Real Vd "Volumetric flow through reactor vessel
                ; L/min";
40          input Real cAi "Reactor vessel influent concentration
                 of A; mol/min";
```

```
41          input Real Ti "Reactor vessel influent temperature; K
               ";
42          input Real Tci "Cooling circuit influent temperature;
                K";
43          // Outputs
44          output Real cA "Reactor concentration; mol/L";
45      equation
46          nA = V*cA;
47          ndAi = Vd*cAi;
48          ndAe = Vd*cA;
49          ndAg = -V*r;
50          r = k0*exp(-EdR/T)*cA;
51          Qd = UA*(Tci-T);
52          //
53          der(nA) = ndAi - ndAe + ndAg;
54          V*rho*cp*der(T) = rho*cp*Vd*(Ti-T) + (-dHr)*r*V + Qd;
55      end ModReactor;
56      // End package
57  end Reactor;
```

4.5.4 Nominal study

We consider the nominal case where input cooling temperature is $T_{\mathrm{ci},\triangleright} = 300\,\mathrm{K}$, as well as the "hot" cooling temperature $T_{\mathrm{ci},\triangleright} + 5$ and the "cool" cooling temperature $T_{\mathrm{ci},\triangleright} - 10$ — as in [9]. First we instantiate the model and set simulation options:

```
>>> r0 = ModelicaSystem("Reactor.mo","Reactor.ModReactor")
>>> r0.setSimulationOptions(stopTime=10,stepSize=0.05)
```

Next, we set the nominal inputs:

```
>>> r0.setInputs(Tci=300)
>>> r0.setInputs(Ti=350)
>>> r0.setInputs(Vd=100)
>>> r0.setInputs(cAi=1)
```

Then, we simulate and extract results for the three cases:

```
>>> r0.simulate()
>>> tm0, T0, Tc0, cA0 = r0.getSolutions("time","T","Tci","cA
   ")
>>> r0.setInputs(Tci=300+5)
>>> r0.simulate()
>>> tm0p, T0p, Tc0p, cA0p = r0.getSolutions("time","T","Tci
   ","cA")
>>> r0.setInputs(Tci=300-10)
>>> r0.simulate()
>>> tm0m, T0m, Tc0m, cA0m = r0.getSolutions("time","T","Tci
   ","cA")
```

To study the temperature behavior, we plot the temperatures:

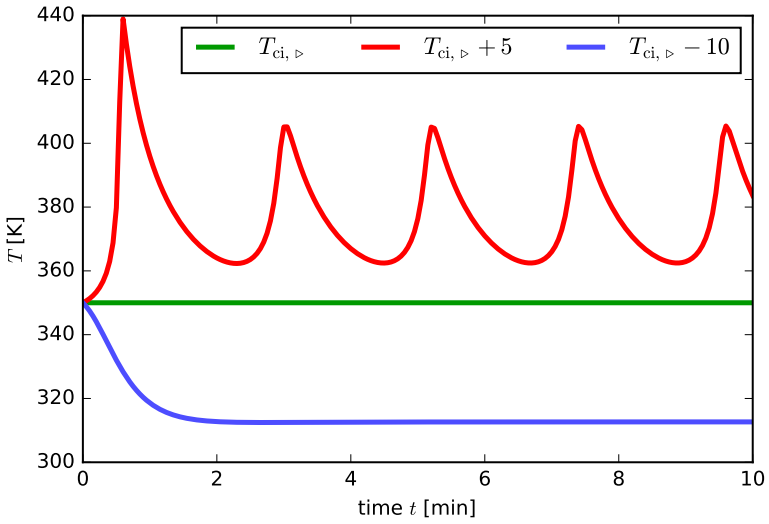


Figure 4.11: Nominal reactor temperatures T for the three input coolant temperatures.

```
>>> plt.plot(tm0,T0,linewidth=LW1,color=Cg1,label=r"$T_{\
   mathrm{ci},\triangleright}$")
>>> plt.plot(tm0p,T0p,linewidth=LW1,color=Cr1,label=r"$T_{\
   mathrm{ci},\triangleright}+5$")
>>> plt.plot(tm0m,T0m,linewidth=LW1,color=Cb1,label=r"$T_{\
   mathrm{ci},\triangleright}-10$")
>>> plt.legend(ncol=3)
>>> plt.xlabel(r"time␣$t$␣[min]")
>>> plt.ylabel(r"$T$␣[K]")
>>> figfile = "tempSeborgReactor.pdf"
>>> plt.savefig(figpath+figfile)
```

The resulting evolution of reactor temperature T is displayed in Fig. 4.11.
We can also check the concentration c_{A}:

```
>>> plt.plot(tm0,cA0,linewidth=LW1,color=Cg1,label=r"$T_{\
   mathrm{ci},\triangleright}$")
>>> plt.plot(tm0p,cA0p,linewidth=LW1,color=Cr1,label=r"$T_{\
   mathrm{ci},\triangleright}+5$")
>>> plt.plot(tm0m,cA0m,linewidth=LW1,color=Cb1,label=r"$T_{\
   mathrm{ci},\triangleright}-10$")
>>> plt.legend()
>>> plt.xlabel(r"time␣$t$␣[min]")
>>> plt.ylabel(r"$c_{\mathrm{A}}$␣[mol/L]")
>>> figfile = "concSeborgReactor.pdf"
>>> plt.savefig(figpath+figfile)
```

see results in Fig. 4.12.

For completeness, the input T_{ci} is:

```
>>> plt.plot(tm0,Tc0,linewidth=LW1,color=Cg1,label=r"$T_c^{\
   triangleright}$")
```

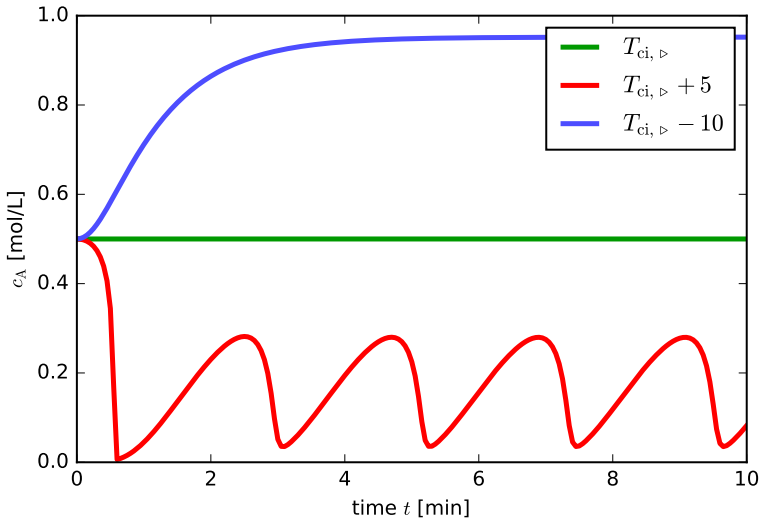


Figure 4.12: Nominal reactor concentration c_A for the three input coolant temperatures.

```
>>> plt.plot(tm0p,Tc0p,linewidth=LW1,color=Cr1,label=r"$T_c
   ^{\triangleright}$+5")
>>> plt.plot(tm0m,Tc0m,linewidth=LW1,color=Cb1,label=r"$T_c
   ^{\triangleright}-10$")
>>> plt.title("Cooling␣liquid␣temperature")
>>> plt.xlabel(r"time␣$t$␣[s]")
>>> plt.ylabel(r"$T_c$␣[K]")
>>> plt.axis(ymin=288,ymax=307)
>>> plt.legend(ncol=3,loc=7)
>>> figfile = "coolSeborgReactor.pdf"
>>> plt.savefig(figpath+figfile)
```

as in Fig. 4.13.

4.5.5 Varying input

As the nominal study has shown, the system starts in an equilibrium point, but is unsteady in this point. Thus, a tiny excitation makes the system drift away from the initial state. Because the system is nonlinear, it does not drift to infinity, but instead is attracted to other solutions — to a new steady state in the case of input $T_{ci,\triangleright} - 10$, and to an oscillatoric stationary solution in the case of input $T_{ci,\triangleright} + 5$.

It may be of interest to see how the system behaves under the influence of time varying inputs. To this end, we consider variations in inputs T_{ci} and T_i:

```
>>> uTci = [(0,300),(2,300),(2,300-10),(5,300-10),(5,300)
   ,(7,300),(7,300+5),(30,300+5)]
>>> uTi = [(0,350),(15,350),(15,340),(20,340),(20,360)
   ,(30,360)]
```

Reusing the previously specified inputs for `Vd` and `cAi`, we change the inputs T_{ci} and T_i:

```
>>> r0.setInputs(Tci=uTci)
```

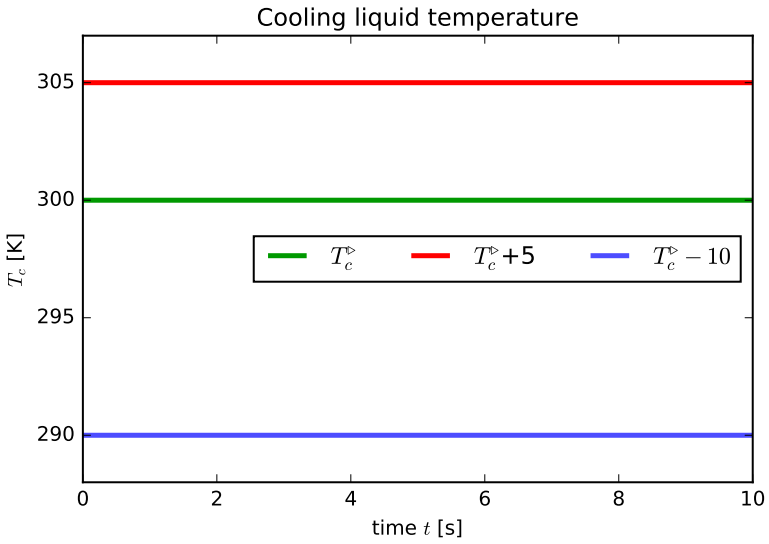


Figure 4.13: The three input coolant temperatures.

