Chapter 1. Optimization

1. Introduction

JModelica.org supports optimization of dynamic and steady state models. Many engineering problems can be cast as optimization problems, including optimal control, minimum time problems, optimal design, and model calibration. In this these different types of problems will be illustrated and it will be shown how they can be formulated and solved. The chapter starts with an introductory example in Section 2 and in ???, the details of how the optimization algorithms are invoked are explained. The following sections contain tutorial exercises that illustrates how to set up and solve different kinds of optimization problems.

When formulating optimization problems, models are expressed in the Modelica language, whereas optimization specifications are given in the Optimica extension which is described in Section 9. The tutorial exercises in this chapter assumes that the reader is familiar with the basics of Modelica and Optimica.

2. A first example

In this section, a simple optimal control problem will be solved. Consider the optimal control problem for the Van der Pol oscillator model:

```
optimization VDP_Opt (objective = cost(finalTime),
                      startTime = 0,
                      finalTime = 20)

  // The states
  Real x1(start=0,fixed=true);
  Real x2(start=1,fixed=true);

  // The control signal
  input Real u;

  Real cost(start=0,fixed=true);

equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;
  der(cost) = x1^2 + x2^2 + u^2;
constraint
   u<=0.75;
end VDP_Opt;
```

Create a new file named VDP_Opt.mop and save it in you working directory. Notice that this model contains both the dynamic system to be optimized and the optimization specification. This is possible since Optimica is an extension of Modelica and thereby supports also Modelica constructs such as variable declarations and equations. In most cases, however, Modelica models are stored separately from the Optimica specifications.

Next, create a Python script file and a write (or copy paste) the following commands:

```
# Import the function for compilation of models and the JMUModel class
from jmodelica.fmi import compile_fmux
from jmodelica.casadi_interface import CasadiModel

# Import the plotting library
import matplotlib.pyplot as plt
```

Next, we compile and load the model:

```
# Compile model
fmux_name = compile_fmux("VDP_Opt","VDP_Opt.mop")
# Load model
vdp = CasadiModel(fmux_name)
```

The function `compile_jmu` invokes the Optimica compiler and compiles the model into a DLL, which is then loaded when the `vdp` object is created. This object represents the compiled model and is used to invoke the optimization algorithm:

```
res = vdp.optimize(algorithm = 'LocalDAECollocationAlg')
```

In this case, we use the default settings for the optimization algorithm. The result object can now be used to access the optimization result:

```
# Extract variable profiles
x1=res['x1']
x2=res['x2']
u=res['u']
t=res['time']
```

The variable trajectories are returned as numpy arrays and can be used for further analysis of the optimization result or for visualization:

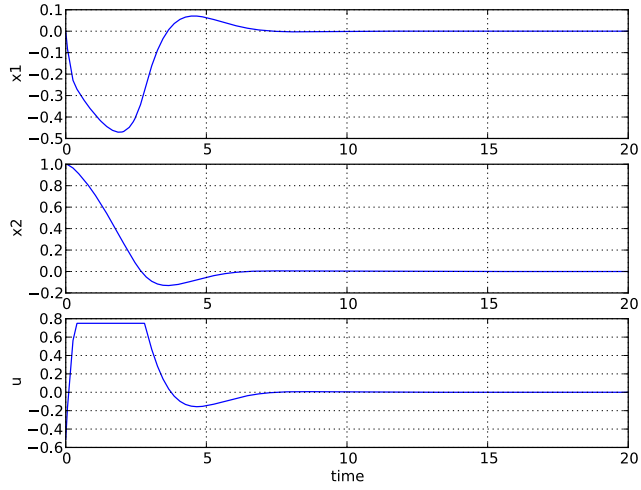
```
plt.figure(1)
plt.clf()
plt.subplot(311)
plt.plot(t,x1)
plt.grid()
plt.ylabel('x1')

plt.subplot(312)
plt.plot(t,x2)
plt.grid()
plt.ylabel('x2')

plt.subplot(313)
plt.plot(t,u)
plt.grid()
plt.ylabel('u')
```

```
plt.xlabel('time')
plt.show()
```

You should now see the optimization result as shown in Figure 1.1.



Optimal control and state profiles for the Van Der Pol optimal control problem.
Figure 1.1

3. Optimal control

This tutorial 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:

$$\dot{c}(t) = \frac{F_0(c_0 - c(t))}{V} - k_0 c(t) e^{-\mathrm{EdivR}/T(t)}$$

$$\dot{T}(t) = \frac{F_0(T_0 - T(t))}{V} - \frac{dHk_0 c(t)}{\rho C_p} e^{-\mathrm{EdivR}/T(t)} + \frac{2U}{r\rho C_p}(Tc(t) - T(t))$$

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.mop to your working directory. An on-line version of CSTR.mop 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.

3.1. 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 numpy as N
import matplotlib.pyplot as plt

from jmodelica.jmi import compile_jmu
from jmodelica.jmi import JMUModel
from jmodelica.fmi import compile_fmux
from jmodelica.casadi_interface import CasadiModel
```

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 Section 4.

