Chapter 1. Getting started

1. Compilation of models

This tutorial covers how to compile Modelica and Optimica models into C and XML and how to load the resulting DLLs into Python.

1.1. Compilation

There are two compilers available, `ModelicaCompiler` and `OptimicaCompiler`. Any model containing Optimica code has to be compiled with the `OptimicaCompiler`. This is the case with the CSTR model which will be used in the following examples.

1.1.1. Simple compilation example

Compiling a model file can be done with just a few lines of code.

Instantiating the compiler

First the `OptimicaCompiler` must be imported and instantiated. The compiler instance can then be used multiple times on different model files.

```
# Import the compiler
from jmodelica.compiler import OptimicaCompiler

# Get an instance of OptimicaCompiler
oc = OptimicaCompiler()
```

Options

Compiler options are read from an XML file 'options.xml' which can be found in the JModelica.org installation folder. The options are loaded from the file as the compiler is instantiated. Options for a compiler instance can be modified and new options can be added interactively. There are four type categories: string, real, integer and boolean. The following example demonstrates how to get and set a string option.

```
# Get the string option default_msl_version
oc.get_string_option('default_msl_version')
>> '3.0.1'

# Set the default_msl_version to 3.0.1
oc.set_string_option('default_msl_version','3.0.1')
```

Compiling

A model can now be compiled with the compiler instance using the method `compile_model` which takes model file name and model class name as arguments.

```
# Compile the model
oc.compile_model('CSTR.mo', 'CSTR.CSTR_Opt')
```

On a successful compilation a C file, some XML files and a shared object file (DLL) have been generated. If the compilation has failed an exception will be raised.

1.1.2. Targets

The `compile_model` method takes an optional argument target which is '`model`' by default. There are two other options for this argument, 'algorithms' and 'ipopt'. It is necessary to compile with target '`ipopt`' to use the Ipopt algorithm interface for optimization.

```
# Compile the model with support for Ipopt
oc.compile_model('CSTR.mo', 'CSTR.CSTR_Opt', target='ipopt')
```

1.1.3. Compilation in more detail

Compiling with `compile_model` actually bundles a few steps required for the compilation which can be run one by one. These steps will be described briefly here, for more information on these steps, see the Architecture section in the Concept overview.

Flattening

In the first step, the model is transformed into a flat representation which can be used to generate C and XML code. Before this can be done the model must be parsed and instantiated. If there are errors in the model, for example syntax or type errors, Python exceptions will be thrown during these steps.

```
# Parse the model and get a reference to the source root
source_root = oc.parse_model('CSTR.mo')

# Generate an instance tree representation and get a reference to the model instance
model_instance = oc.instantiate_model(source_root, 'CSTR.CSTR_Opt')

# Perform flattening and get a flat representation
flat_rep = oc.flatten_model(model_instance)
```

Code generation

The next step is the code generation which produces C code containing the model equations and some XML files (slightly different for `ModelicaCompiler` and `OptimicaCompiler`) containing model meta data such as variable names and types.

```
# Generate code
oc.generate_code(flat_rep)
```

There are several files generated in this step.

Generate Shared Object file (DLL)

Finally, the DLL file is built where the C code is linked with the JModelica.org Model Interface (JMI) runtime library. The target argument must be set here if something other than the default 'model' is wanted.

```
# Compile DLL
oc.compile_dll('CSTR_CSTR_Opt', target='ipopt')
```