```
>>> r0.setInputs(Ti=uTi)
```

We also reuse the `stepSize` and only change the `stopTime` in the simulation:

```
>>> r0.setSimulationOptions(stopTime=30)
>>> r0.simulate()
```

The result is then plotted:

```
>>> tm, T, Tc, Ti = r0.getSolutions("time","T","Tci","Ti")
>>> plt.plot(tm,Tc,linewidth=LW1,color=Cg1,label=r"$T_{\
   mathrm{c}}$")
>>> plt.plot(tm,Ti,linewidth=LW1,color=Cb1,label=r"$T_{\
   mathrm{i}}$")
>>> plt.plot(tm,T,linewidth=LW1,color=Cr1,label=r"$T$")
>>> plt.title("Temperatures")
>>> plt.xlabel(r"time $t$ [min]")
>>> plt.ylabel("temperature [K]")
>>> plt.legend()
>>> figfile = "varInputSeborgReactor.pdf"
>>> plt.savefig(figpath+figfile)
```

with result as in Fig. 4.14.

4.5.6 Sensitivity/Monte Carlo

It is of interest to see how sensitive the model solution is to variations in the heat transfer product UA. To this end, we study the solutions of 20 random values of UA around the nominal value $\overline{UA}$, drawn from a uniform distribution and with a 10% uncertainty. Because of the dramatically different dynamics for input $T_{\mathrm{ci},\triangleright} + 5$ and input $T_{\mathrm{ci},\triangleright} - 10$, we study both cases. First we instansiate to models, and set their simulation options and their inputs:

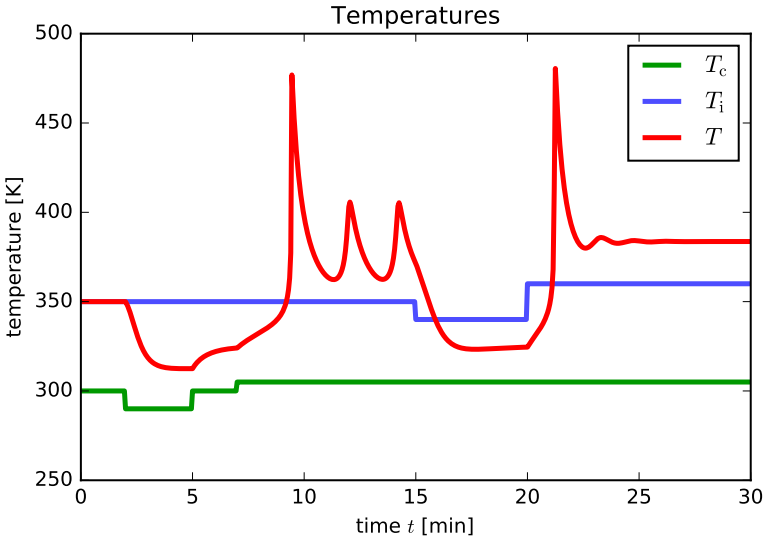


Figure 4.14: Coolant input temperature T_c, reactor influent temperature T_i and reactor temperature T.

```
>>> r0p = ModelicaSystem("Reactor.mo","Reactor.ModReactor")
>>> r0m = ModelicaSystem("Reactor.mo","Reactor.ModReactor")

>>> r0p.setSimulationOptions(stopTime=10,stepSize=0.05)
>>> r0m.setSimulationOptions(stopTime=10,stepSize=0.05)

>>> r0p.setInputs(Tci=300+5)
>>> r0m.setInputs(Tci=300-10)
>>> r0p.setInputs(Ti=350,Vd=100,cAi=1)
>>> r0m.setInputs(Ti=350,Vd=100,cAi=1)
```

Next, we query the default/nominal value $\overline{UA}$ and generate 20 random numbers for UA:[6]

```
>>> p = r0p.getParameters()
>>> UA = p["UA"]
>>> UUAA = UA*(1 + (nr.randn(20)-0.5)/10)
```

Finally, we run the simulations and plot the results.

```
>>> r0p.simulate()
>>> tm0p,T0p = r0p.getSolutions("time","T")
>>> r0m.simulate()
>>> tm0m,T0m = r0m.getSolutions("time","T")
#
>>> for ua in UUAA:
        r0p.setParameters(UA=ua);
        r0p.simulate()
        tm0p1,T0p1 = r0p.getSolutions("time","T")
        r0m.setParameters(UA=ua);
        r0m.simulate()
```

[6]The two model instantiations have the same default parameter values.

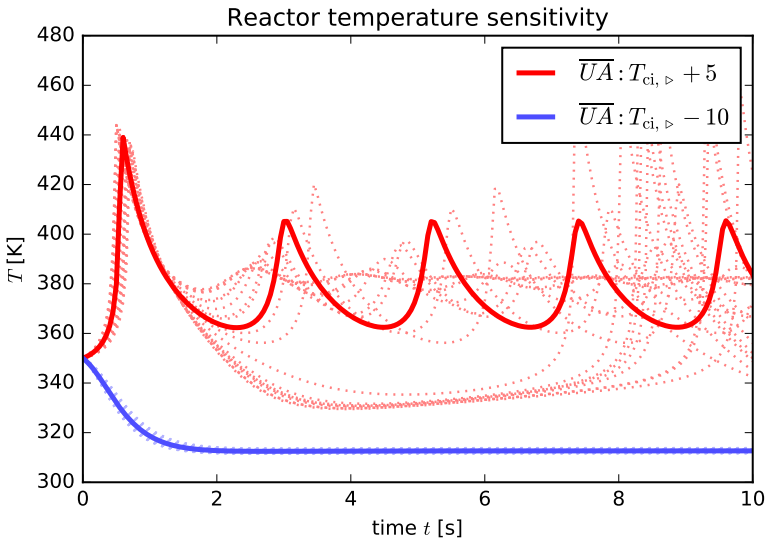


Figure 4.15: Reactor temperature T for various heat transfer parameters UA.

