The example model library

Internship documentation

Sabine Egerer

The example model library: Internship documentation

Sabine Egerer

Publication date 2012-02-29

Table of Contents

List of Figures

Chapter 1. Introduction

1. Theme of the internship

The theme of the internship is to create and implement case studies for the solutions for simulation and optimization problems based on technical models in Modelica. The models are developed from mathematical descriptions and Matlab source code and collected in the JModelica.org example library. These case studies can be very useful for customers to get an impression how their physical problem could possibly be realized and solved using JModelica.org. Therefore one of the created case studies is supposed to be added to the *.JModelica Users Guide*

2. Abstract

In a first step the models should be set up in Modelica. To get started with Dymola, a software that uses the language Modelica, I implemented several examples from an already prepared workshop. For the examples of the "Van der Pol oscillator" (VDP) and the "Continuous Stirred Tank Reactor" (CSTR) the source code was already given and my task was to analyze and understand the problem, simulate and interpret the results, experiment with different inputs and finally write some documentation and create an icon in Dymola.

The first example I implemented was the model of a distillation column, which demonstrates a typical simulation problem. The models are taken from the Hedengren repository (http://www.hedengren.net/research/models.htm). Based on four different versions written in Matlab, I translated the problem into the Modelica language, which is applied in Dymola. The models consist of ordinary differential equations (ODE) as well as algebraic differential equations (DAE). Comparing the plots I could verify the solution of the models written in Matlab.

Thereafter the Dymola examples should be run in JModelica.org. I went through some examples to get started and learned how to compile and load the Dymola models in JModelica. I acquired a basic knowledge of the Python language, which is applied in addition to Modelica in JModelica. Due to the fact that I was developing new code I was introduced to the subversion system which offers the possibility to change and edit code by different persons at the same time. On the JModelica project planing platform the different projects are made public to provide an open base for team work.

To solve not only simulation but also optimization problems the Modelica language involves the extension Optimica. In the next step I used this approach to set up various examples of the wide range of optimization problems such as optimal control, minimal time and parameter estimation problems. Therefore the PROPT page - a Matlab Optimal Control Software for DAEs and ODEs- was very useful since it provides source code based on Matlab for a huge number of models applied to optimization. In addition the references of the models are given, which turned out to be very useful to gain background information about the models. For all models I could verify my results by comparing them with the displayed plots and solutions.

The most complex and interesting model was thereby the model of a Penicillin Plant, where the objective was to produce a maximal amount of Penicillin. The complexity arose from the fact that the model consists of two different phases and a method to provide initial guesses is integrated.

Based on simulation models I also created self-made optimization problems and experimented with different inputs, objectives and constraints based on the background information of the references in the PROPT example library.

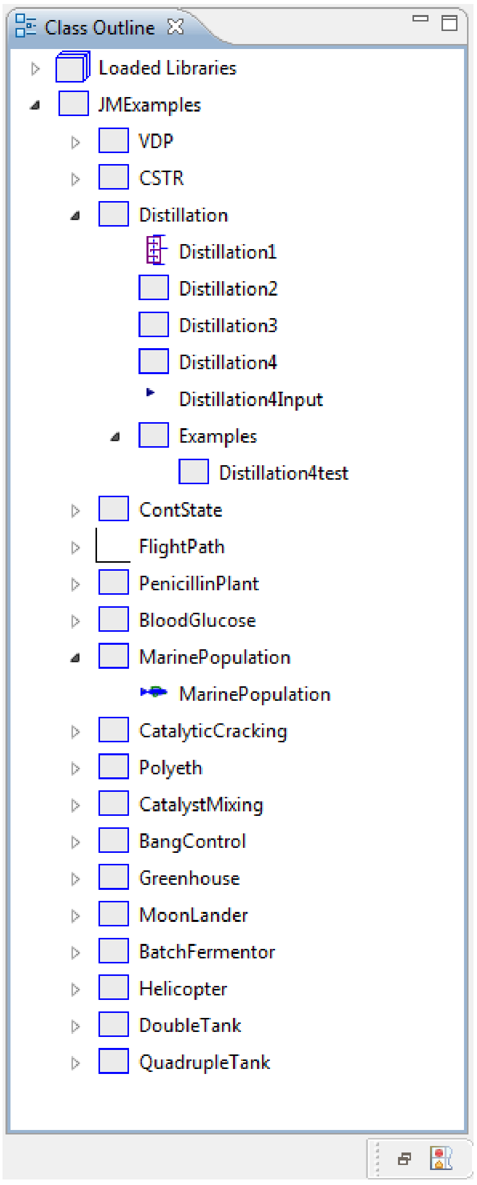
The last chapter includes a personal review of the internship and introduces my approaches related to the content.

In summary I could set up a huge facility of examples which motivate to create a wide range of various models. In this documentation only a few selected models are presented.

Chapter 2. The example model library

All the created models are collected in an example model library. Each model package contains one model or more models in the case of several modifications. For the most packages a model is added, where the input is set as a RealInput. That offers the possibility to experiment with different input sources, which are then assembled in an "Examples" folder. For each model a short summary of the background including sources can be found in the documentation layer. The creative part of the work is represented by the individual design of icons for each model.

Figure 2.1. Structure of the example model library



Class Outline

- Loaded Libraries
- JMExamples
  - VDP
  - CSTR
  - Distillation
    - Distillation1
    - Distillation2
    - Distillation3
    - Distillation4
    - Distillation4Input
    - Examples
      - Distillation4test
  - ContState
  - FlightPath
  - PenicillinPlant
  - BloodGlucose
  - MarinePopulation
    - MarinePopulation
  - CatalyticCracking
  - Polyeth
  - CatalystMixing
  - BangControl
  - Greenhouse
  - MoonLander
  - BatchFermentor
  - Helicopter
  - DoubleTank
  - QuadrupleTank

Chapter 3. Simulation

1. Simulation problems

Simulation is the process of executing experiments on a parameterized model over time. The model represents thereby the behaviour of a real physical process. The purpose of creating physical models and running simulations is to fit the parameter values in that way that the process can be improved (e.g. less energy wastage, higher production rate, less time).

The models are based on mathematical equations, ordinary differential equations (ODEs) as well as differential algebraic equations (DAEs). They contain states (variables) with related initial conditions, parameters and inputs, which can be set external.

2. Getting started

As a first example we consider a *Van der Po*l (VDP) oscillator, which is an oscillatory system with non-linear damping. The model contains two states `x1` and `x2` and is represented by a second order differential equation. For small amplitudes the damping is negative. When the amplitude is increased up to a certain limit, the damping becomes positive. The system stabilises and a limit cycle develops. In Modelica all parameters, inputs and states with initial values have to be listed. Introducing an additional state `x2` the second order equation can be transformed into two first order equations.

```
model VDP "Van der Pol model"

   // State start values
   parameter Real x1_0 = 0;
   parameter Real x2_0 = 1;

   // The states
   Real x1(start = x1_0);
   Real x2(start = x2_0);

   // Parameters
   parameter Real d=1;

   // The control signal
   Modelica.Blocks.Interfaces.RealInput u

equation
   der(x1) = d*(1 - x2^2) * x1 - x2 + u;
   der(x2) = x1;
end VDP;
```

The following plots are the results of the simulation. When x2 is plotted against x1the occurring limit cycle bespeaks an stable system.