1.2. Loading the Shared Object file (DLL)

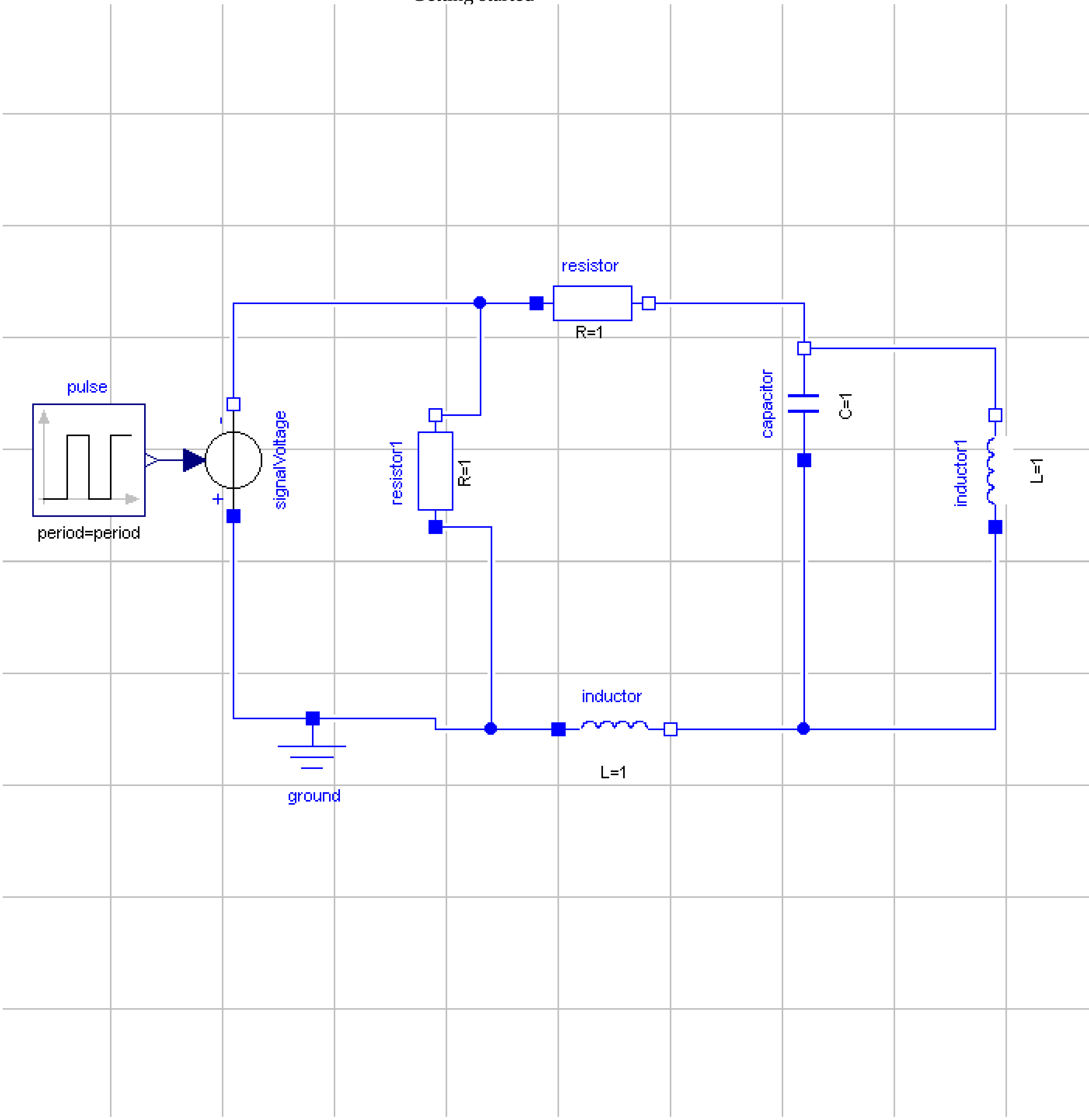
Once compilation has completed successfully a DLL file along with a few other files will have been created on the file system. The DLL file can then be loaded in Python using the class Model from which the JMI Model interface can be reached.

```
# Import Model
from jmodelica.jmi import Model
# Load dll file
model = Model('CSTR_CSTR_Opt')
```

The model object can now be used to manipulate parameters and for optimization and simulation.

2. Simulation of models

This example focus on how to use the high-level simulation functionality on a model of an electric circuit. The model is depicted in Figure (RLC.png) and consists of resistances, inductors and a capacitor. The circuit is connected to a voltage source which generates a square-wave with an amplitude of 1.0 and a frequency of 1 Hz. This model is written in Modelica code and saved in the file RLC_Circuit.mo and is depicted in below.


resistor
R=1
pulse
period=period
signalVoltage
resistor1
R=1
capacitor
C=1
inductor1
L=1
inductor
L=1
ground

To use the functionality provided by the JModelica.org platform they first have to be imported into the Python script. So we start by importing the following:

```
from jmodelica import simulate
import pylab as P
```

The method 'simulate' is the high-level simulation method and the 'pylab' package is used here for plotting.