```
        tm0m1,T0m1 = r0m.getSolutions("time","T")
        plt.plot(tm0p1,T0p1,linewidth=LW2,color=Cr2,linestyle
           =LS2,label="_nolabel_")
        plt.plot(tm0m1,T0m1,linewidth=LW2,color=Cb2,linestyle
           =LS2,label="_nolabel_")
#
>>> plt.plot(tm0p,T0p,linewidth=LW1,color=Cr1,label=r"$\
   overline{UA}␣:␣T_{\mathrm{ci},\triangleright}+5$")
>>> plt.plot(tm0m,T0m,linewidth=LW1,color=Cb1,label=r"$\
   overline{UA}␣:␣T_{\mathrm{ci},\triangleright}-10$")
#
>>> plt.title("Reactor␣temperature␣sensitivity")
>>> plt.xlabel(r"time␣$t$␣[s]")
>>> plt.ylabel(r"$T$␣[K]")
>>> plt.legend()
>>> figfile = "sensitivitySeborgReactor.pdf"
>>> plt.savefig(figpath+figfile)
```

The result is displayed in Fig. 4.15.

4.6 Carbon dioxide level in class room

4.6.1 Learning goals

The *learning goals* in this section are how to study an even more complex model with three states, with hybrid inputs in the sense that some are integers and others are real numbers. To this end, we use a model of CO_2 level in a classroom.[7] In previous sections,

[7]The model is taken from a project in course *FM1015 Modeling of Dynamic Systems* at University College of Southeast Norway.

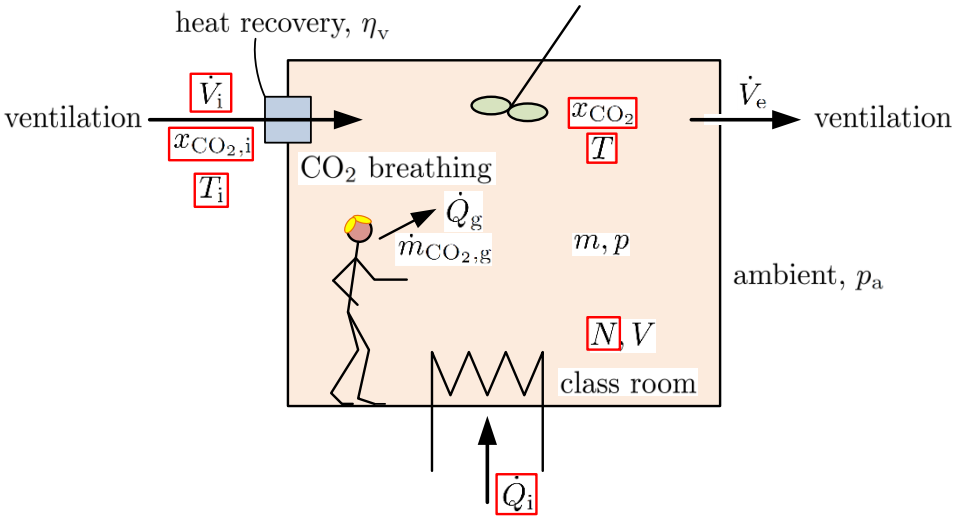


Figure 4.16: Ventilated classroom with students breathing CO_2.

we have illustrated how to simulate uncertain *parameters*. Here, we illustrate how to simulate uncertain *input sequences*.

4.6.2 Problem description

We consider the ventilated classroom in Fig. 4.16, with N students.

The room is ventilated with an influent volumetric flow rate $\dot{V}_i$ where the influent mole fraction of CO_2 is $x_{CO_2,i}$ and the influent temperature is T_i. The mole fraction of CO_2 in the room is x_{CO_2}, and the temperature of the room is T. The room is heated with heat flow $\dot{Q}_i$. If we consider a cold climate (e.g. a Nordic country in the winter), ventilation leads to cooling of the room, and it is of interest to recover heat at a fraction η_v;[8] $\eta_v = 1$ implies full heat recovery of the ventilation loss, while $\eta_v = 0$ implies no heat recovery.

The mass rate generation of CO_2 by breathing in the room is assumed to be given as

$$\dot{m}_{CO_2,g} = \dot{m}^1_{CO_2,g} \cdot N$$

where $\dot{m}^1_{CO_2,g}$ is the nominal breathing rate per person. The heat generated by the presence of people is given as

$$\dot{Q}_g = \dot{Q}^1_g \cdot N$$

where $\dot{Q}^1_g$ is the nominal heat produced per person.

The effluent mass flow rate is driven by the pressure difference from the classroom to the ambient,

$$\dot{m}_e = K_e \left(p - p_a\right),$$

where p is the total pressure in the classroom and p_a be the ambient/atmospheric pressure. The air in the classroom can be assumed to be ideal gas. $p_{CO_2,i}$ is the partial pressure of CO_2 in the influent/atmosphere.

It is of interest to find out how the ventilation given by influent volumetric flow rate $\dot{V}_i$, influent CO_2 fraction $x_{CO_2,i}$, the number of people N present, and the heating $\dot{Q}_i$ influences the air quality in the class room as measured by the CO_2 fraction x_{CO_2} and the temperature T. Clearly, $x_{CO_2,i}$ changes on a much slower timescale than $\dot{V}_i$, N and

[8] Recovered heat is returned to the room.

Table 4.2: Possible parameters for study of air quality in classroom.

Parameter	Value	Unit	Comment
$\hat{c}_p$	1	$\frac{\text{kJ}}{\text{kg K}}$	Heat capacity of air
η_v	0.6	—	Ventilation heat recovery
$\hat{H}^\circ$	0	$\frac{\text{kJ}}{\text{kg}}$	Standard state specific enthalpy
K_e	$\frac{100}{1.01\times10^5}$	$\frac{\text{kg}}{\text{s Pa}}$	Effluent mass flow "valve" constant
$\dot{m}^1_{\text{CO}_2,\text{g}}$	$\frac{1}{24\cdot3600}$	$\frac{\text{kg}}{\text{s persons}}$	CO_2 breathing per person, 1 kg/d
M_air	29	$\frac{\text{kg}}{\text{kmol}}$	Molar mass of air
M_{CO_2}	$12+2\cdot16$	$\frac{\text{kg}}{\text{kmol}}$	Molar mass of CO_2
p_a	1.01×10^5	Pa	Atmospheric pressure
$\dot{Q}^1_\text{g}$	100	W	Generation of heat per person
R	8.31	$\frac{\text{kJ}}{\text{kmol K}}$	Gas constant
T°	298.15	K	Standard state temperature
V	$45\cdot3$	m^3	Volume of classroom

Table 4.3: Operating conditions for study of air quality in classroom.

Quantity	Value	Unit	Comment
$p(0)$	p_a	Pa	Initial pressure in classroom
$m(0)$	$\frac{VM_\text{air}}{RT}p(0)$	kg	Initial air mass in classroom
$T(0)$	20	°C	Initial temperature in classroom
$x_{\text{CO}_2}(0)$	400	ppm	Initial CO_2 mole fraction in classroom
$p_{\text{CO}_2}(0)$	$x_{\text{CO}_2}(0)\cdot p(0)$	Pa	Initial partial pressure of CO_2
$n_{\text{CO}_2}(0)$	$\frac{V}{RT}p_{\text{CO}_2}(0)$	kmol	Initial number of moles of CO_2 in classroom
N	15	—	Number of persons in classroom
$\dot{Q}_\text{i}$	0	W	Heating
T_i	10	°C	Influent temperature
$\dot{V}_\text{i}$	$\frac{4V}{3600}$	m^3/s	Influent volumetric flow rate, typically $4V$ per 1 h
$x_{\text{CO}_2,\text{i}}$	400	ppm	Influent molar concentration of CO_2

T_i. Still, predictions of future CO_2 fraction $x_{\text{CO}_2,\text{i}}$ makes it interesting to investigate how a climate change may influence future ventilation policies and costs.

Possible parameters for use in a model are given in Table 4.2.

Possible operating conditions (initial state, input) are given in Table 4.3.

4.6.3 Model of CO_2 level

A *functional diagram* indicating inputs and outputs for a model for studying x_{CO_2} and T, and how these depend on $\dot{V}_\text{i}$, $x_{\text{CO}_2,\text{i}}$, T_i, N, and $\dot{Q}_\text{i}$ is given in Fig. 4.17.

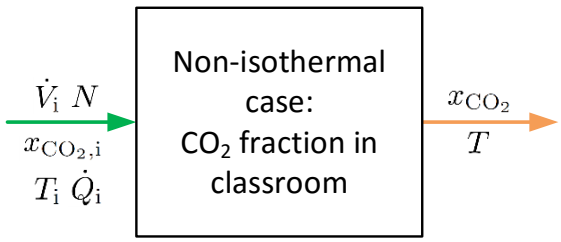


Figure 4.17: Functional diagram relating inputs to outputs for CO_2 fraction and temperature in classroom.

The model can by summarized as

$$\begin{aligned}
\frac{dm}{dt} &= \dot{m}_\mathrm{i} - \dot{m}_\mathrm{e} \\
\frac{dn_{\mathrm{CO}_2}}{dt} &= \dot{n}_{\mathrm{CO}_2,\mathrm{i}} - \dot{n}_{\mathrm{CO}_2,\mathrm{e}} + \dot{n}_{\mathrm{CO}_2,\mathrm{g}} \\
\frac{dU}{dt} &= \dot{H}_\mathrm{i} - \dot{H}_\mathrm{e} + \dot{Q}.
\end{aligned}$$

Here,

$$\begin{aligned}
m &= nM_\mathrm{air} \\
p_\mathrm{a}\dot{V}_\mathrm{i} &= \dot{n}_\mathrm{i}RT_\mathrm{i} \\
\dot{m}_\mathrm{i} &= \dot{n}_\mathrm{i}M_\mathrm{air} \\
\dot{m}_\mathrm{e} &= K_\mathrm{e}\left(p - p_\mathrm{a}\right) \\
pV &= nRT.
\end{aligned}$$

Furthermore,

$$\begin{aligned}
p_{\mathrm{CO}_2,\mathrm{i}} &= x_{\mathrm{CO}_2,\mathrm{i}}p_\mathrm{a} \\
p_{\mathrm{CO}_2,\mathrm{i}}\dot{V}_\mathrm{i} &= \dot{n}_{\mathrm{CO}_2,\mathrm{i}}RT_\mathrm{i} \\
\dot{n}_\mathrm{e} &= \frac{\dot{m}_\mathrm{e}}{M_\mathrm{air}} \\
x_{\mathrm{CO}_2} &= \frac{n_{\mathrm{CO}_2}}{n} \\
\dot{n}_{\mathrm{CO}_2,\mathrm{e}} &= x_{\mathrm{CO}_2}\dot{n}_\mathrm{e} \\
\dot{n}_{\mathrm{CO}_2,\mathrm{g}} &= \frac{\dot{m}_{\mathrm{CO}_2,\mathrm{g}}}{M_{\mathrm{CO}_2}} \\
\dot{m}_{\mathrm{CO}_2,\mathrm{g}} &= \dot{m}^1_{\mathrm{CO}_2,\mathrm{g}}N
\end{aligned}$$

Finally,

$$\begin{aligned}
U &= H - pV \\
H &= m\hat{H} \\
\hat{H} &= \hat{H}(T) = \hat{H}^\circ + \hat{c}_p (T - T^\circ) \\
\hat{H}_\mathrm{i} &= \hat{H}(T_\mathrm{i}) = \hat{H}^\circ + \hat{c}_p (T_\mathrm{i} - T^\circ) \\
\dot{H}_\mathrm{i} &= \dot{m}_\mathrm{i} \left(\hat{H}_\mathrm{i} + \eta_\mathrm{v} \left(\hat{H} - \hat{H}_\mathrm{i} \right) \right) \\
\dot{H}_\mathrm{e} &= \dot{m}_\mathrm{e} \hat{H} \\
\dot{Q} &= \dot{Q}_\mathrm{i} + \dot{Q}_\mathrm{g} \\
\dot{Q}_\mathrm{g} &= \dot{Q}_\mathrm{g}^1 N.
\end{aligned}$$

Here, the standard state is normally $(T^\circ, p^\circ) = (298.15\,\mathrm{K}, 1.01 \times 10^5\,\mathrm{Pa})$. For ideal gas, p° is not needed. Since no reaction takes place, we set $\hat{H}^\circ = 0$.

This model can be written in standard DAE form as

$$\begin{aligned}
\frac{dx}{dt} &= f(x, z, u; \theta) \\
0 &= g(x, z, u; \theta)
\end{aligned}$$

where

$$\begin{aligned}
x &= (m, n_{\mathrm{CO}_2}, U) \\
z &= (\dot{m}_\mathrm{i}, \dot{m}_\mathrm{e}, \dot{n}_{\mathrm{CO}_2,\mathrm{i}}, \dot{n}_{\mathrm{CO}_2,\mathrm{e}}, \dot{n}_{\mathrm{CO}_2,\mathrm{g}}, p, n, \dot{n}_\mathrm{i}, x_{\mathrm{CO}_2}, p_{\mathrm{CO}_2,\mathrm{i}}, \dot{n}_\mathrm{e}, \dot{m}_{\mathrm{CO}_2,\mathrm{g}}, \\
&\quad H, \hat{H}, T, \dot{H}_\mathrm{i}, \hat{H}_\mathrm{i}, \dot{H}_\mathrm{e}, \dot{Q}, \dot{Q}_\mathrm{g}) \\
u &= \left(\dot{V}_\mathrm{i}, x_{\mathrm{CO}_2,\mathrm{i}}, N, T_\mathrm{i}, \dot{Q}_\mathrm{i} \right) \\
\theta &= \left(V, R, M_\mathrm{air}, K_\mathrm{e}, p_\mathrm{a}, M_{\mathrm{CO}_2}, \dot{m}^1_{\mathrm{CO}_2,\mathrm{g}}, \hat{c}_p, \dot{Q}^1_\mathrm{g}, \eta_\mathrm{v}, T^\circ, \hat{H}^\circ \right).
\end{aligned}$$

We have indicated possible outputs x_{CO_2} and T, confer Fig. 4.17. It may also be of interest to consider $x_{\mathrm{CO}_2,\mathrm{i}}$ as output (it is also an input), as well as input temperature T_i, and heat flows $\dot{Q}$, $\dot{Q}_\mathrm{g}$ and $\dot{Q}_\mathrm{i}$. To ease the interpretation, mole fractions will be given in ppm, temperatures in °C, and heat flows in kW.

4.6.4 Modelica encoding of model

The Modelica code describes the core model of the tank, `ModFracCO2`, which is embedded in package `FracCO2` in file `FracCO2.mo`. In addition, a function `FcnHh` is included in the package to compute $\hat{H}(T)$. Model `ModFracCO2` consists of a *first section* where constants and variables are specified, and a *second section* where the model equations are specified.