Figure 3.1. VDP- x1 and x2 depended on time

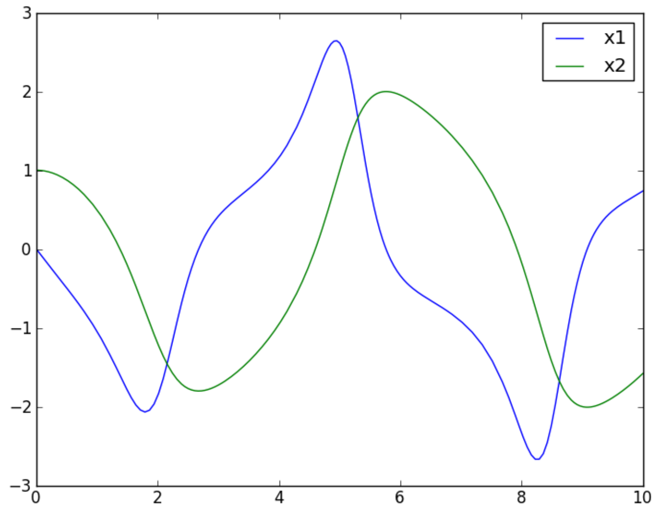
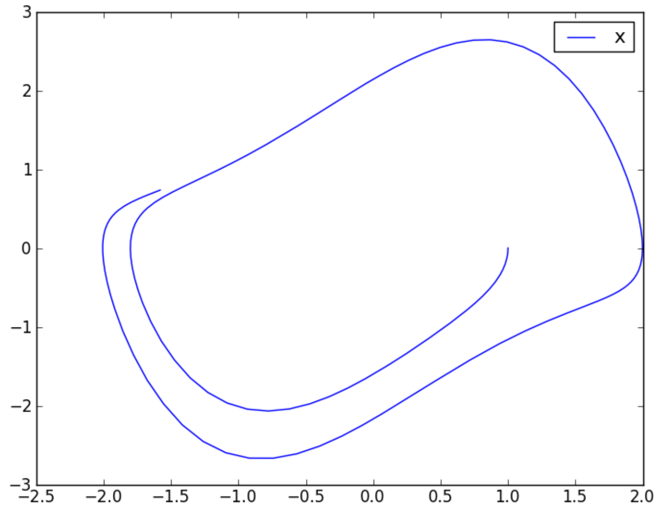


Figure 3.2. VDP- x1 depended on x2 (limit cycle)



3. Distillation column

3.1. Model description

Distillation is the physical process of separating mixtures based on the different volatilities of their components. In this binary distillation column model the two components cyclohexane and n-heptane are separated over 30 and 40 trays respectively. The model is set up in four different ways. The first two models only contain ODEs of the mole fraction x of cyclohexane at each tray. From the equilibrium assumption and mole balances the vapor mole fraction of the first component as well as both values for the second component can be calculated:

- vol = (yA/xA) / (yB/xB)
- xA + xB = 1
- yA + yB = 1

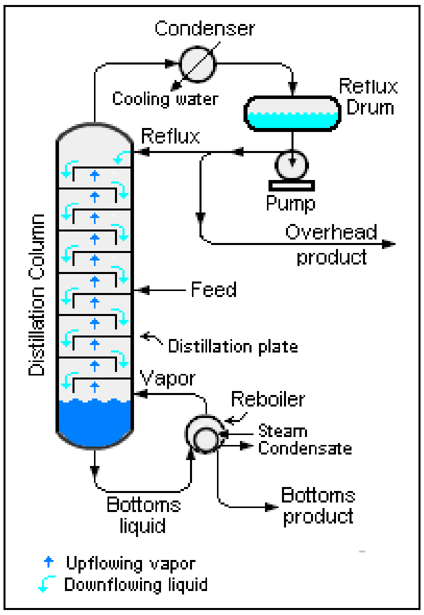
In the third version additional to the 30 states of the mole fraction DAEs for the temperature at each tray are adjoined. This version should be represented more in detail later.

The fourth version represents the most complex model. The number of trays is raised to 40 and it is described through the following equations:

- 42 ODEs for the mole fraction of cyclohexane `xA` at each tray and for the reboiler and condenser
- 42 DAEs for the temperature at each tray and for the reboiler and condenser
- 41 DAEs for the vapor molar flux between the trays, the reboiler and the condenser

The model depends on two inputs: The heat input to the reboiler provided by an electric heater and the reflux flow rate of the recycled distillate `Vdot_L1`. The reflux ratio displays the amount of condensed overhead liquid product that is returned back to the upper part of the system. The down flowing reflux ratio provides cooling and condensation of the up flowing vapors and so the efficiency of distillation column can be increased. To introduce an input Dymola provides a special component Modelica.Blocks.Interfaces.RealInput. The use of this component makes it possible to experiment with different values an even time depended functions of the input.

Figure 3.3. Picture of a distillation column (from Wikipedia: Fractional distillation)



3.2. Set up and simulate the model in Modelica

We consider the third version of the distillation column more in detail now. As before all parameter and variable declarations as well as start values, ODEs and DAEs have to be defined. To ensure the physical correctness of a model it is very helpful to work with SIunits. Thus the units of all equations are automatically adjusted and in the case of a mismatch an error occurs.

```
model Distillation3

  import SI = Modelica.SIunits;

  parameter Real rr=2.7 "reflux ratio";
  parameter SI.MolarFlowRate Feed =  24.0/3600 "Feed Flowrate";
  parameter Real x_Feed = 0.5 "Mole Fraction of Feed";
  parameter SI.MolarFlowRate D=0.5*Feed "Distillate Flowrate";
  parameter SI.MolarFlowRate L=rr*D
    "Flowrate of the Liquid in the Rectification Section";
  parameter SI.MolarFlowRate V=L+D "Vapor Flowrate in the Column";
  parameter SI.MolarFlowRate FL=Feed+L
    "Flowrate of the Liquid in the Stripping Section";
  parameter Real atray=0.25 "Total Molar Holdup in the Condenser";
  parameter Real acond=0.5 "Total Molar Holdup on each Tray";
  parameter Real areb=1.0 "Total Molar Holdup in the Reboiler";
  parameter SI.Pressure P=101000 "total pressure in column (Pa)";
  parameter Real[5,2] DIPPR= {{5.1087E1,    8.7829E1},
         {-5.2264E3,   -6.9964E3},
         {-4.2278E0,   -9.8802E0},
          {9.7554E-18,  7.2099E-6},
          {6.0000E0,    2.0000E0}} "Saturated Vapor Pressures
                                   Data from the DIPPR Database (empirical fit";
  parameter Real L12 = 1.618147731 "Activity Coefficients of Liquid Mixture";
  parameter Real L21 = 0.50253532
    "Wilson Activity Coefficient Model Parameters";

  parameter Real x_init[32]={0.97287970129754,
   0.95636038934316,
   .
   .
   .
   0.02712029870246};
  parameter SI.Temperature T_init[32]=100 *{3.54170894061095,
   3.54384309151657,
   .
   .
   .
   3.70819628750959};

  Real x[32](start=x_init) "mole fraction of A at each state, column vector";
  SI.Temperature T[32](start=T_init) "Temperature at each state";
  Real y[32] "vapor Mole Fractions of Component A";
  Real PsatA[32];
  Real PsatB[32];
  Real gammaA[32];
  Real gammaB[32];

equation
  for i in 1:32 loop
   PsatA[i] = exp(DIPPR[1,1] + DIPPR[2,1]/T[i] + DIPPR[3,1] * log(T[i]) +
      DIPPR[4,1] * (T[i]^DIPPR[5,1]));
   PsatB[i] = exp(DIPPR[1,2] + DIPPR[2,2]/T[i] + DIPPR[3,2] * log(T[i]) +
      DIPPR[4,2] * (T[i]^DIPPR[5,2]));
  end for;

  for i in 1:32 loop
     gammaA[i] = exp(-log(x[i] + L12 * (1 - x[i])) + (1 - x[i]) *
                 (L12 / (x[i] + L12 * (1 - x[i])) - (L21 / (L21 * x[i] + (1 - x[i])))));
     gammaB[i] = exp(-log((1 - x[i]) + L21 * x[i]) + x[i] *
                 (L21 / ((1 - x[i]) + L21 * x[i]) - (L12 / (L12 * (1 - x[i]) + x[i]))));
  end for "Wilson Equations";

  for i in 1:32 loop
     y[i] = x[i]*gammaA[i]*(PsatA[i] / P) "Vapor Mole Fractions of Component A";
  end for;

//ODE
   der(x[1]) = 1/acond *(V*(y[2]-x[1])) "condenser";
   der(x[2:16])  = 1/atray *(L*(x[1:15]-x[2:16]) - V*(y[2:16]-y[3:17]))
    "15 column stages";
   der(x[17]) = 1/atray * (Feed*x_Feed + L*x[16] - FL*x[17] - V*(y[17]-y[18]))
    "feed tray";
   der(x[18:31]) = 1/atray * (FL*(x[17:30]-x[18:31]) - V*(y[18:31]-y[19:32]))
    "14 column stages";
   der(x[32]) = 1/areb  * (FL*x[31] - (Feed-D)*x[32] - V*y[32]) "reboiler";

//DAE
   for i in 1:32 loop
      0= ((x[i]*gammaA[i]*PsatA[i]) + ((1-x[i])*gammaB[i]*PsatB[i])-P)/P
      "der( T[i])=0";
   end for;
end Distillation3;
```

In contrast to the extensive DAE handling in Matlab, it is sufficient to set the state derivatives 0 in Modelica.