Next, we need to provide the 'simulate' method with information about which model we would like to simulate and where it is stored. We also need information about the simulation interval. The information is then passed down in the following way,

```
(jmi_mod, res) = simulate(model='RLC_Circuit', file_name='RLC_Circuit.mo',
                          alg_args={'start_time':0.0,'final_time':30.0,'num_communication_points':0})
```

The return arguments from 'simulate' are the compiled model and the simulation results. Here 'alg_args' are the arguments for the algorithm stored in a dictionary. The 'num_communication_points' represents the number of communication points stored by the algorithm. The default is 500 points and when set to zero (0), the internal steps calculated by the algorithm are stored. If the problem requires that the default options in the specific solver needs to be changed, they should be passed down in a dictionary called 'solver_args'. Typically these options can be the tolerances. Using the default simulation package, Assimulo, information regarding which algorithms are supported and the solver arguments can be found here, http://www.jmodelica.org/assimulo . The default solver is IDA.

After a successful simulation the statistics are printed in the prompt and the results are stored in the variable 'res'. To view the result, we have to retrieve information about the variables we are interested of. This is easily done in the following way,

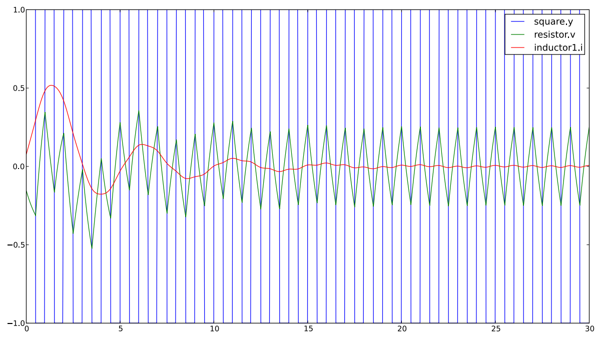
```
square_y = res.get_variable_data('y')
resistor_v = res.get_variable_data('resistor.v')
inductor1_i = res.get_variable_data('inductor1.i')
```

And then plotted with the help from pylab,

```
P.plot(square_y.t, square_y.x, resistor_v.t, resistor_v.x, inductor1_i.t, inductor1_i.x)
P.legend(('square.y','resistor.v','inductor1.i'))
P.show()
```

The simulation result is shown in the figure below:

Figure 1.2. Simulation result



3. Solving optimal control problems

A Continuously Stirred Tank Reactor (CSTR) model will be used to demonstrate how to solve an optimal control problem using the JModelica.org platform.

3.1. Python packages

The optimization examples in JModelica.org uses the Python scientific library SciPy, which is dependent on NumPy and the mathematical plotting library matplotlib for plotting. These packages will thus also be used in the example below. Most Python functions have built in documentation, which can be accessed from the Python shell by invoking the help function, for example 'help(numpy.size)'. Use this feature frequently to learn more about the packages used in this tutorial, including the jmodelica package.

3.2. The Hicks Ray Continuously Stirred Reactor (CSTR)

This example is based on the Hicks-Ray Continuously Stirred Tank Reactors (CSTR) system. The model was originally presented in [1]. The system has two states, the concentration, c, and the temperature, T. The control input to the system is the temperature, Tc, of the cooling flow in the reactor jacket. The chemical reaction in the reactor is exothermic, and also temperature dependent; high temperature results in high reaction rate. The CSTR dynamics is given by:

$$\begin{aligned}
\dot{c}(t) &= \frac{F_0(c_0-c(t))}{V} - k_0 c(t)\, e^{-\mathrm{EdivR}/T(t)} \\
\dot{T}(t) &= \frac{F_0(T_0-T(t))}{V} - \frac{dH k_0 c(t)}{\rho C_p} e^{-\mathrm{EdivR}/T(t)} + \frac{2U}{r\rho C_p}(Tc(t)-T(t))
\end{aligned}$$

This tutorial will cover the following topics:

- How to solve a DAE initialization problem. The initialization model have equations specifying that all derivatives should be identically zero, which implies that a stationary solution is obtained. Two stationary points, corresponding to different inputs, are computed. We call the stationary points A and B respectively. Point A corresponds to operating conditions where the reactor is cold and the reaction rate is low, whereas point B corresponds to a higher temperature where the reaction rate is high. For more information about the DAE initialization algorithm, see the JMI API documentation.
- An optimal control problem is solved where the objective is to transfer the state of the system from stationary point A to point B. The challenge is to ignite the reactor while avoiding uncontrolled temperature increase. It is also demonstrated how to set parameter and variable values in a model. More information about the simultaneous optimization algorithm can be found at JModelica.org API documentation.
- The optimization result is saved to file and then the important variables are plotted.