```
1 package FracCO2
2     // Package for simulating CO2 fraction in non-isothermal
         room
3     // author:  Bernt Lie
4     //          University College of Southeast Norway
5     //          September 17, 2015
```

```
6       //           Revised: November 12, 2015
7       //           Revised: May 9, 2016
8       //           Revised: June 17, 2016
9       //
10      model ModFracCO2
11          // Model of CO2 fraction in non-isothermal room
12          // author:   Bernt Lie
13          //           University College of Southeast Norway
14          //           September 17, 2015
15          //           Revised: November 12, 2015
16          //           Revised: May 9, 2016
17          //           Revised: June 17, 2016
18          //
19          // Constant
20          constant Real R = 8.31e3 "Gas constant, J/(kmol.K)";
21          // Parameters
22          parameter Real V = 45*3 "Classroom volume, m3";
23          parameter Real Mair = 29 "Molar mass of air, kg/kmol
                ";
24          parameter Real Ke = 100/1.01e5 "Effluent valve
                constant, kg/(s.Pa)";
25          parameter Real pa = 1.01e5 "Atmospheric pressure, Pa
                ";
26          parameter Real MCO2 = 12+2*16 "Molar mass of CO2, kg/
                kmol";
27          parameter Real mdCO2g_1 = 1/(24*3600) "CO2 breathing
                per person, kg/(s.#persons)";
28          parameter Real Qdg_1 = 100 "Heat generated per person
                , W";
29          parameter Real etav = 0.6 "Heat recovery factor, -";
30          // Initial state parameters
31          parameter Real p0 = pa "Initial pressure in classroom
                , Pa";
32          parameter Real m0 = p0*V*Mair/(R*T0) "Initial mass in
                 classroom, kg";
33          parameter Real xCO20 = 400e-6 "Initial CO2 fraction
                in classroom, -";
34          parameter Real nCO20 = V*xCO20*p0/(R*T0) "Initial
                number of CO2 moles,kmol";
35          parameter Real T0 = 293 "Initial temperature in
                classroom, K";
36          parameter Real Hh0 = FncHh(T0) "Initial specific
                enthalpy, kJ/kg";
37          parameter Real U0 = Hh0*m0 - p0*V "Initial internal
                energy, kJ";
38          // Declaring variables
39          // -- states
```

```
40          Real m(start = m0, fixed = true) "Mass of air in
               classroom, kg";
41          Real nCO2(start = nCO20, fixed = true) "Number of
               moles CO2 in classroom, kmol";
42          Real U(start = U0, fixed = true) "Internal energy, kJ
               ";
43          // -- auxiliary variables
44          Real mdi "Influent mass flow rate of air, kg/s";
45          Real mde "Effluent mass flow rate of air, kg/s";
46          Real ndCO2i "Influent molar flow rate of CO2, kmol/s
               ";
47          Real ndCO2e "Effluent molar flow rate of CO2, kmol/s
               ";
48          Real ndCO2g "Molar rate of generation of CO2, kmol/s
               ";
49          Real p "Total pressure in classroom, Pa";
50          Real n "Total number of moles in classroom, kmol";
51          Real ndi "Influent total number of moles to classroom
               , kmol/s";
52          Real xCO2 "Mole fraction of CO2 in classroom, -";
53          Real pCO2i "Infuent partial pressure of CO2, Pa";
54          Real nde "Effluent molar flow rate, kmol/s";
55          Real mdCO2g "Mass of generation of CO2 through
               breathing, kg/s";
56          Real H "Total enthalpy, kJ";
57          Real Hh "Specific enthalpy, kJ/kg";
58          Real T "Temperature, K";
59          Real Hdi "Influent enthalpy flow, kW";
60          Real Hhi "Influent specific enthalpy, kW/kg";
61          Real Hde "Effluent enthalpy flow, kW";
62          Real Qd "Heat flow, kW";
63          Real Qdg "Heat generated by people, kW";
64          // -- input variables
65          input Real Vdi "Influent volumetric flow rate, m3/s";
66          input Real xCO2i "Influent mole fraction of CO2, -";
67          input Integer N "Number of persons in classroom, -";
68          input Real Ti "Influent temperature -- ambient
               temperature, K";
69          input Real Qdi "Heating, kW";
70          // -- output variables
71          output Real xCO2ppm "Mole fraction CO2 in classroom,
               -";
72          output Real xCO2ippm "Influent mole fraction CO2, -";
73          output Real TC "Temperature in room, C";
74          output Real TiC "Influent temperature, C";
75          output Real QdkW "Total heating, kW";
76          output Real QdgkW "Heating from persons, kW";
77          output Real QdikW "Heating, kW";
```

```
78      // Equations constituting the model
79      equation
80          // Differential equations
81          der(m) = mdi - mde;
82          der(nCO2) = ndCO2i - ndCO2e + ndCO2g;
83          der(U) = Hdi - Hde + Qd;
84          // Algebraic equations
85          m = n*Mair;
86          pa*Vdi = ndi*R*Ti;
87          mdi = ndi*Mair;
88          mde = Ke*(p-pa);
89          p*V = n*R*T;
90          //
91          pCO2i = xCO2i*pa;
92          pCO2i*Vdi = ndCO2i*R*Ti;
93          nde = mde/Mair;
94          xCO2 = nCO2/n;
95          ndCO2e = xCO2*nde;
96          ndCO2g = mdCO2g/MCO2;
97          mdCO2g = mdCO2g_1*N;
98          //
99          U = H-p*V;
100         H = m*Hh;
101         Hh = FncHh(T);
102         Hhi = FncHh(Ti);
103         Hdi = mdi*(Hhi + etav*(Hh-Hhi));
104         Hde = mde*Hh;
105         Qd = Qdi+Qdg;
106         Qdg = Qdg_1*N;
107         // outputs
108         xCO2ppm = xCO2*1e6;
109         xCO2ippm = xCO2i*1e6;
110         TC = T - 273.15;
111         TiC = Ti - 273.15;
112         QdkW = Qd/1e3;
113         QdgkW = Qdg/1e3;
114         QdikW = Qdi/1e3;
115     end ModFracCO2;
116     //
117     function FncHh "Specific enthalpy of air"
118     // Function for computing specific enthalpy of air
119     // author:   Bernt Lie
120     //           University College of Southeast Norway
121     //           November 12, 2015
122     //           Revised: May 9, 2016
123     //           Revised: June 17, 2016
124     //
125     // Function input arguments
```