```
# Compile the stationary initialization model into a JMU
jmu_name = compile_jmu("CSTR.CSTR_Init","CSTR.mop",
    compiler_options={"enable_variable_scaling":True})

# load the JMU
init_model = JMUModel(jmu_name)
```

Notice that automatic scaling of the model is enabled by setting the compiler option `enable_variable_scaling` to true. At this point, you may open the file CSTR.mop, 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(compile_jmu)
```

3.2. 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('Tc',Tc_0_A)

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

# Store stationary point A
c_0_A = init_result['c'][0]
T_0_A = init_result['T'][0]

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

Notice how the method `set` is used to set the value of the control input. The initialization algorithm is invoked by calling the `JMUModel` method `initialize`, which returns a result object from which the initialization result can be accessed. The `initialize` method 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` method 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('Tc',Tc_0_B)

# Solve the DAE initialization system with Ipopt
init_result = init_model.initialize()
# Store stationary point B
c_0_B = init_result['c'][0]
T_0_B = init_result['T'][0]

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

We have now computed two stationary points for the system based on constant control inputs. In the next section, these will be used to set up an optimal control problem.

3.3. Solving an optimal control problem

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

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

subject to

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

$$T(t) \le 350$$

and is expressed in Optimica format in the class CSTR.CSTR_Opt in the CSTR.mop 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.mop file, a step input is applied to the system in order obtain an initial guess. Notice 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.

First, compile the model and set model parameters:

```
# Compile the optimization initialization model
jmu_name = compile_jmu("CSTR.CSTR_Init_Optimization","CSTR.mop")

# Load the model
init_sim_model = JMUModel(jmu_name)

# Set model parameters
init_sim_model.set('cstr.c_init',c_0_A)
init_sim_model.set('cstr.T_init',T_0_A)
init_sim_model.set('c_ref',c_0_B)
init_sim_model.set('T_ref',T_0_B)
init_sim_model.set('Tc_ref',Tc_0_B)
```

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

```
res = init_sim_model.simulate(start_time=0.,final_time=150.)
```

The method `simulate` first computes consistent initial conditions and then simulates the model in the interval 0 to 150 seconds. Take a moment to read the interactive help for the `simulate` method.

The simulation result object is returned and to retrieve the simulation data use Python dictionary access to retrieve the variable trajectories.

```
# Extract variable profiles
c_init_sim=res['cstr.c']
T_init_sim=res['cstr.T']
Tc_init_sim=res['cstr.Tc']
t_init_sim = res['time']

# Plot the results
plt.figure(1)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(t_init_sim,c_init_sim)
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(t_init_sim,T_init_sim)
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(t_init_sim,Tc_init_sim)
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Look at the plots and try to relate the trajectories to the optimal control problem. Why is this a good initial guess?

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

```
# Compile model
fmux_name = compile_fmux("CSTR.CSTR_Opt2", "CSTR.mop")

# Load model
cstr = CasadiModel(fmux_name)
```

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 [The CasADi collocation algorithm does not yet support setting of parameters, therefore these have been preset in the model CSTR.CSTR_Opt2. Look at the model and make sure you understand how the stationary points you computed previously are used] :

```
# Set reference values
#cstr.set('Tc_ref',Tc_0_B)
#cstr.set('c_ref',c_0_B)
```

```
#cstr.set('T_ref',T_0_B)

# Set initial values
#cstr.set('cstr.c_init',c_0_A)
#cstr.set('cstr.T_init',T_0_A)
```

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.mop 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:

```
n_e = 100 # Number of elements

# Set options
opt_opts = cstr.optimize_options(algorithm="LocalDAECollocationAlg")
opt_opts['n_e'] = n_e
opt_opts['init_traj'] = res.result_data

res = cstr.optimize(algorithm="LocalDAECollocationAlg",options=opt_opts)
```

In this case, we would like to increase the number of finite elements in the mesh from 50 to 100. This is done by setting the corresponding option and provide it as an argument to the `optimize` method. 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. The optimization result object is returned and the optimization data are stored in `res`.

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

```
# Extract variable profiles
c_res=res['cstr.c']
T_res=res['cstr.T']
Tc_res=res['cstr.Tc']
time_res = res['time']

c_ref=res['c_ref']
T_ref=res['T_ref']
Tc_ref=res['Tc_ref']
```

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

```
# Plot the result
plt.figure(2)
plt.clf()
```

```
plt.hold(True)
plt.subplot(311)
plt.plot(time_res,c_res)
plt.plot([time_res[0],time_res[-1]],[c_ref,c_ref],'--')
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(time_res,T_res)
plt.plot([time_res[0],time_res[-1]],[T_ref,T_ref],'--')
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(time_res,Tc_res)
plt.plot([time_res[0],time_res[-1]],[Tc_ref,Tc_ref],'--')
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

Notice that parameters are returned as scalar values whereas variables are returned as vectors and that this must be taken into account when plotting. Your should now the plot shown in ???.