Experimenting with the inputs the purity of the components can be raised at the bottom and the top of the distillation column by raising the heat reflux.

3.3. Compile, load and simulate Modelica models in JModelica.org

To run simulations in JModelica the Python language is used. Some imports are necessary to be able to compile and load the models.

```
# Import library for path manipulations
import os.path

# Import numerical libraries
import numpy as N
import matplotlib.pyplot as plt

# Import the JModelica.org Python packages
from pymodelica import compile_fmu
from pyfmi import FMUModel
```

The distillation3 model should be simulated for 7200 s.

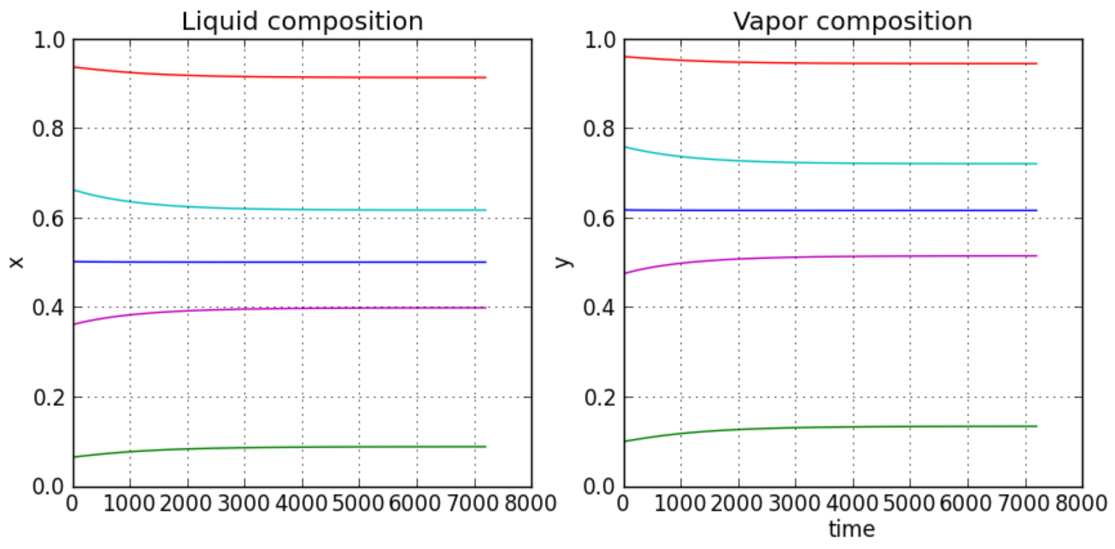
```
curr_dir = os.path.dirname(os.path.abspath(__file__));

fmu_name1 = compile_fmu("JMExamples.Distillation.Distillation1",
curr_dir+"/files/JMExamples.mo")
dist3 = FMUModel(fmu_name1)

res = dist3.simulate(final_time=7200)
```

At the end all variables are extracted, the results can be printed and plotted. More details considering this topic can be found in the JModelica User's Guide.

3.4. Simulation results and interpretation

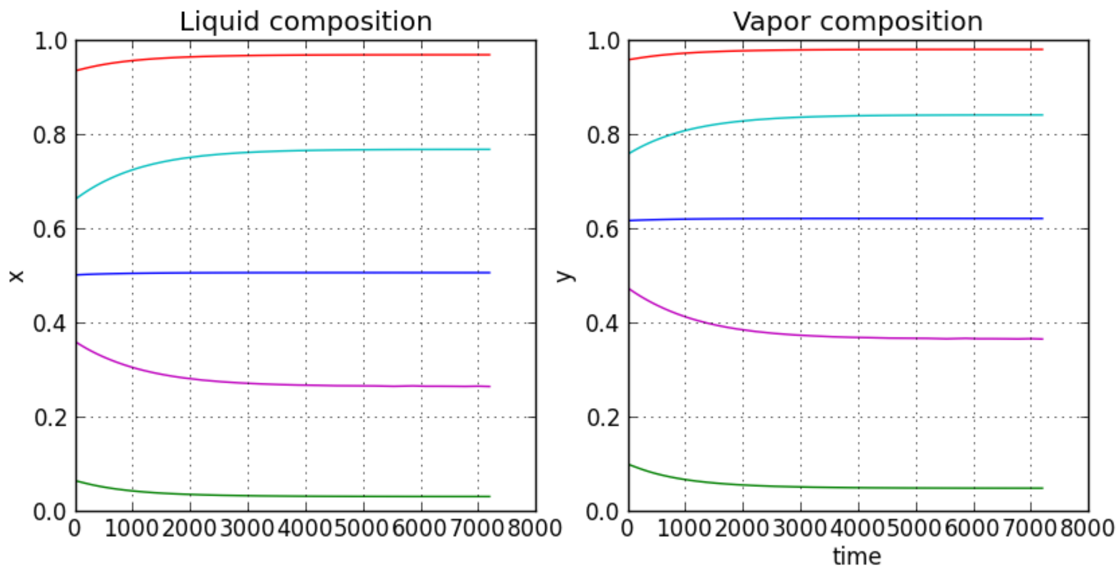
The plots of the mole fractions x and the vapor mole fractions y of component A at tray 1,8,16 and 32 are shown. The reflux flow rate was reduced from 3 to 2.7. Therefore the purity of the components at the top and the bottom decreases.

Figure 3.4. Distillation column (reflux ratio = 2.7)



Though a high purity at the trays is desirable since the purpose of distillation is the separation of the components. From theoretical considerations the conclusion can be drawn that a high reflux ratio implies a high purity, which means that x is getting closer to 1 at the top and closer to 0 at the bottom of the distillation column. This statement can be verified by setting the reflux ratio = 3.7.

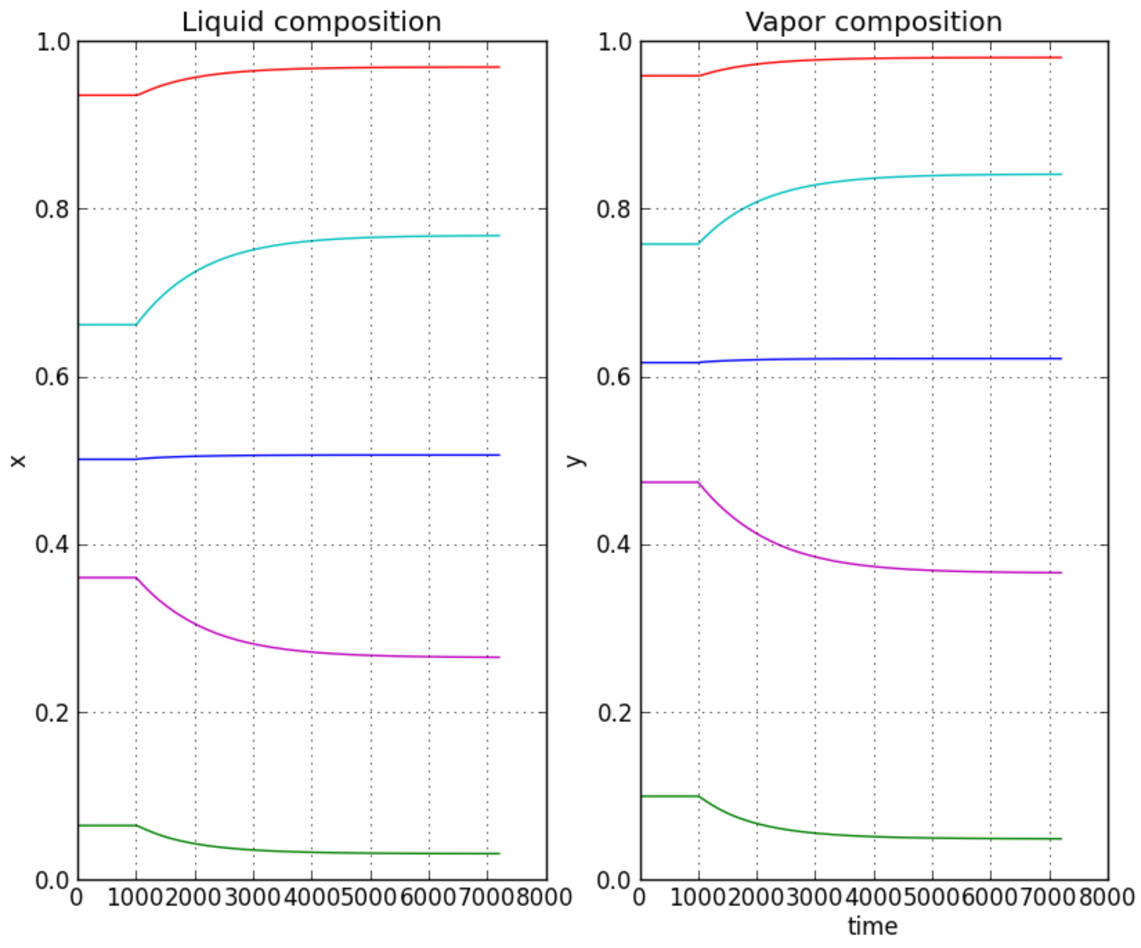
Figure 3.5. Distillation column (reflux ratio = 3.7)



It can be figured out that the purity of the components is directly depended on the reflux flow rate as an input. Only experiments certainly seem not to be convincing when searching for the optimal input to get the highest purity considering all circumstances. In the next chapter I will go one step further using optimization which leads to the aim of finding the optimal solution.