```
126         input Real T "Temperature, K";
127         // Function output (response) value
128         output Real Hh "Specific enthalpy";
129     // Local (protected) quantities
130     protected
131         parameter Real Hh_o = 0 "Standard state specific
               enthalpy, J/kg";
132         parameter Real T_o = 298.15 "Standard state
               temperature, K";
133         parameter Real chp = 1e3 "Specific heat capacity, J/(
               kg.K)";
134     // Algorithm for computing specific enthalpy
135     algorithm
136       Hh := Hh_o + chp*(T - T_o);
137     end FncHh;
138   // End package
139 end FracCO2;
```

4.6.5 Nominal case

We consider the nominal case where 10 out of the 15 students leave the class room in the 15 min break after 45 min of lectures. Furthermore, we assume that the ventilation rate is increased by 50% after 2 h, and the *cooling* is turned on (200 W) after 3 h.[9] First we instantiate the model and set simulation options:

```
>>> m1 = ModelicaSystem("FracCO2.mo", "FracCO2.ModFracCO2")
>>> m1.setSimulationOptions(startTime=0,stopTime=5*3600,
   stepSize=10)
```

Next, read the classroom *volume* V, since the ventilation policy is related to the volume:

```
>>> V = m1.getParameters("V")
```

Next, we specify the nominal experiment (inputs) to the model:

```
>>> uN = [(0,15), (2700,15),(2700,5),(3600,5),(3600,15)
   ,(6300,15),(6300,5),(7200,5),(7200,15),(9900,15),(9900,5)
   ,(10800,5),(10800,15),(13500,15),(13500,5),(14400,5)
   ,(14400,15),(17100,15),(17100,5),(18000,5)]
>>> uVdi = [(0,4*V/3600),(7200,4*V/3600),(7200,6*V/3600)
   ,(18000,6*V/3600)]
>>> uQdi = [(0,0),(10800,0),(10800,-200),(18000,-200)]
>>> m1.setInputs(N=uN,Qdi=uQdi,Ti=273.15+10,Vdi=uVdi,xCO2i
   =400e-6)
```

Finally, we simulate the system and extract the solution:

```
>>> m1.simulate();
>>> tm,N,xCO2ppm,TC,Qdi,Vdi = m1.getSolutions("time","N","
   xCO2ppm","TC","Qdi","Vdi")
```

[9] Cooling = negative heating.

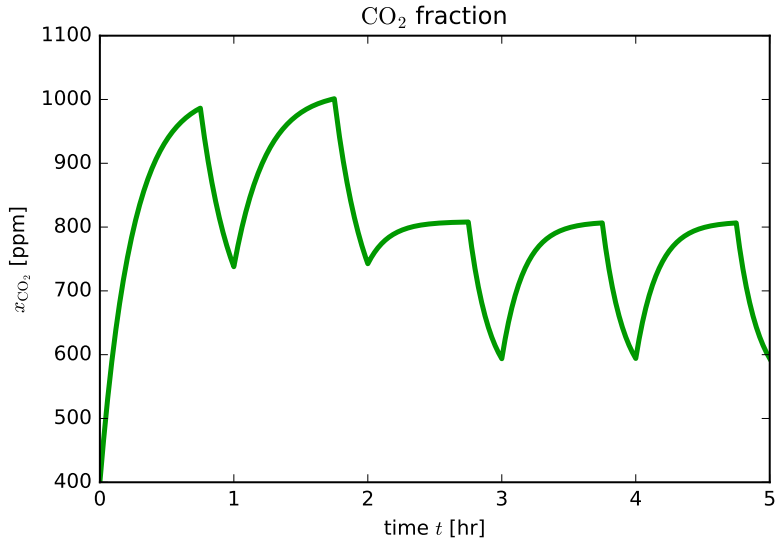


Figure 4.18: Mole fraction of CO_2 in classroom (ppm), nominal case.

We can now plot the resulting CO_2 fraction:

```
>>> plt.plot(tm/3600,xCO2ppm,linewidth=LW1,color=Cg1,label=r"
   $N$")
>>> plt.title("$\mathrm{CO}_2$ fraction")
>>> plt.xlabel(r"time $t$ [hr]")
>>> plt.ylabel(r"$x_{\mathrm{CO}_2}$ [ppm]")
>>> figfile = "xCO2FracCO2.pdf"
>>> plt.savefig(figpath+figfile)
```

see result in Fig. 4.18.

Next, we plot the resulting temperature T:

```
>>> plt.plot(tm/3600,TC,linewidth=LW1,color=Cr1,label=r"$T$ [
   C]")
>>> plt.title("Room temperature")
>>> plt.xlabel(r"time $t$ [hr]")
>>> plt.ylabel(r"$T$ [C]")
>>> figfile = "TFracCO2.pdf"
>>> plt.savefig(figpath+figfile)
```

see result in Fig. 4.19.

For completeness, it is of interest to see the number of students in the room:

```
>>> plt.plot(tm/3600,N,linewidth=LW1,color=Cb1,label=r"$N$")
>>> plt.title("#students in room")
>>> plt.xlabel(r"time $t$ [hr]")
>>> plt.ylabel(r"$N$ [#]")
>>> figfile = "NFracCO2.pdf"
>>> plt.savefig(figpath+figfile)
```

as in Fig. 4.20,

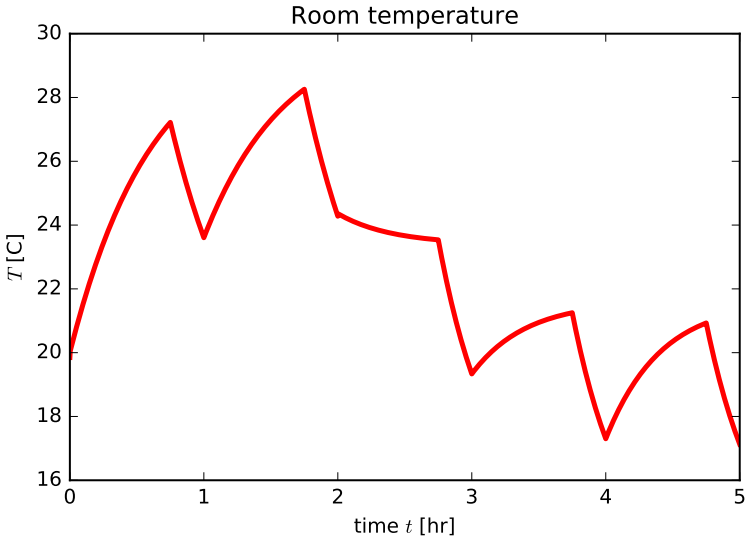


Figure 4.19: Temperature in classroom (C), nominal case.

as well as the ventilation rate:

```
>>> plt.plot(tm/3600,Vdi*1e3,linewidth=LW1,color=Cg2,label=r"
   $\dot{V}_{\mathrm{i}}$")
>>> plt.title("Ventilation rate")
>>> plt.xlabel(r"time $t$ [hr]")
>>> plt.ylabel(r"$\dot{V}_{\mathrm{i}}$ [L/s]")
>>> plt.ylim(130,250)
>>> figfile = "VdiFracCO2.pdf"
>>> plt.savefig(figpath+figfile)
```

as in Fig. 4.21,
and the "heating" (cooling):