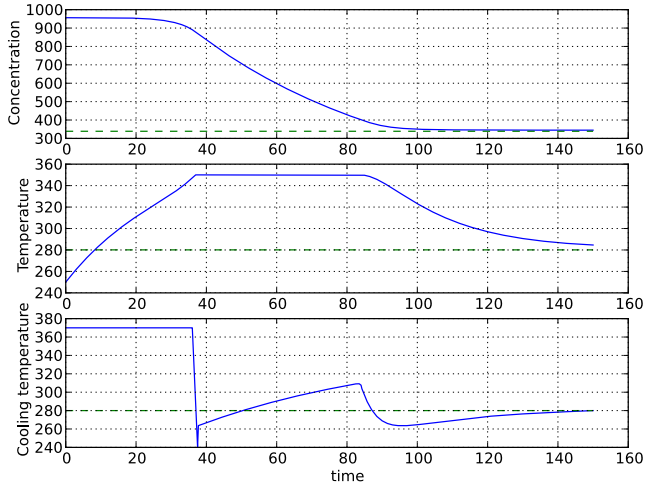


Figure 1.2 Optimal profiles for the CSTR problem.

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?

3.4. 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 the input trajectory
t = time_res
u = Tc_res
u_traj = N.transpose(N.vstack((t,u)))

# Compile the Modelica model to a JMU
jmu_name = compile_jmu("CSTR.CSTR", "CSTR.mop")

# Load model
sim_model = JMUModel(jmu_name)

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

res = sim_model.simulate(start_time=0.,final_time=150.,
    input=('Tc',u_traj))
```

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

```
# Extract variable profiles
c_sim=res['c']
T_sim=res['T']
Tc_sim=res['Tc']
time_sim = res['time']

# Plot the results
plt.figure(3)
plt.clf()
plt.hold(True)
plt.subplot(311)
plt.plot(time_res,c_res,'--')
plt.plot(time_sim,c_sim)
plt.legend(('optimized','simulated'))
```

```
plt.grid()
plt.ylabel('Concentration')

plt.subplot(312)
plt.plot(time_res,T_res,'--')
plt.plot(time_sim,T_sim)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Temperature')

plt.subplot(313)
plt.plot(time_res,Tc_res,'--')
plt.plot(time_sim,Tc_sim)
plt.legend(('optimized','simulated'))
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

You should now see the plot shown in Figure 1.3.

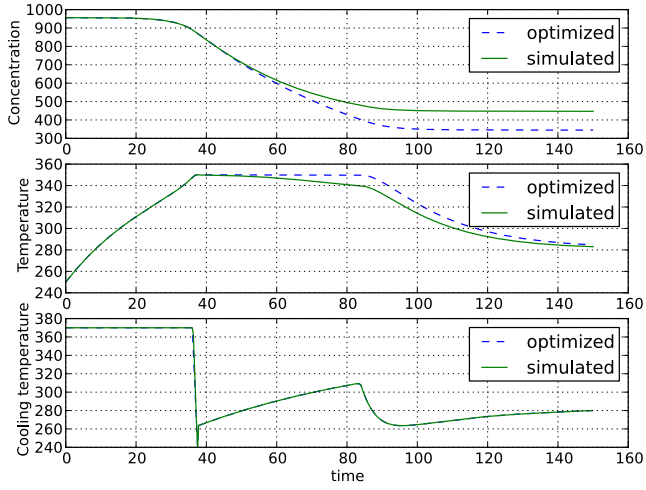


Figure 1.3 Optimal control profiles and simulated trajectories corresponding to the optimal control input.

Discuss why the simulated trajectories differs from the optimized counterparts.

3.5. 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.

3.6. 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. Scaling

Many physical models contains variables with values that differs several orders of magnitude. A typical example is thermodynamic models containing pressures, temperatures and mass flows. Such large differences in values may have a severe deteriorating effect on the performance of numerical algorithms, and may in some cases even lead to the algorithm failing. In order to relieve the user from the burden of manually scaling variables, Modelica offers the `nominal` attribute, which can be used to automatically scale a model. Consider the Modelica variable declaration:

```
Real pressure(start=101.3e3, nominal=1e5);
```

Here, the `nominal` attribute is used to specify that the variable pressure takes on values which are about 1e5. In order to use `nominal` attributes for scaling, the compiler option `enable_variable_scaling` is set to `True`, see Section 2.2.2. All variables with a nominal attribute set to true, is then scaled by dividing the variable value with its nominal value, i.e., from an algorithm point of view, all variables will take on values close to one. Notice that variables typically vary during a simulation or optimization and that it is therefore not possible to obtain perfect scaling. In order to ensure that model equations are fulfilled, each occurrence of a variable is multiplied with its nominal value in equations. For example, the equation:

```
T = f(p)
```

is replaced by the equation

```
T_scaled*T_nom = f(p_scaled*T_nom)
```

when `enable_variable_scaling` is set to `true`.

For debugging purposes, is is sometimes useful to write a simulation/optimization/initialization result to file in scaled format, in order to detect if there are some variables which requires additional scaling. The option write_scaled_result has been introduced as an option to the `initialize`, `simulate` and `optimize` methods of `JMUModel` for this purpose.