The Python commands in this tutorial may be copied and pasted directely into a Python shell, in some cases with minor modifications. Alternatively, you may copy the commands into a text file, e.g., cstr.py.

Start the tutorial by creating a working directory and copy the file $JMODELICA_HOME/Python/jmodelica/examples/files/CSTR.mo to your working directory. An on-line version of CSTR.mo is also available (depending on which browser you use, you may have to accept the site certificate by clicking through a few steps). If you choose to create Python script file, save it to the working directory.

3.2.1. Start the Python shell

Next, open a Python shell, preferably using the Pylab mode. If you are running Windows, select the menu option provided with the JModelica.org installation. If you are running Linux or Mac OS X, open a terminal and enter the command:

```
> /path/to/jmodelica_installation/Python/jm_ipython.sh -pylab
```

As your first action, go to the working directory you have created:

```
In [1]: cd '/path/to/working/directory'
```

In order to run the Python script, use the 'run' command:

```
in [2]: run cstr.py
```

3.2.2. Compile and instantiate a model object

The functions and classes used in the tutorial script need to be imported into the Python script. This is done by the following Python commands. Copy them and past them either directly into you Python shell or, preferably, into your Python script file.

```
import os.path
```

```
from jmodelica import initialize
from jmodelica import simulate
from jmodelica import optimize

import jmodelica.jmi as jmi
from jmodelica.compiler import OptimicaCompiler

import numpy as N
import matplotlib.pyplot as plt
```

Before we can do operations on the model, such as optimizing it, the model file must be compiled and the resulting DLL file loaded in Python. These steps are described in more detail in the tutorial on Compilation of models.

```
# Create a Modelica compiler instance
oc = OptimicaCompiler()

# Compile the stationary initialization model into a DLL and load it
init_model = oc.compile_model("CSTR.mo", "CSTR.CSTR_Init", target='ipopt')
```

At this point, you may open the file CSTR.mo, containing the CSTR model and the static initialization model used in this section. Study the classes CSTR.CSTR and CSTR.CSTR_Init and make sure you understand the models. Before proceeding, have a look at the interactive help for one of the functions you used:

```
In [8]: help(oc.compile_model)
```

3.2.3. Solve the DAE initialization problem

In the next step, we would like to specify the first operating point, A, by means of a constant input cooling temperature, and then solve the initialization problem assuming that all derivatives are zero.

```
# Set inputs for Stationary point A
Tc_0_A = 250
init_model.set_value('Tc',Tc_0_A)

# Solve the DAE initialization system with Ipopt
(init_model, init_result) = initialize(init_model)

# Store stationary point A
c_0_A = init_result.get_variable_data('c').x[0]
T_0_A = init_result.get_variable_data('T').x[0]

# Print some data for stationary point A
print(' *** Stationary point A ***')
print('input Tc = %f' % Tc_0_A)
print('state c = %f' % c_0_A)
print('state T = %f' % T_0_A)
```

Notice how the function set_value is used to set the value of the control input. The initialization algorithm is invoked by calling the function 'initialize', which returns the initialization result in the object 'init_result'. The 'initialize' function relies on the algorithm Ipopt for computing the solution of the initialization problem. The values of the states corresponding to grade A can then be extracted from the result object. Look carefully at the printouts in the Python shell to see a printout of the stationary values. Display the help text for the 'initialize' function and take a moment to look through it. The procedure is now repeated for operating point B:

```
# Set inputs for Stationary point B
Tc_0_B = 280
init_model.set_value('Tc',Tc_0_B)

# Solve the DAE initialization system with Ipopt
(init_model, init_result) = initialize(init_model)

# Store stationary point B
c_0_B = init_result.get_variable_data('c').x[0]
T_0_B = init_result.get_variable_data('T').x[0]

# Print some data for stationary point B
print(' *** Stationary point B ***')
print('input Tc = %f' % Tc_0_B)
print('state c = %f' % c_0_B)
print('state T = %f' % T_0_B)
```

We have now computed two stationary points for the system based on constant control inputs.