```
plt.plot(tm/3600,Qdi,linewidth=LW1,color=Cr2,label=r"$N$")
plt.title("Room heating")
plt.xlabel(r"time $t$ [hr]")
plt.ylabel(r"$\dot{Q}_{\mathrm{i}}$ [kW]")
plt.ylim(-250,50)
figfile = "QdiFracCO2.pdf"
plt.savefig(figpath+figfile)
```

as in Fig. 4.22.

4.6.6 Uncertainty in number of students taking a break

It is of interest to see how sensitive the model solution is to variations in the number of students tanking a break. We denote the variation as in the previous section as $N_{\triangleright}$, with solutions $x_{\mathrm{CO}_2,\triangleright}$ and $T_{\triangleright}$. Next, we allow the number of students who are present in the class room to vary randomly with a uniform distribution $[0, 15]$, and repeat the simulation 20 times.

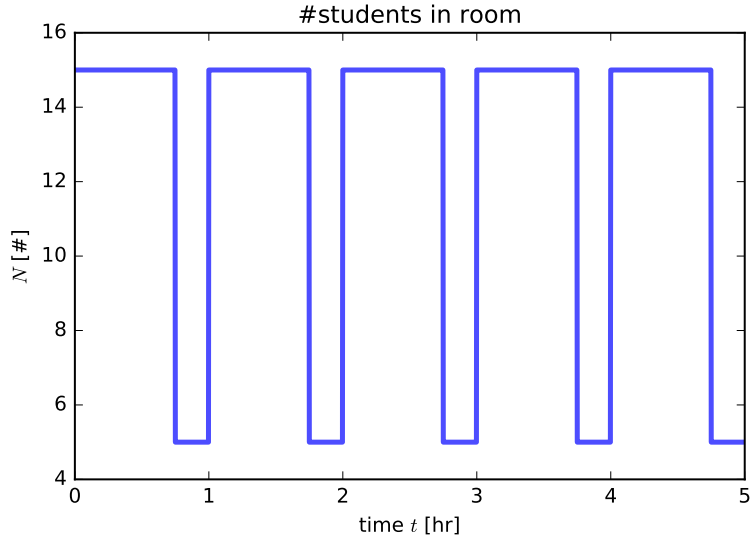


Figure 4.20: Number of students in classroom, nominal case.

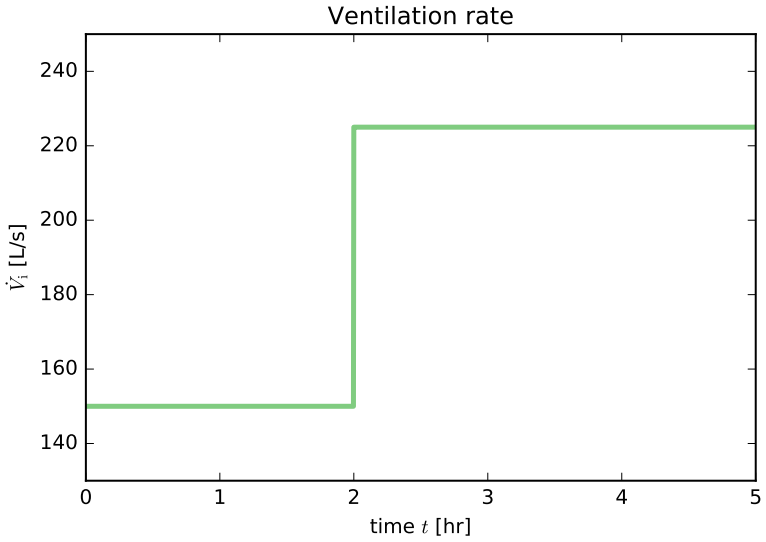


Figure 4.21: Changing ventilation room for classroom (L/s), nominal case.

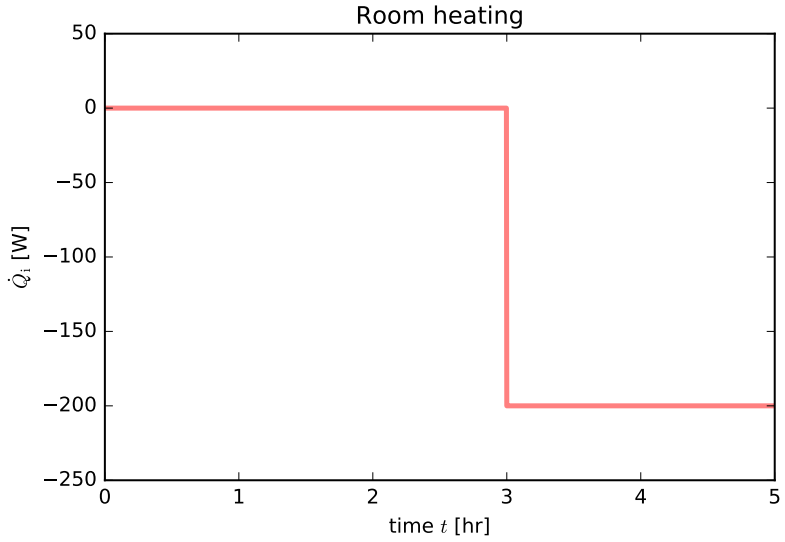


Figure 4.22: "Heating" (cooling) of classroom (kW), nominal case.

In order to produce separate figures for x_{CO_2}, T and N, we utilize Matplotlib's object-oriented interface to graphics. In particular, we create three separate *figures*, set up an *axis* in each figure, and plot to the axes. For the nominal plots, we thus have:

```
>>> f1 = plt.figure(1)
>>> ax1 = f1.add_subplot(1,1,1)
>>> f2 = plt.figure(2)
>>> ax2 = f2.add_subplot(1,1,1)
>>> f3 = plt.figure(3)
>>> ax3 = f3.add_subplot(1,1,1)
>>> ax1.plot(tm/3600,xCO2ppm,linewidth=LW1,color=Cg1,label=r"
   $x_{\mathrm{CO}_2,\triangleright}$ [ppm]")
>>> ax2.plot(tm/3600,TC,linewidth=LW1,color=Cr1,label=r"$T_{\
   \triangleright}$ [C]")
>>> ax3.plot(tm/3600,N,linewidth=LW1,color=Cb1,label=r"$N_{\
   \triangleright}$")
```

Next, we repeat the simulation 20 times for different sequences of $N\,(t)$:

```
>>> for k in range(20):
        rN = np.round(nr.rand(5)*15)
        uN = [(0,15), (2700,15),(2700,rN[0]),(3600,rN[0])
           ,(3600,15),(6300,15),(6300,rN[1]),(7200,rN[1])
           ,(7200,15),(9900,15),(9900,rN[2]),(10800,rN[2])
           ,(10800,15),(13500,15),(13500,rN[3]),(14400,rN[3])
           ,(14400,15),(17100,15),(17100,rN[4]),(18000,rN[4])
          ]
        m1.setInputs(N=uN)
        m1.simulate()
        tm1,N1,xCO2ppm1,TC1 = m1.getSolutions("time","N","
           xCO2ppm","TC")
        ax1.plot(tm1/3600,xCO2ppm1,linewidth=LW2,color=Cg2,
```

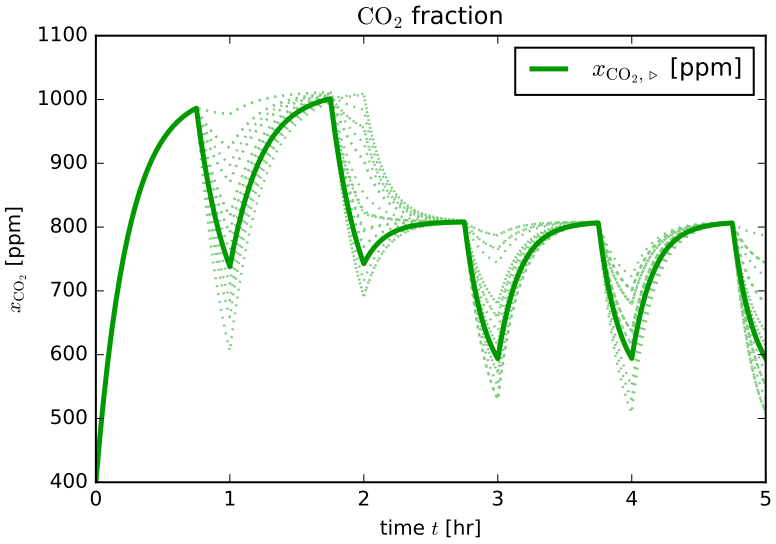


Figure 4.23: Mole fraction of CO_2 in classroom (ppm), nominal case and random variations in # students.

```
    linestyle=LS2,label="_nolabel_")
ax2.plot(tm1/3600,TC1,linewidth=LW2,color=Cr2,
    linestyle=LS2,label="_nolabel_")
ax3.plot(tm1/3600,N1,linewidth=LW2,color=Cb2,
    linestyle=LS2,label="_nolabel_"
```

Then, we set up the axes information:

```
>>> ax1.set_title("$\mathrm{CO}_2$ fraction")
>>> ax1.set_xlabel(r"time $t$ [hr]")
>>> ax1.set_ylabel(r"$x_{\mathrm{CO}_2}$ [ppm]")
>>> ax1.legend()
>>> ax2.set_title("Room temperature")
>>> ax2.set_xlabel(r"time $t$ [hr]")
>>> ax2.set_ylabel(r"$T$ [C]")
>>> ax2.legend()
>>> ax3.legend()
>>> ax3.set_title("#students in room")
>>> ax3.set_xlabel(r"time $t$ [hr]")
>>> ax3.set_ylabel(r"$N$ [#]")
>>> ax3.legend()
```

Finally, we save the plots of the three figures to files:

```
>>> figfile = "xCO2FracCO2RND.pdf"
>>> f1.savefig(figpath+figfile)
>>> figfile = "TFracCO2RND.pdf"
>>> f2.savefig(figpath+figfile)
>>> figfile = "NFracCO2RND.pdf"
>>> f3.savefig(figpath+figfile)
```

The resulting solutions of x_{CO_2}, T and N are displayed in Figs. 4.23–4.25.

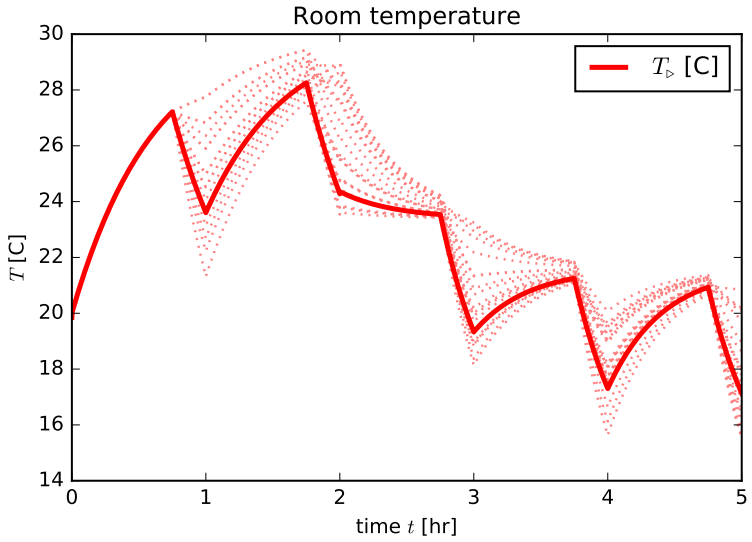


Figure 4.24: Temperature in classroom (C), nominal case and random variations in # students.

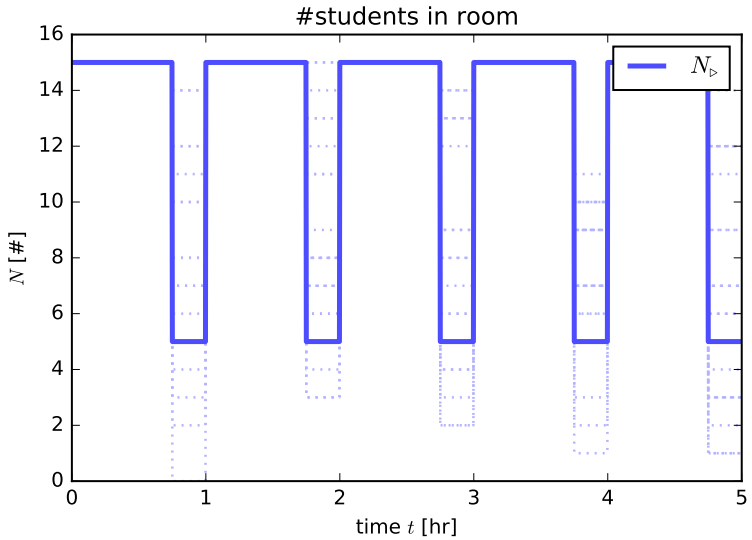


Figure 4.25: Number of students in classroom, nominal case and random variations.

For the case of N in Fig. 4.25, the plot is not really ideal: due to the nature of the variation used here (full class during lectures), it is impossible to distinguish between the 20 trajectories of N.

Chapter 5

Python API for optimization

5.1 Optimization with dynamic models

We consider the following optimization problem with cost index $\mathcal{J}$:[1]

$$\min_{u(t),\theta} \mathcal{J}, \qquad t \in [t_\mathrm{s}, t_\mathrm{f}]$$

subject to satisfying the model constraints:[2]

$$F\left(\frac{dx}{dt}, x, z, u, \theta, t\right) = 0,$$

path constraints[3]

$$\begin{aligned} c_\mathrm{i}\left(x, z, u, \theta\right) &\leq 0 \\ c_\mathrm{e}\left(x, z, u, \theta\right) &= 0, \end{aligned}$$

and point constraints[4]

$$\begin{aligned} \hat{c}_\mathrm{i}\left(x\left(t_i\right), z\left(t_i\right), u\left(t_i\right), \theta\right) &\leq 0, \qquad i \in \{1, 2, \ldots, N_\mathrm{i}\} \\ \hat{c}_\mathrm{e}\left(x\left(t_i\right), z\left(t_i\right), u\left(t_i\right), \theta\right) &= 0, \qquad i \in \{1, 2, \ldots, N_\mathrm{e}\}. \end{aligned}$$

Here, the state is $x \in \mathbb{R}^{n_x}$, the algebraic variables are $z \in \mathbb{R}^{n_z}$, the control inputs are $u \in \mathbb{R}^{n_u}$ and model parameters are $\theta \in \mathbb{R}^{n_\theta}$.

From the optimization problem, we observe that we allow for optimizing both wrt. control input variable $u\left(t\right)$, and wrt. tunable parameters θ. Tunable parameters θ are constant within the optimization horizon $[t_\mathrm{s}, t_\mathrm{f}]$. Start time t_s and final time t_f may be fixed, or may be part of the optimization problem.

Cost index $\mathcal{J}$ may depend on variables at various points in time in the optimization horizon $[t_\mathrm{s}, t_\mathrm{f}]$, as well as on parameters. In *optimal control problems*, $\mathcal{J}$ is optimized wrt. $u\left(t\right)$ only. The most common choice for cost function is that $\mathcal{J} = \mathcal{J}\left(x\left(t_\mathrm{f}\right), z\left(t_\mathrm{f}\right), u\left(t\right)\right)$; $x\left(t_\mathrm{s}\right)$, $z\left(t_\mathrm{s}\right)$ and θ are assumed to be known/given and can not change. As an example, we typically have the following cost function in optimal control:[5]

$$\mathcal{J} = \Phi\left(x\left(t_\mathrm{f}\right), z\left(t_\mathrm{f}\right)\right) + \int_{t_\mathrm{s}}^{t_\mathrm{f}} L\left(x, z, u, \theta\right) dt,$$

[1] See Section 7.3 in [10], but with modified notation.
[2] DAE formulation, which is supported by Modelica.
[3] i.e., valid for any time t
[4] i.e., valid for points t_j in time
[5] Using an extension/variation of the notation in [6].

or

$$\frac{d\Lambda}{dt} = L\left(x, z, u, \theta\right); \qquad \varphi\left(t_{\mathrm{s}}\right) = 0$$
$$\mathcal{J} = \Phi\left(x\left(t_{\mathrm{f}}\right), z\left(t_{\mathrm{f}}\right)\right) + \Lambda\left(t_{\mathrm{f}}, u\left(t\right)\right).$$

In estimation problems, on the other hand, we may have $\mathcal{J} = \sum_{j=1}^{N_J} J_j$ where $J_j = J_j\left(x\left(t_j\right), z\left(t_j\right), u\left(t_j\right), \theta\right)$ and $j \in \{1, 2, \ldots, N_J\}$.

The above general formulation allows for path constraints $c_{\mathrm{i}} \leq 0$, $c_{\mathrm{e}} = 0$ valid for every $t \in [t_{\mathrm{s}}, t_{\mathrm{f}}]$, and point constraints $\hat{c}_{\mathrm{i}} \leq 0$, $\hat{c}_{\mathrm{e}} = 0$ valid at specific points $t_j \in [t_{\mathrm{s}}, t_{\mathrm{f}}]$.

The above general formulation also allows for *static* problems where $\frac{dx}{dt} \equiv 0$ in the model, with obvious restrictions wrt. cost function $\mathcal{J}$ and the merging of path and point constraints.

5.2 The Optimica extension of Modelica

Optimica is an extension of Modelica, and was introduced in [10]; see also [11]. The Optimica extension allows for optimization problems as in the previous section by:

1. Introducing a new, specialized class `optimization` which includes the constructs used in the `model` class, and then introduces some new possibilities.[6]

2. Introducing new attributes for built-in Modelica type `Real`:[7] attribute `free` with options `true` or `false`[8], and attribute `initialGuess` with a scalar numeric value.[9] The existing Modelica attributes for specifying minimum value and maximum value (`min` and `max`, respectively) have the extended interpretation of simple bounds.

3. Introducing a new function/syntax for specifying time instance: with variable `v` denoting $v\left(t\right)$, the syntax for $v\left(t_i\right)$ is `v(ti)` where `ti` denotes t_i.

4. Introducing attributes for class `optimization`: attribute `objective` sets the cost index, attribute `startTime` sets t_{s}, and attribute `finalTime` sets t_{f}. Static optimization problems with $\frac{dx}{dt} \equiv 0$ are specified by adding attribute `static` and setting it as `static=true`.

5. Introducing a new section `constraint` used to specify path and point constraints.[10]

6. Introducing an optional annotation structure for providing transcription information on details regarding the discretization of the optimization problem. The annotation depends on the chosen discretization method and optimization tool.

In the implementation `JModelica.org`,[11] it is possible to set a *time varying* initial guess *trajectory* — not of the control input alone, but of a specified input evolution and all

[6] The new possibilities are only valid within class `optimization`.

[7] To be used for `Real` quantities of type `input` and `parameter`.

[8] To set `free = false` is not really meaningful; `false` is the default value of this optional attribute.

[9] A scalar initial guess is fine for parameters. For control inputs, a scalar initial guess implies the assumption that it is constant in the optimization horizon. A constant initial guess is often useful for control inputs, but not always.

[10] Similar to sections `equation`, `initial equation`, `algorithm`, etc. in standard Modelica class `model`.

[11] `www.JModelica.org`, a Python package for operating on Modelica models.

resulting solution trajectories from this input; an initial guess trajectory may be necessary for some nonlinear problems, and extends what is possible with attribute `initialGuess` for `Real` quantities in class `optimization`.

Possibilities with the Optimica extension to specify optimization problems with dynamic models, is illustrated below.

Example 2 *Double integrator with fixed time horizon.*

We consider the dynamic model

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = u$$

with state $\tilde{x} = (x, v)$ and initial condition $\tilde{x}(t_s) = (0, 0)$, while u is the control input. In this example, we introduce the cost index

$$\mathcal{J} = \int_0^1 u^2 dt$$

with path constraints

$$|v(t)| \leq 0.5$$
$$|u(t)| \leq 1,$$

and point constraints

$$x(t_f) = 1$$
$$v(t_f) = 0$$
$$u(t_f) = 0.$$

In our example, let $t_s = 0$ and $t_f = 2$.

The *model* can be *described*/embedded within the `optimization` class, or can be *instantiated*/linked within the `optimization` class. In this example, we embed the model of the double integrator within the `optimization` class. To this end, the following Optimica extended Modelica code should describe the optimal control problem considered here.