To verify the existence of a steady state condition for rr=3, a step function is created as an input, where rr=3 is kept constant for 1000s and is then increased to 3.7. In case of a steady state condition, the derivative is 0, the states stay constant.

Figure 3.6. Distillation column (reflux ratio = 3.0 for 1000s, then 3.7)



Chapter 4. Optimization

1. Optimization problems

A typical optimization problem is defined by minimizing or maximizing a given cost function f(x,u) with f: A $\rightarrow$ R, where x describes the vector of variables and u the vector of inputs. The problem is solved by systematically choosing input values. The set A is thereby defined by constraints.

A distinction is drawn between optimization of dynamic and steady state models. For steady state models algebraic equations are given for the variables. In the case of a dynamic model a system of differential equations has to be solved. JModelica.org supports both, steady state as well as dynamic models.

Optimization problems can be divided into different classes. In the following section I am going to deal with optimal control, minimum time and parameter estimation problems.

Most of the models that are introduced in this chapter are based on the Matlab models taken from the " PROPT - Matlab Optimal Control Software (DAE, ODE)" page. The challenge thereby was to modify them into the Modelica language and to solve the problem in Python. Furthermore it offers the possibility to compare and verify my results and plots.

In JModelica optimization problems are handled with the Modelica extension Optimica. In summary the Optimica extension consists of the following elements:

- a new specialized class: `optimization`
- new attributes for the built-in type Real: `free` and `initialGuess`
- a new function for accessing the value of a variable at a specified time instant
- class attributes for the specialized class `optimization: objective, startTime, finalTime and static`
- a new section: `constraint`
- inequality constraints

2. Getting started

We begin with introducing an optimal control problem to provide an insight into the theme. The example considered is the "Continuous state constraint problem".

At first the model is set up in Modelica equivalent to a simulation problem. States, start values, the control input and differential and algebraic equations are stated.

```
model ContState

     //State start values
     parameter Real x1_0 = 0;
     parameter Real x2_0= -1;

     //States
     Real x1(start = x1_0, fixed=true);
     Real x2(start = x2_0, fixed=true);
     Real p;

     //Control Signal
      Modelica.Blocks.Interfaces.RealInput u
equation
      p = 8*(time-0.5)^2-0.5-x2;
      der(x1) = x2;
      der(x2) = -x2+u;
```

```
end ContState;
```

In the next step the optimization problem is formulated using the Optimica class `optimization`. The `objective`, `startTime`, `finalTime` and the constraints are established. To import variable declarations and equations the comment "extends" is provided in Optimica.

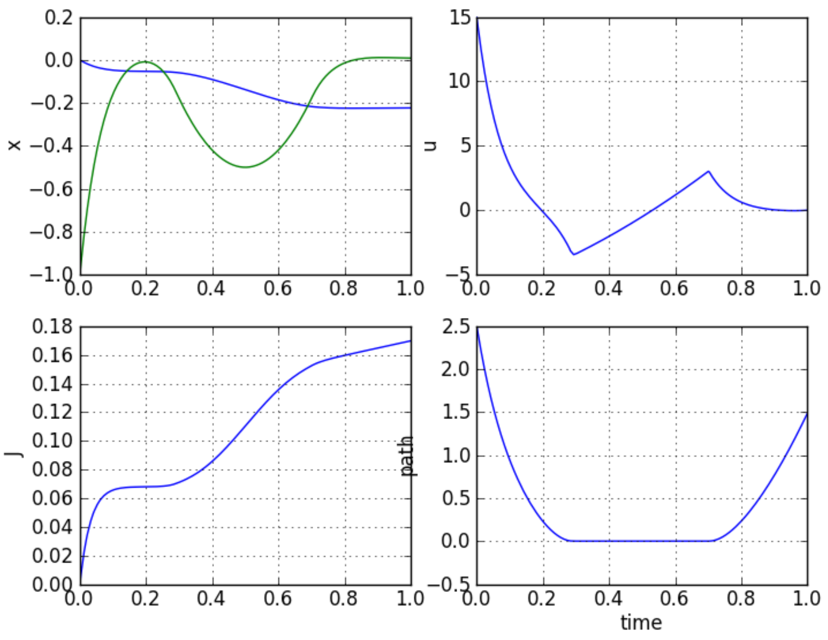
Instead it would also be possible to define an instance of the class ContState. In this case the variables and inputs are called by *instancename.variablename*.

```
optimization ContState_opt (objective = J(finalTime),
                            startTime = 0,
                            finalTime = 1)

      extends JMExamples.ContState.ContState(x1(fixed=true),x2(fixed=true),p);
  Real J(fixed=true);

  equation
     der(J) = x1^2+x2^2+0.005*u^2;
  constraint
     -10<=x1;
     x1<=10;
     -10<=x2;
     x2<=10;
     -20<=u;
     u<=20;
     8*(time-0.5).^2-0.5-x2 >= 0;
end ContState_opt;
```

Alternatively the states, control inputs and equations can be stated in the `optimization` class as well since Optimica as an extension of Modelica supports Modelica constructs such as variable declarations and equations. Though the separation of the the model into a `model` and an `optimization` class offers the possibility to experiment with different inputs in Modelica and the model can be reused.

Now the problem is solved in Python, the results are printed and plotted. This is performed similar to the procedure described in the simulation chapter. The only difference id that FMUModel is replaced by JMUModel, accordingly `compile_fmu` is replaced by `compile_jmu` and the *`optimize`* command is used instead of *`simulate`*. The states and the input are plotted:

Figure 4.1. ContState



3. Parameter estimation problems

A model description through variables, parameters and equations forms the base for parameter estimation problems. Additional a set of measured data for one or more variables over time is given. The aim is now to fit the computed state values of the model at the corresponding points in time with the measured data by adapting the parameter. This is realized by transforming the problem into an optimization problem. The objective is then the sum of the squared errors of the computed state values and the measured data.

3.1. Marine Population

In this example the given data set represents the size of the population of a marine species at each stage over time (for example infant, juvenile, adult), which depends on the stage specific growth (g) and morality rate (m). It is assumed that the species either dies or grows into the next stage, but can not skip a stage. The model equation for the population dynamics at each stage is:

$$y'[N] = g[N-1]*y[N-1] - (m[N] + g[N])*y[N] \; ; \text{ for } 1 \leq N \leq 8 \qquad (4.1)$$

Now the model is set up in Modelica and Optimica. In Modelica m and g are at first defined as parameter. In Optimica the attribute `each free` (which means free for each component of the vector) is set to handle them as tunable parameters. The first line of the matrix of measured data should be consistent with the initial value of y.

```
model MarinePopulation

  //parameter
  parameter Real m[8]={0,0,0,0,0,0,0,0};
  parameter Real g[7]={0,0,0,0,0,0,0};

  //states
  Real y[8](start={20000,17000,10000,15000,12000,9000,7000,3000}, each fixed=true);

equation
  der(y) = cat(1,{0},g).*cat(1,{0},y[1:7]) - (m+cat(1,g,{0})).*y;
end MarinePopulation;
```