3.2.4. Solving an optimal control problem

The optimal control problem we are about to solve is given by:

$$\min_{u(t)} \int_0^{150} \left(c^{ref} - c(t)\right)^2 + \left(T^{ref} - T(t)\right)^2 + \left(T_c^{ref} - T_c(t)\right)^2 dt$$

$$\text{subject to}$$

$$230 \leq u(t) \leq 370$$

$$T(t) \leq 350$$

and is expressed in Optimica format in the class CSTR.CSTR_Opt in the CSTR.mo file above. Have a look at this class and make sure that you understand how the optimization problem is formulated and what the objective is.

Direct collocation methods often require good initial guesses in order to ensure robust convergence. Since initial guesses are needed for all discretized variables along the optimization interval, simulation provides a convenient mean to generate state and derivative profiles given an initial guess for the control input(s). It is then convenient to set up a dedicated model for computation of initial trajectories. In the model CSTR.CSTR_Init_Optimization in the CSTR.mo file, a step input is filtered through a first order filter in order to generate a smooth input for the CSTR system. The filtering is done in order not to excite unstable modes of the system, and in particular to avoid sudden ignition. Notice also that the variable names in the initialization model must match those in the optimal control model. Therefore, also the cost function is included in the initialization model.

Start by creating an input trajectory to be passed to the simulator:

```
# Create the time vector
t = N.linspace(1,150.,100)
# Create the input vector from the target input value. The
# target input value is here increased in order to get a
# better initial guess.
u = (Tc_0_B+35)*N.ones(N.size(t,0))
# Create a matrix where the first column is time and the second column represents
# the input trajectory.
u_traj = N.transpose(N.vstack((t,u)))
```

Next, compile the model and set model parameters:

```
# Compile the optimization initialization model and load the DLL
init_sim_model = oc.compile_model("CSTR.mo","CSTR.CSTR_Init_Optimization",target='ipopt')

# Set model parameters
init_sim_model.set_value('cstr.c_init',c_0_A)
init_sim_model.set_value('cstr.T_init',T_0_A)
init_sim_model.set_value('Tc_0',Tc_0_A)
init_sim_model.set_value('c_ref',c_0_B)
init_sim_model.set_value('T_ref',T_0_B)
init_sim_model.set_value('Tc_ref',u_traj.eval(0.)[0])
```

Having initialized the model parameters, we can simulate the model using the 'simulate' function.

```
(init_sim_model,res) = simulate(init_sim_model,alg_args={'start_time':0.,'final_time':150.,
                                                         'input_trajectory':u_traj})
```

The function 'simulation' first computes consistent initial conditions and then simulates the model in the interval 0 to 150 seconds with the input trajectory specified by 'u_traj'. Notice that the arguments to the simulation function is specified in a Python dictionary. Take a moment to read the interactive help for the 'simulate' function.

The simulation result is returned in the output argument 'res', from which you may now retrieve trajectories for plotting:

```
# Extract variable profiles
c_init_sim=res.get_variable_data('cstr.c')
T_init_sim=res.get_variable_data('cstr.T')
Tc_init_sim=res.get_variable_data('cstr.Tc')

# Plot the results
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_init_sim.t,c_init_sim.x)
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_init_sim.t,T_init_sim.x)
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_init_sim.t,Tc_init_sim.x)
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Look at the plots and make sure you understand the effect of the filter. Think about alternative, better ways to chose the input profile. Also, try to increase the value 35 that was added to the target input: how much can you increase this value without experiencing sudden ignition of the reactor?

Once the initial guess is generated, we compile the model containing the optimal control problem:

```
cstr = oc.compile_model("CSTR.mo", "CSTR.CSTR_Opt", target='ipopt')
```

We will now initialize the parameters of the model so that their values correspond to the optimization objective of transferring the system state from operating point A to operating point B. Accordingly, we set the parameters representing the initial values of the states to point A and the reference values in the cost function to point B:

```
cstr.set_value('Tc_ref',Tc_0_B)
cstr.set_value('c_ref',c_0_B)
cstr.set_value('T_ref',T_0_B)

cstr.set_value('cstr.c_init',c_0_A)
cstr.set_value('cstr.T_init',T_0_A)
```

In order to solve the optimization problem, we need to specify the mesh on which the optimization is performed. The simultaneous optimization algorithm is based on a collocation method that corresponds to a fixed step implicit Runge-Kutta scheme, where the mesh defines the length of each step. Also, the number of collocation points in each element, or step, needs to be provided. This number corresponds to the stage order of the Runge-Kutta scheme. The selection of mesh is analogous to the choice of step length in a one-step algorithm for solving differential equations. Accordingly, the mesh needs to be fine-grained enough to ensure sufficiently accurate approximation of the differential constraint. For an overview of simultaneous optimization algorithms, see [2].

Collocation-based optimization algorithms often require a good initial guess in order to achieve fast convergence. Also, if the problem is non-convex, initialization is even more critical. Initial guesses can be provided in Optimica by the 'initialGuess' attribute, see the CSTR.mo file for an example for this. Notice that initialization in the case of collocation-based optimization methods means initialization of all the control and state profiles as a function of time. In some cases, it is sufficient to use constant profiles. For this purpose, the 'initialGuess' attribute works well. In more difficult cases, however, it may be necessary to initialize the profiles using simulation data, where an initial guess for the input(s) has been used to generate the profiles for the dependent variables. This approach for initializing the optimization problem is used in this tutorial.

We are now ready to solve the actual optimization problem. This is done by invoking the method optimize:

```
# Initialize the mesh
n_e = 100 # Number of elements
hs = N.ones(n_e)*1./n_e # Equidistant points
n_cp = 3; # Number of collocation points in each element

(cstr,res) = optimize(cstr,alg_args={'n_e':n_e,'hs':hs,'n_cp':n_cp,'init_traj':res})
```

You should see the output of Ipopt in the Python shell as the algorithm iterates to find the optimal solution. Ipopt should terminate with a message like 'Optimal solution found' or 'Solved to an acceptable level' in order for an optimum to be found. Again, the arguments to the algorithm (number of elements, number of collocation points, element length vector and initial guess object) are given in a Python dictionary. The optimization result is returned in the output argument 'res'.

We can now retrieve the trajectories of the variables that we intend to plot:

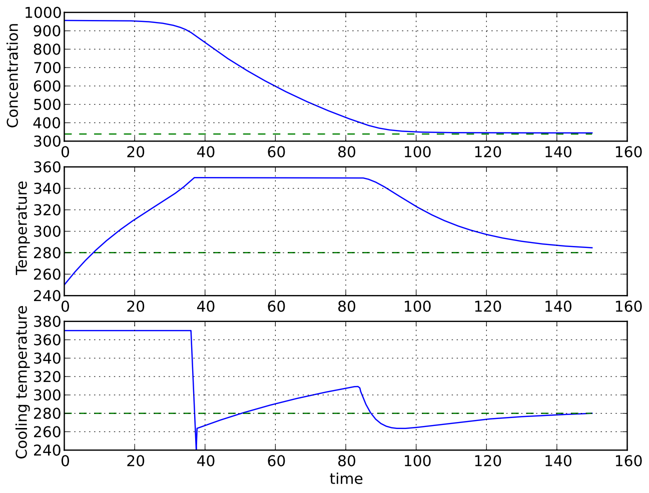
```
# Extract variable profiles
c_res=res.get_variable_data('cstr.c')
T_res=res.get_variable_data('cstr.T')
Tc_res=res.get_variable_data('cstr.Tc')

c_ref=res.get_variable_data('c_ref')
T_ref=res.get_variable_data('T_ref')
Tc_ref=res.get_variable_data('Tc_ref')
```

Finally, we plot the result using the functions available in matplotlib:

```
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_res.t,c_res.x)
plt.plot(c_ref.t,c_ref.x,'--')
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_res.t,T_res.x)
plt.plot(T_ref.t,T_ref.x,'--')
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_res.t,Tc_res.x)
plt.plot(Tc_ref.t,Tc_ref.x,'--')
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Your should now see a plot as the one below:

Figure 1.3. Optimization result



Take a minute to analyze the optimal profiles and to answer the following questions:

1. Why is the concentration high in the beginning of the interval?
2. Why is the input cooling temperature high in the beginning of the interval?