```
1  optimization DIntegrator1(objective=J,
2                            startTime=0, finalTime=2)
3      // cost index
4      Real J = Lambda(finalTime);
5      // model variables
6      Real x(start=0, fixed=true);
7      Real v(start=0, fixed=true, min=-0.5, max=0.5);
8      Real Lambda(start=0, fixed=true);
9      input Real u(free=true, initialGuess=0, min=-1,max=1);
10 equation
11     der(x) = v;
```

```
12      der(v) = u;
13      der(Lambda) = u*u;
14  constraint
15      x(finalTime) = 1;
16      v(finalTime) = 0;
17      u(finalTime) = 0;
18  end DIntegrator1;
```

Observe that:

1. We could have skipped introducing `J`, and simply specified the objective index as `objective=Lambda(finalTime)` in the class attribute.
2. We have used attribute `fixed` for variables `x`, `v` and `Lambda` to enforce that they start with the given initial values.
3. We have given the path constraints of `v` and `u` by using the `Real` type attributes `min` and `max`. We could alternatively have dropped these attributes, and instead specified the constraints as path constraints in the `constraint` section.

Next, we consider the case of fixed start time, but flexible end time. We also change the cost index to only put emphasis on the end time. Finally, we modify the Modelica code so that the model is described in a separate class, which is then instantiated in the optimization class.

Example 3 *Double integrator with flexible final time.*

In comparison to the previous example, we allow t_f to vary, and seek to minimize t_f. Thus, we change the cost index to

$$\mathcal{J} = t_\mathrm{f}.$$

We clearly must have $t_\mathrm{f} > t_\mathrm{s}$, say, $t_\mathrm{f} \geq 0.5$. In order to help the solver, it may be useful to restrict the final time to $t_\mathrm{f} \leq 10$. The model class and the optimization class are embedded into package `DIOptimization`. The following Modelica + Optimica code describes the problem.

```
1  package DIOptimization
2      // optimization
3      optimization DIntegrator2(objective=Lambda(finalTime),
4                                startTime=0,
5                                finalTime(free=true, initialGuess
                                    =1))
6          // cost index
7          Real Lambda(start=0, fixed=true);
8          // instantiating model
9          DIntegrator dint;
10         // control input
```

```
        input Real _u(free=true,initialGuess=0);
    equation
        der(Lambda) = 1;
        dint.u = _u;
    constraint
        finalTime >= 0,5;
        finalTime <= 10;
        dint.x(finalTime) = 1;
        dint.v(finalTime) = 0;
        _u(finalTime) = 0;
        dint.v >= -0.5;
        dint.v <= 0.5;
        _u >= -1;
        _u <= 1;
    end DIntegrator2;
    //
    // double integrator model
    model DIntegrator
        // model variables
        Real x(start=0, fixed=true);
        Real v(start=0, fixed=true);
        input Real u;
    equation
        der(x) = v;
        der(v) = u;
    end DIntegrator;
    //
end DIOptimization;
```

Here, we have chosen to introduce a new control input `_u` in the `optimization` class. This is strictly speaking not necessary: we could alternatively have dropped introducing `_u` and instead modified the instantiated `DIntegrator` model as follows:

```
DIntegrator dint(u(free=true, initialGuess=0));
```

and replaced the constraints on `_u` with constraints on `dint.u`.

These two examples illustrate the essential features of the Optimica extension, and shows how the Optimica extension can be used to encode optimization problems with dynamic models as constraints.

5.3 Optimica support in OpenModelica

The discussion here is restricted to the built-in optimization solver of OpenModelica, based on simultaneous collocation and optimization using solver `IpOpt`; this is the solver that is supported by the Python API.[12]

[12] At the moment, three solvers for optimization problems are available in the OpenModelica Shell: a built-in solver based on simultaneous collocation and optimization in `IpOpt`, an alternative solver via

Table 5.1: The Optimica extension and support for this in the Python API for OpenModelica.

Optimica Construct	**OM Python API**	**Comment**
Class `optimization`	$\vee$	OMPython adds attribute `objectiveIntegrand`
— attribute `objective`	$\vee$	— ok
— attribute `startTime`	$\vee$	— ok
— attribute `finalTime`	$\vee$	— ok
— attribute `static`	$\times$	Not stated, so probably not
`Real` attributes	$\approx$	Partially
— `free` for inputs	$\vee$	— required, or assumed by default?
— `free` for parameters	$\times$	— not supported at the moment
— `initialGuess`	$\vee$	Ok
Time instance function	$\approx$	Partially
— for arbitrary time instances	$\approx$	— only for `finalTime` at the moment?
Section `constraint`	$\approx$	Partially
— path constraints	$\vee$	— ok
— point constraints	$\times$	— not supported, even for `finalTime`
Transcription annotation	$\times$	Not stated, so probably not
— specifying discretization details	$\times$	— no information
Initial trajectory as refinement of `initialGuess`	$\times$	Not stated, so probably not

OpenModelica supports parts of the Optimica extension, Table 5.1.

5.4 Optimal operation of three tank system

We consider three tanks in sequence, each identical to the tank in Sections 2.1–2.2.

exporting the problem to `CasADi` — which offers more solution algorithms, and a sweeping parameter solver for static problems.

Chapter 6

Discussion and conclusions

This updated document on Python API for linearization aims to clarify what is needed by linearization. The linear approximation matrices A, B, C, D are the core result of the linearization, and are important for use within linear control theory. It is, of course, also of interest to be able to simulate the linear model approximation, but only in order to compare the linear approximation model with the behavior of the original nonlinear model.

Appendix A

Jupyter studies

Bibliography

[1] Bachmann, B., Ochel, L., Ruge, V., Gebremedhin, M., Fritzson, P., Nezhadali, V., Eriksson, L., and Sivertsson, M. (2012). "Parallel Multiple-Shooting and Collocation Optimization with OpenModelica". *Proceedings of the 9th International Modelica Conference*, September 3–5, 2012, Munich, Germany. DOI 10.3384/ecp12076659.

[2] Ruge, V., Braun, W., Bachmann, B., Walther, A., Kulshreshtha, K. (2014). "Efficient Implementation of Collocation Methods for Optimization using OpenModelica and ADOL-C". *Proceedings of the 10th International Modelica Conference*, March 10–12, 2014, Lund, Sweden. DOI 10.3384/ecp140961017.

[3] Bajracharya, S. (2016). *Enhanced OpenModelica Python Interface.* MSc thesis, Department of Computer and Information Science, Linköping University, Sweden.

[4] Brenan, K.E., Campbell, S.L., Petzold, L.R. (1987). *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, 2nd edition.* SIAM, Philadelphia.

[5] Fritzson, P. (2014). *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, second edition.* Wiley-IEEE Press, Piscataway, NJ. ISBN 978-1-118-85912-4.

[6] Lewis, Frank L., and Syrmos, Vassilis L. (1995). *Optimal Control, second edition.* John Wiley & Sons, Inc., New York. ISBN 0-471-03378-2.

[7] Perera, M. Anushka S., Lie, B., and Pfeiffer, C.F. (2015). "Structural Observability Analysis of Large Scale Systems Using Modelica and Python". *Modeling, Identification and Control*, **Vol 36**, No 1, pp. 53–65.

[8] Perera, M. Anushka S., Hauge, T.A., and Pfeiffer, C.F. (2015). "Parameter and State Estimation of Large-Scale Complex Systems Using Python Tools". *Modeling, Identification and Control*, **Vol 36**, No 3, pp. 189–198.

[9] Seborg, D.E., Edgar, T.F., Mellichamp, D.A., and Doyle III, F.J. (2011). *Process Dynamics and Control, third edition.* John Wiley & Sons, Inc., Hoboken, NJ. ISBN 978-0-470-12867-1.

[10] Åkesson, J. (2007). *Languages and Tools for Optimization of Large-Scale Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund University.

[11] Åkesson, J. (2008). "Optimica — An Extension of Modelica Supporting Dynamic Optimization". *Paper presented at 6th International Modelica Conference 2008, Bielefeld, Germany.*