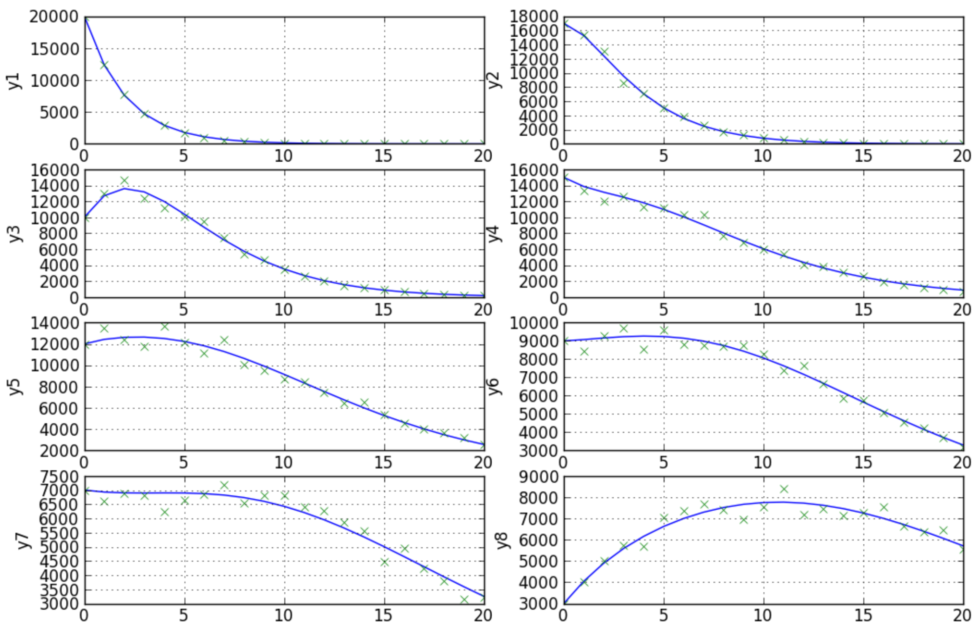
```
optimization MarinePopulation_opt (objective = sum(sum((mp.y[j](tm[i])-ym[i,j])^2
                                   for i in 1:21) for j in 1:8),
                            startTime = 0,
                            finalTime = 20)

  JMExamples.MarinePopulation.MarinePopulation
   mp(y(each fixed=true),g(each free=true), m(each free=true));

  //parameter
  parameter Real ym[21,8]=
  {{ 20000, 17000, 10000, 15000, 12000, 9000, 7000, 3000},
   { 12445, 15411, 13040, 13338, 13484, 8426, 6615, 4022},
   {  7705, 13074, 14623, 11976, 12453, 9272, 6891, 5020},
   {  4664,  8579, 12434, 12603, 11738, 9710, 6821, 5722},
   {  2977,  7053, 11219, 11340, 13665, 8534, 6242, 5695},
   {  1769,  5054, 10065, 11232, 12112, 9600, 6647, 7034},
   {   943,  3907,  9473, 10334, 11115, 8826, 6842, 7348},
   {   581,  2624,  7421, 10297, 12427, 8747, 7199, 7684},
   {   355,  1744,  5369,  7748, 10057, 8698, 6542, 7410},
   {   223,  1272,  4713,  6869,  9564, 8766, 6810, 6961},
   {   137,   821,  3451,  6050,  8671, 8291, 6827, 7525},
   {    87,   577,  2649,  5454,  8430, 7411, 6423, 8388},
   {    49,   337,  2058,  4115,  7435, 7627, 6268, 7189},
   {    32,   228,  1440,  3790,  6474, 6658, 5859, 7467},
   {    17,   168,  1178,  3087,  6524, 5880, 5562, 7144},
   {    11,    99,   919,  2596,  5360, 5762, 4480, 7256},
   {     7,    65,   647,  1873,  4556, 5058, 4944, 7538},
   {     4,    44,   509,  1571,  4009, 4527, 4233, 6649},
   {     2,    27,   345,  1227,  3677, 4229, 3805, 6378},
   {     1,    20,   231,   934,  3197, 3695, 3159, 6454},
   {     1,    12,   198,   707,  2562, 3163, 3232, 5566}};
  parameter Real tm[21]= 0:20;

 equation
end MarinePopulation_opt;
```

Before solving the optimization problem in Python we ensure that the number of computed points for each component of y is adapted to the number of measured data to be able to calculate the error at each point as well as the objective function. This can be realized by setting additional options. If no options are set, default values are used.

```
# optimize
    opts = mp.optimize_options()
    opts['n_e'] = 20
    opts['n_cp'] = 1
    res = mp.optimize(options=opts)
```

The parameter `n_e` represents the number of elements, `n_cp` the number of collocation points at each element, which is equal to the order of the used Runge Kutta method. The number of elements of the result vector of each variable results from: `n_e*n_cp+1`. In this example the accuracy of the solution does not change if I use a different value for `n_cp`.

Figure 4.2. MarinePopulation



Optimal parameter values and objective:

- g = {0.015, 0.510, 0.231, 0.264, 0.229, 0.315, 0.281}
- m = {0.015, 0.091, 0.169, 0.030, 0.030, -0.009, 0.141, 0.222}
- J = 3277.676

4. Minimum time problems

For this class of problems the objective is the class attribute `finalTime`. Hence the attribute free=true is set for `finalTime`. Additional to the common configurations for optimization problems variable values at the `finalTime` have to be given. In other words: Instead of searching for a minimal value at a fixed time we aim for a fixed value in minimal time.

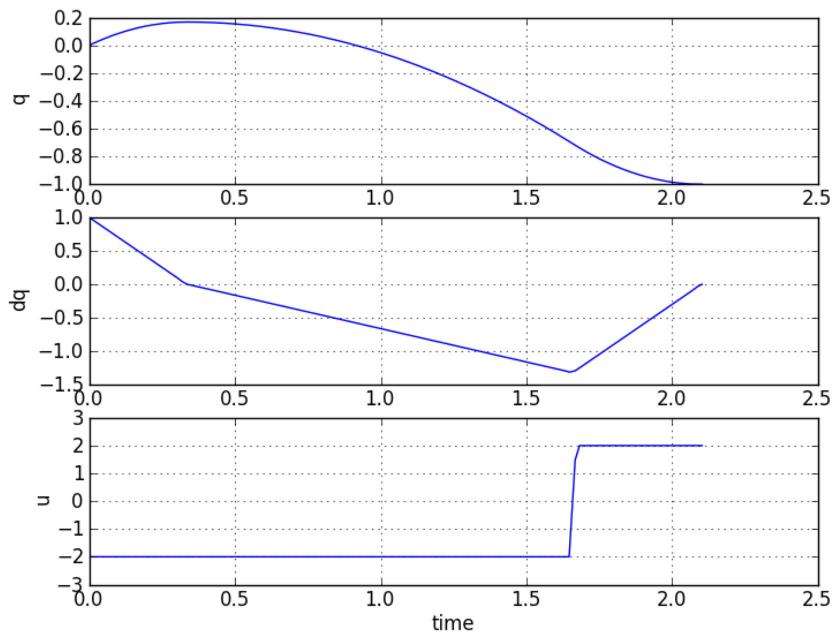
4.1. Coloumb Friction

The Coloumb Friction optimal time problem is based on a nonlinear dynamic system. The nonlinearity occurs due to the presence of a sign() term in the state equation which arises from the kinetic friction. Using the sign() function a discontinuity occurs when the sign of the variable changes. Hence Modelica generates an event which involves that some behavior is associated with th event. It is possible and at this point necessary to avoid this by using the noEvent() function.