3.2.5. Verify optimal control solution

Solving optimal control problems by means of direct collocation implies that the differential equation is approximated by a discrete time counterpart. The accuracy of the solution is dependent on the method of collocation and the number of elements. In order to assess the accuracy of the discretization, we may simulate the system using a DAE solver using the optimal control profile as input. With this approach, the state profiles are computed with high accuracy and the result may then be compared with the profiles resulting from optimization. Notice that this procedure does not verify the optimality of the resulting optimal control profiles, but only the accuracy of the discretization of the dynamics.

The procedure for setting up and executing this simulation is similar to above:

```
# Simulate to verify the optimal solution
# Set up input trajectory
```

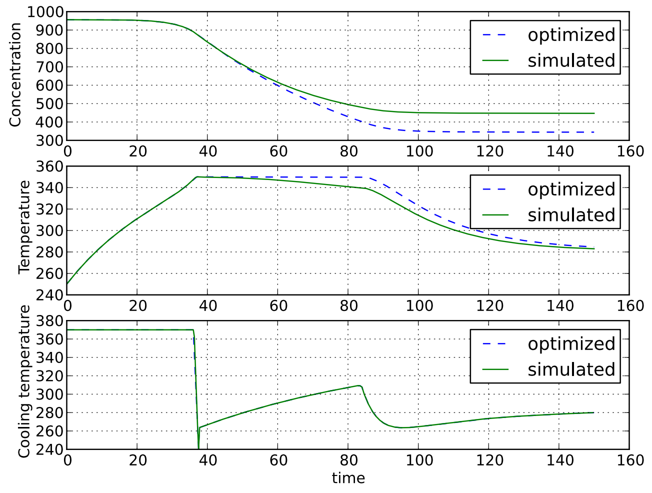
```
t = Tc_res.t
u = Tc_res.x
u_traj = N.transpose(N.vstack((t,u)))

# Comile the Modelica model first to C code and
# then to a dynamic library
sim_model = oc.compile_model("CSTR.CSTR",curr_dir+"/files/CSTR.mo",target='ipopt')

sim_model.set_value('c_init',c_0_A)
sim_model.set_value('T_init',T_0_A)
sim_model.set_value('Tc',u[0])

(sim_model,res) = simulate(sim_model,compiler='optimica',
                           alg_args={'start_time':0.,'final_time':150.,
                                     'input_trajectory':u_traj})
```

Finally, we load the simulated data and plot it to compare with the optimized trajectories:

```
# Extract variable profiles
c_sim=res.get_variable_data('c')
T_sim=res.get_variable_data('T')
Tc_sim=res.get_variable_data('Tc')

# Plot the results
plt.figure(3)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(c_res.t,c_res.x,'--')
plt.plot(c_sim.t,c_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(T_res.t,T_res.x,'--')
plt.plot(T_sim.t,T_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(Tc_res.t,Tc_res.x,'--')
plt.plot(Tc_sim.t,Tc_sim.x)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

You should now see a plot similar to:

Figure 1.4. Simulated system response



Discuss why the simulated trajectories differs from the optimized counterparts.

3.2.6. Exercises

After completing the tutorial you may continue to modify the optimization problem and study the results.

1. Remove the constraint on cstr.T. What is then the maximum temperature?
2. Play around with weights in the cost function. What happens if you penalize the control variable with a larger weight? Do a parameter sweep for the control variable weight and plot the optimal profiles in the same figure.
3. Add terminal constraints ('cstr.T(finalTime)=someParameter') for the states so that they are equal to point B at the end of the optimization interval. Now reduce the length of the optimization interval. How short can you make the interval?
4. Try varying the number of elements in the mesh and the number of collocation points in each interval. 2-10 collocation points are supported.

3.2.7. References

[1] G.A. Hicks and W.H. Ray. Approximation Methods for Optimal Control Synthesis. *Can. J. Chem. Eng.*, 40:522–529, 1971.

[2] Bieger, L., A. Cervantes, and A. Wächter (2002): "Advances in simultaneous strategies for dynamic optimization." *Chemical Engineering Science*, **57**, pp. 575-593.

4. Working with file I/O

In this tutorial you will learn how to load and save simulation/optimization results and how this can be used to initialize an optimization problem.

4.1. I/O functionality

The module io provides useful functions for exporting and loading simulation or optimization results from Dymola. The result files can be in Dymola textural or Dymola binary format. The variable data is saved together with the variable names which makes it possible to load the result files and match result data with a specific variable.

4.2. Exporting result data

In the following example, the model file CSTR.mo is assumed to have been compiled, loaded in a Model object and initialized. (See the tutorial on optimization for a complete review of solving the CSTR optimization problem.)

Initialize mesh and create an NLP object using the SimultaneousOptLagPols class. This class inherits the interface SimultaneousOpt which conveniently wrapps the IO export function.