For both states a restriction at the final time is given.

```
optimization ColoumbFriction_opt (objective = finalTime,
                            startTime = 0,
                            finalTime(free=true,min=0))

  //states
  Real q(start = 0, fixed=true);
  Real dq(start = 1, fixed=true);
  Real sign;

  //control signal
  input Real u(free=true, min=-2, max=2);

 equation
  der(q) = dq;
  sign = noEvent(if dq>0 then 1 else if dq<0 then -1 else 0);
      der(dq)= u- sign;
 constraint
  q(finalTime) = -1;
  dq(finalTime) = 0;
end ColoumbFriction_opt;
```

Figure 4.3. ColoumbFriction



5. A complex example for optimal control: Penicillin Plant

This example is the most complex and in my opinion most interesting model I dealt with. The example represents a penicillin fermentation process and the objective is to maximize the concentration of penicillin at a certain final time. The occurring isothermal reactions are thereby: S -> X and S -> P. The input is the feed flow rate of the substrate concentration S. The parameter, states and input can be find in the model set up.

```
model PenicillinPlant1

  import SI = Modelica.SIunits;

  parameter Real miu_m = 0.02;
  parameter Real Km =   0.05;
  parameter Real Ki = 5;
  parameter Real Yx =    0.5;
  parameter Real Yp =   1.2;
  parameter Real v = 0.004;
  parameter Real Sin =   200;

  //state start values
  parameter Real X1_0=1;
  parameter Real S1_0=0.5;
  parameter Real P1_0=0;
  parameter Real V1_0=150;
  parameter Real u1_0=1;

  //state start values
  Real X1(start=X1_0,fixed=true) "Cell mass concentration";
  Real S1(start=S1_0,fixed=true) "Substrate concentration";
  Real P1(start=P1_0,fixed=true) "Penicillin concentration";
  Real V1(start=V1_0,fixed=true) "Volume of medium";
  Real u1(start=u1_0,fixed=true);
  Real miu1;

  //control signal
  Modelica.Blocks.Interfaces.RealInput du1 "Feed Flowrate derivative";

equation
  miu1 = (miu_m*S1)/(Km+S1+S1^2/Ki);
  der(X1) = miu1*X1-u1/V1*X1;
  der(S1) = -miu1*X1/Yx - v*X1/Yp + u1/V1*(Sin - S1);
  der(P1) = v*X1 - u1/V1*P1;
  der(V1) = u1;
  der(u1) = du1;

end PenicillinPlant1;
```

5.1. Optimization

As already mentioned, the objective for this model is to get a maximal value for P at the finalTime. Since the objective function is minimized in Optimica, it is -P(finalTime) in this case. A constraint is put on the biomass concentration. If the biomass concentration reaches the value 3.7, the upper input constraint is set to 0.03. This guarantees X=3.7 and S=0 in that interval. Therefore a phase change is required. Since JModelica does (not yet) support multi phase models another solution has to be found. Two problems remain:

- How to create a change of constraints in a model?
- How can the optimal time for the change be computed (when does X reach the value 3.7)?

The first problem I solved by creating a second equivalent model with a different upper constraint for u. At first I assumed that after half of the time (finalTime=75) the phase change occurs.

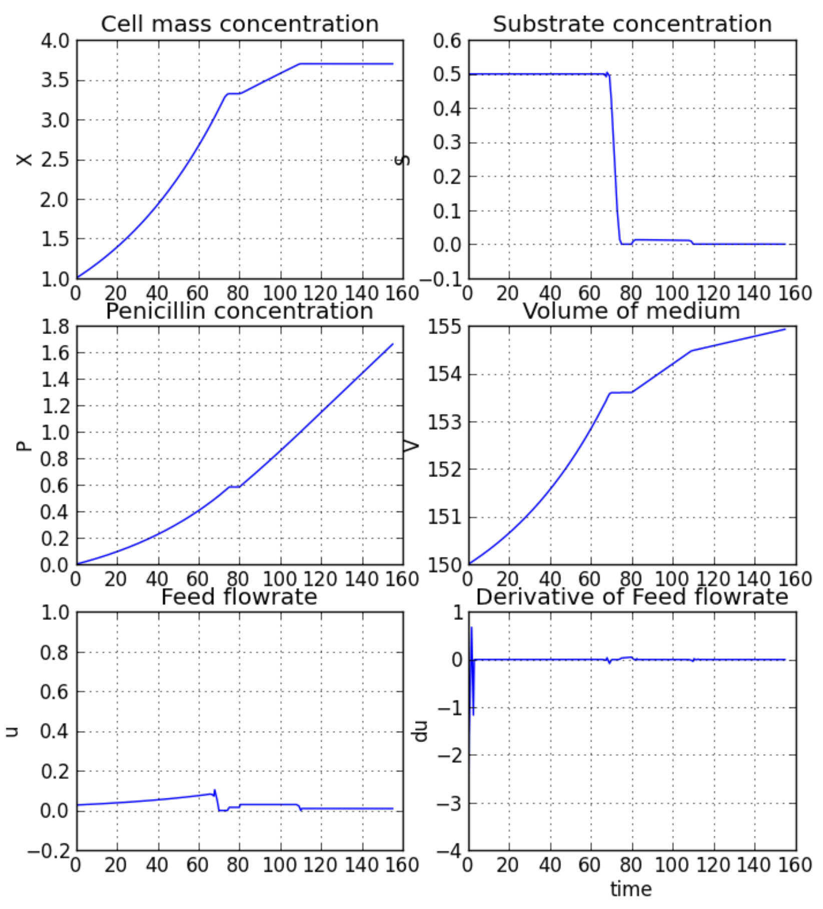
```
optimization PenicillinPlant_opt1 (objective = -P1(finalTime),
                            startTime = 0,
                            finalTime = 75)
  extends JMExamples.PenicillinPlant.PenicillinPlant1
  (X1(fixed=true),S1(fixed=true),P1(fixed=true),V1(fixed=true),miu1,u1);

 equation
 constraint
  0  <= X1;
  X1 <= 3.7;
  0  <= S1;
  S1 <= 100;
  0  <= P1;
  P1 <= 5;
  1  <= V1;
  V1 <= 300;
  0  <= u1;
  u1 <= 1;
end PenicillinPlant_opt1;
```

The values at the final time of the first phase are then set as initial values for the second phase in Python.

```
#set results of the first state as input for the second state
    X2_init = X1[len(X1)-1]
    S2_init = S1[len(S1)-1]
    P2_init = P1[len(P1)-1]
    V2_init = V1[len(V1)-1]
    u2_init = u1[len(u1)-1]
    du2_init = du1[len(du1)-1]

    pp2.set('X2_0',X2_init)
    pp2.set('S2_0',S2_init)
    pp2.set('P2_0',P2_init)
    pp2.set('V2_0',V2_init)
    pp2.set('u2_0',u2_init)
    pp2.set('du2_0',du2_init)
```

The optimization is accomplished for the second phase and the results are plotted. The penicillin production is 1.6626.

Figure 4.4. PenicillinPlant (phase change time = 75s)



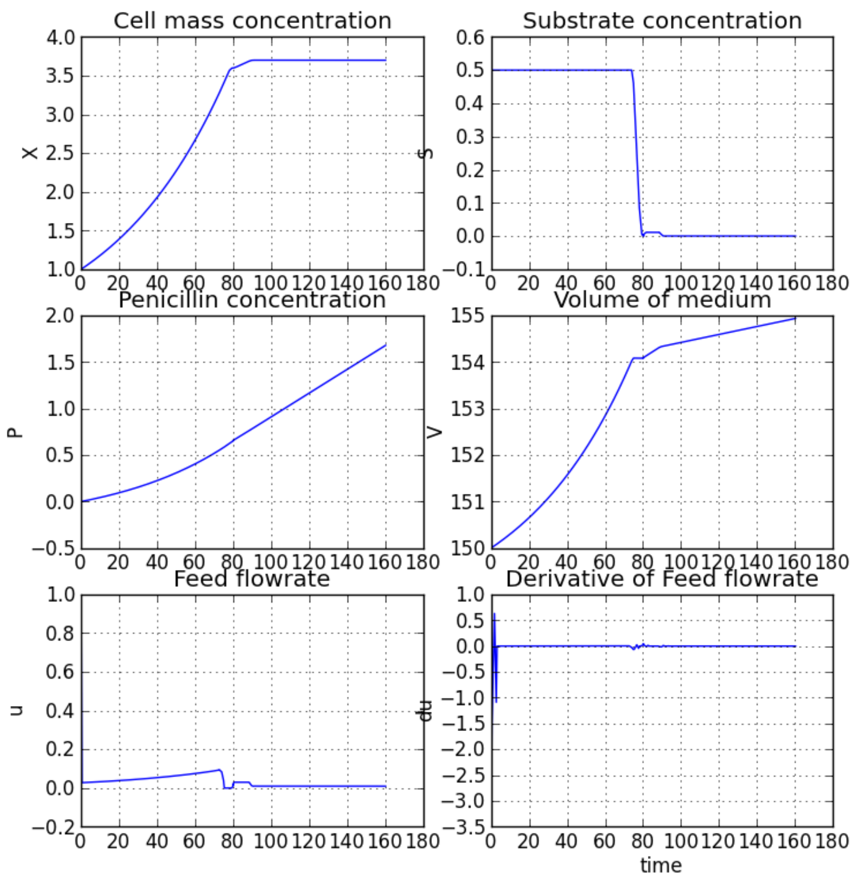
The second problem can be solved by developing a minimal time problem and setting the constraint X(finalTime) = 3.7.

```
optimization PenicillinPlant_opttime (objective = finalTime
                            startTime = 0,
                            finalTime(free=true,min=0))
   extends JMExamples.PenicillinPlant.PenicillinPlant1
   (X1(fixed=true),S1(fixed=true),P1(fixed=true),V1(fixed=true),miu1,u1);

 equation
 constraint
  0  <= X1;
  X1 <= 3.7;
  X1(finalTime) = 3.7;
  0  <= S1;
  S1 <= 0.5;
  0  <= P1;
  P1 <= 5;
  1  <= V1;
  V1 <= 300;
  0  <= u1;
  u1 <= 1;
end PenicillinPlant_opttime;
```

The result of 80.277 s seems reasonable because it is relative close to 75 s.

The optimization is performed again with a change of phases at 80.277 s.

Figure 4.5. PenicillinPlant (phase change time = 80.277s)



Indeed the amount of produced penicillin increased to 1.68 . The result can also be justified by the fact that the solver fails for change time =82 s.

6. Create an own optimization problem

Here comes the recipe how to create an optimization problem yourself. All you need is a model (set up in Modelica), which has some input value or at least parameters that can be transformed to inputs. Depending on the type of optimization problem (optimal control, minimum time, parameter estimation) there are different ways how to set up the optimization model in Optimica:

- think of an arbitrary objective function, which should be minimized depending on states and inputs
- an optimization problem could be mentioned in the references
- reference values to each variable can be created to minimize the absolute error between computed data and reference values, alternatively measurements can be executed
- to avoid an oscillating function for the input a penalty function can be set (or added) as an objective
- final values can be set to create a minimum time problem

6.1. Example: Blood Glucose

In the Blood Glucose model the plasma glucose concentration of a type-1 diabetic patient should be investigated subject to the insulin injection.

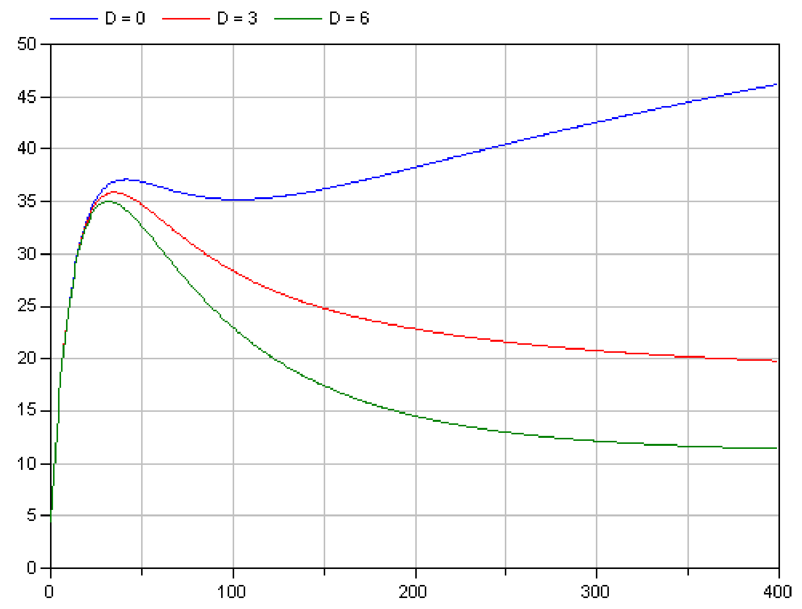
Insulin is a hormone produced by special cells and is needed to move glucose into the cells to use them later as energy supply. In type 1 diabetes the insulin production is detracted. The deficit of insulin effects an increased glucose value in the bloodstream, where it can not be transformed into energy. Therefore insulin injections are necessary.

The optimization problem is based on the physical model: the input in form of insulin injection is supposed to regulate the plasma glucose concentration. The function `dist` represents the glucose disturbance which is caused by a large meal.

```
model BloodGlucose1

  //State start values
  parameter Real G_init = 4.5;
  parameter Real X_init = 15;
  parameter Real I_init = 15;

  //States
  Real G(start = G_init, fixed=true) "Plasma Glucose Conc. (mmol/L)";
  Real X(start = X_init, fixed=true) "Plasma Insulin Conc. (mu/L)";
  Real I(start = I_init, fixed=true) "Plasma Insulin Conc. (mu/L)";
  Real dist "Meal glucose disturbance (mmol/L/min)";

  //parameter
  parameter Real G_basal = 4.5 "mmol/L";
  parameter Real X_basal = 15 "mU/L";
  parameter Real I_basal = 15 "mU/L";
  parameter Real P1 = 0.028735;
  parameter Real P2 = 0.028344;
  parameter Real P3 = 5.035e-5;
  parameter Real V1 = 12;
  parameter Real n = 5/54;

  //Control Signal
  Modelica.Blocks.Interfaces.RealInput D "Insulin Infusion rate"

equation
  der(G) = -P1 * (G - G_basal) - (X - X_basal) * G + dist;
  der(X) = -P2 * (X - X_basal) + P3 * (I - I_basal);
  der(I) = -n * I + D / V1;
  dist = 3*exp(-0.05*time);

end BloodGlucose1;
```

At first some constant values are set for the insulin infusion rate.

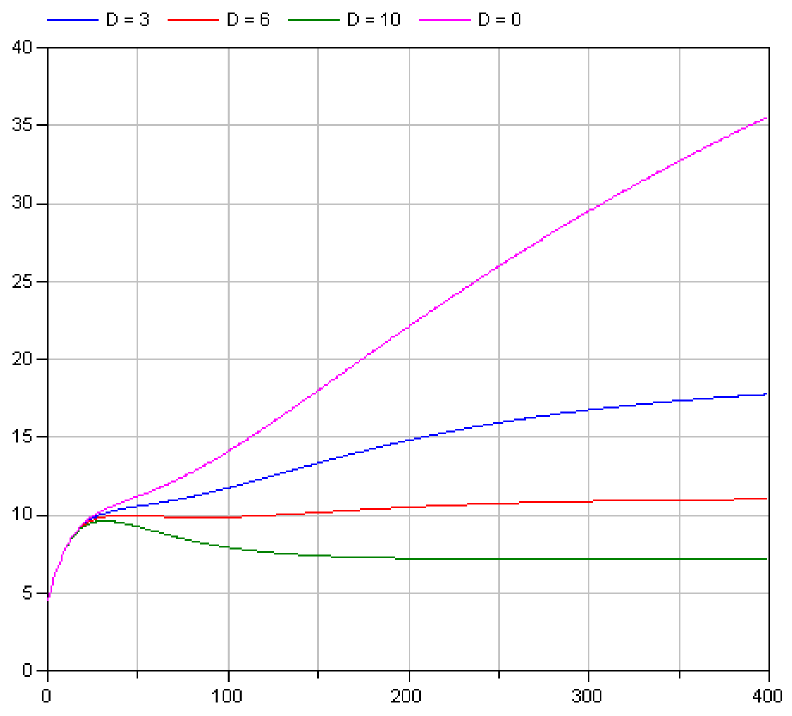
Figure 4.6. BloodGlucose (constant values of D)



The pictures evince a dependence of the Plasma Glucose concentration from the insulin infusion rate. At first a peak in the blood glucose concentration function is observable which is attributed to the large meal (disturbance function). The more insulin was injected, the lower drops the glucose concentration in the blood decreases after the peak. We can conclude that it is possible to control the glucose concentration through an insulin injection in the model and hence an optimal control problem can be stated.

But still the peak value for the blood glucose level seems much too high compared to reference values. From different sources I ascertained that the normal blood glucose level is between 3 and 6 mmol/L, where the value gets up to 8 mmol/L after eating but should definitely be smaller than 10 mmol/L. Changing the disturbance function the peak value of G can be controlled. Therefore dist = 0.5*exp(-0.05*time) seems to be reasonable since that keeps the value for G smaller than 10 mmol/L. Again I set different constant values for D.