```
# Initialize mesh
n_e = 50                          # Number of elements
hs = N.ones(n_e)*1./n_e           # Equidistant points
n_cp = 3;                         # Number of collocation points in each element

# Create an NLP object
nlp = ipopt.NLPCollocationLagrangePolynomials(cstr,n_e,hs,n_cp)
```

Create an Ipopt NLP object and solve the optimization problem.

```
# Create an Ipopt NLP object
nlp_ipopt = ipopt.CollocationOptimizer(nlp)

# Solve the optimization problem
nlp_ipopt.opt_sim_ipopt_solve()
```

After an optimization problem has been run it is a trivial operation to save the result to file.

```
# Write to file
nlp.export_result_dymola()
```

The export_result_dymola method takes an argument format which is 'txt' by default meaning that the result will be in textural format. To export in binary Matlab format, set this argument to 'mat'. The exported file will follow the name convention <DLL lib name>_result.txt|mat.

4.3. Loading result data

To load the result data saved using the export functionality the jmi.io module can be used. The result object can then be used to retrieve data for a specific variable.

```
# Load the CSTR results
res = jmodelica.io.ResultDymolaTextual('CSTR_CSTR_Opt_result.txt')

# Get variable data for T_ref
res.get_variable_data('T_ref').x
>> array([ 320., 320.])
```

There is a similar function for retrieving results from a file in Dymola binary format.

5. Setting and saving model parameters

This tutorial shows how to set model parameters and how to load and save parameter data from/to XML files.

5.1. Model parameter XML files

The model parameter meta data and values are saved in XML files which are generated during the compilation. They follow the name convention:

- <model class name>_variables.xml
- <model class name>_values.xml

The parameter meta data is saved in <model class name>_variables.xml and the parameter values in <model class name>_values.xml. A reference id is used to map a parameter value in the values file to a parameter specification.

5.2. Get and set parameter

The model parameters can be acessed with via the jmi.Model interface. It is possible to look at the whole vector or for one specific parameter. Accessing one specific parameter requires that the parameter name is known.

The following code example assumes the CSTR model has been compiled and the DLL file loaded in jmi.Model.

```
# Get independent parameter vector
cstr.get_pi()
>> array([ 1.66666667e-03, 1.00000000e+03, 1.66666667e-03,
           3.50000000e+02, 2.19000000e-01, 1.20000000e+09,
           8.75000000e+03, 9.15600000e+02, 1.00000000e+03,
           2.39000000e+02, -5.00000000e+04, 1.00000000e+02,
           1.00000000e+03, 3.50000000e+02, 5.00000000e+02,
           3.20000000e+02, 3.00000000e+02, 1.00000000e+00,
           1.00000000e+00, 1.00000000e+00])

# Get independent parameter c_ref
cstr.getparameter('c_ref')
>> 500.0

# Set independent parameter
```

```
cstr.setparameter('c_ref', 450)
```

5.3. Loading from and saving to XML

5.3.1. Loading XML values file

It is possible to load the values from an XML file as is done automatically when the jmi.Model object was first created. If, for example, there were many local changes to parameters it could be desirable to reset everything as it was from the beginning.

```
# Load values XML file
cstr.load_parameters_from_XML()
```

Default behaviour is to load the same file as was created during compilation. If another file should be used this must be passed to the method.

```
# Load other XML file
cstr.load_parameters_from_XML('new_values.xml')
```

5.3.2. Writing to XML values file

Setting a parameter value with Model.setparameter only changes the value in the vector loaded when jmi.Model was created, which means that they will not be saved. To save all changes made to parameters in a model, the values have to be written to the XML values file.

```
# Save parameters to values XML
cstr.write_parameters_to_XML()
```

If this method is called without arguments the values will be written to the XML file which was created when the model was compiled (following the name conventions mentioned above). It is also possible to save the changes in a new XML file. This is quite convienient since different parameter value settings can easily be saved and loaded in the model.

```
# Save to specific XML file
cstr.write_parameters_to_XML('test_values.xml')
```