Figure 4.7. BloodGlucose (constant values of D; scaled function dist)

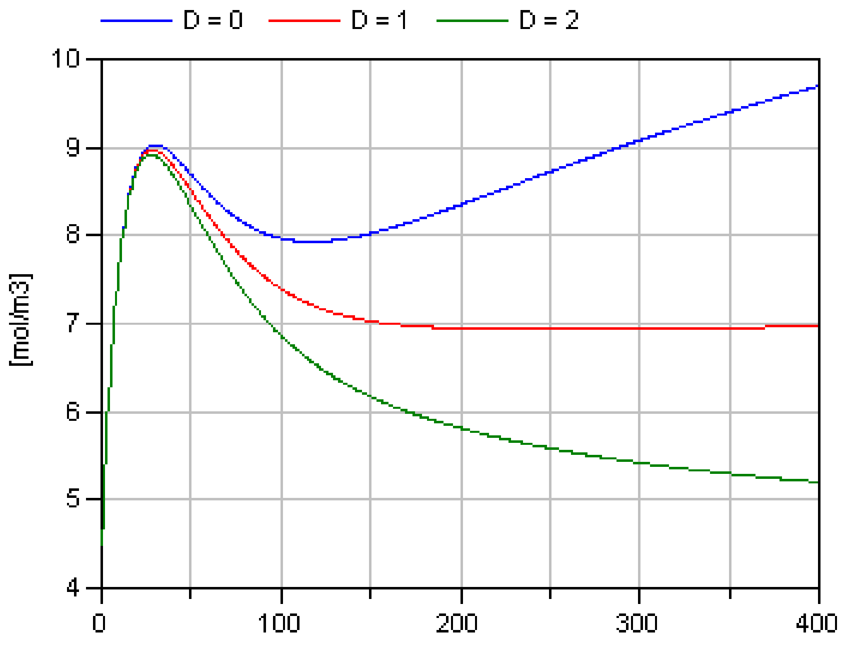


If the dist function is changed, the value still increases unreasonably after the peak which seems to be unphysical. Therefore I decided to scale G instead using the factor 5 to keep the peak value smaller than 10mmol/L. The ODE for G is replaced by:

```
der(G) = (-P1 * (G*5 - G_init) - (X - X_init) * G*5 + dist)/5;
```

The suitable values for D should be smaller now:

Figure 4.8. BloodGlucose (constant values of D; scaled function G)

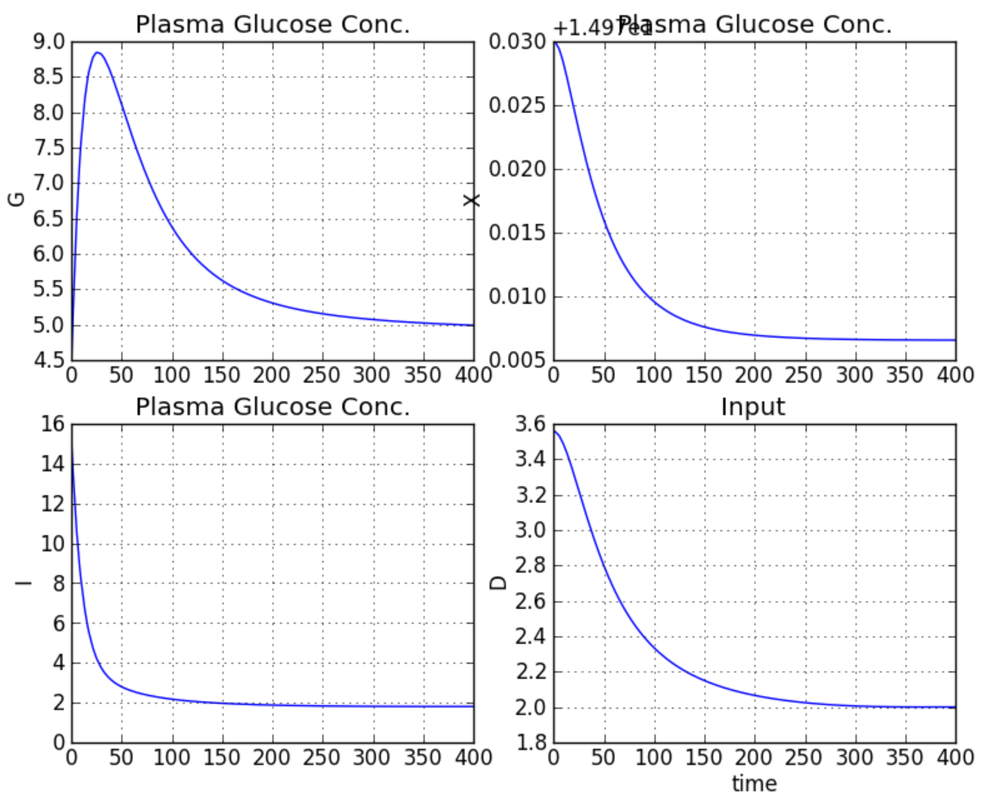


Now the peak value for G is smaller than 10 mmol/L and the function is not rising too much after the peak. I will continue based on this model.

After these investigations the optimization problem can be created. The aim is to predict the relationship between the insulin injection and the blood glucose level. With a sufficient accurate mathematical model of a patient, the correct insulin injection rate could be prescribed. I chose as an objective to minimize the error between the computed value and a reference value for G and D. The difficult part is now to choose reference values. As a reference value for the blood glucose level I set 5 mmol/L, which represents a normal blood glucose level. From the plot above I assume that D = 2 mmol/L presents a suitable approximation for D.

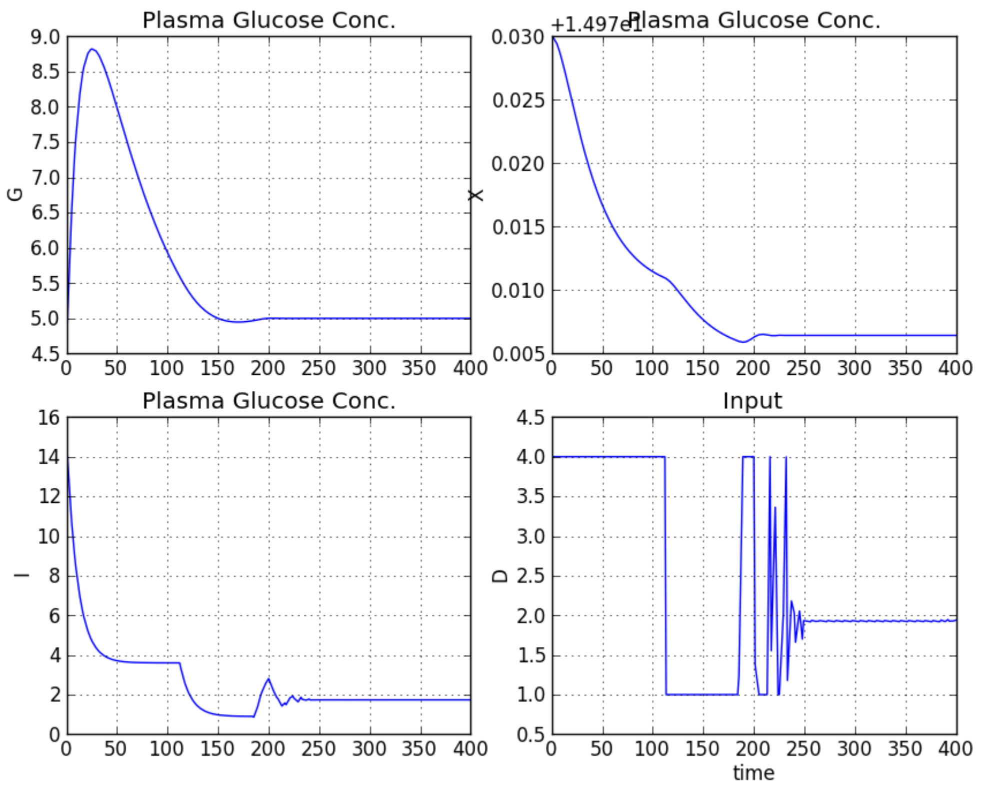
```
optimization BloodGlucose_opt(objective=(cost(finalTime)),
                      startTime=0,
                      finalTime=400)

  input Real u=bc.D;
  JMExamples.BloodGlucose.BloodGlucose1 bc(G,X,I,D);

  Real cost(start=0,fixed=true);
  parameter Real G_ref = 5;
  parameter Real D_ref = 2;

 equation
  der(cost) = (G_ref-bc.G)^2 + (D_ref-u)^2;

 constraint
end BloodGlucose_opt;
```

Figure 4.9. BloodGlucose (reference values for D and G)



The result looks smooth but does not really offer a big effort compared to the simulation. Furthermore if a reference value for D is given the example still seems to be a bit constructed. So I want to go further and only set the error between the computed and the desired value for G as an objective. D should at least be constrained by reasonable values, for example 1 and 4 mmol/L.

Figure 4.10. BloodGlucose (reference value for G, constraints for D)

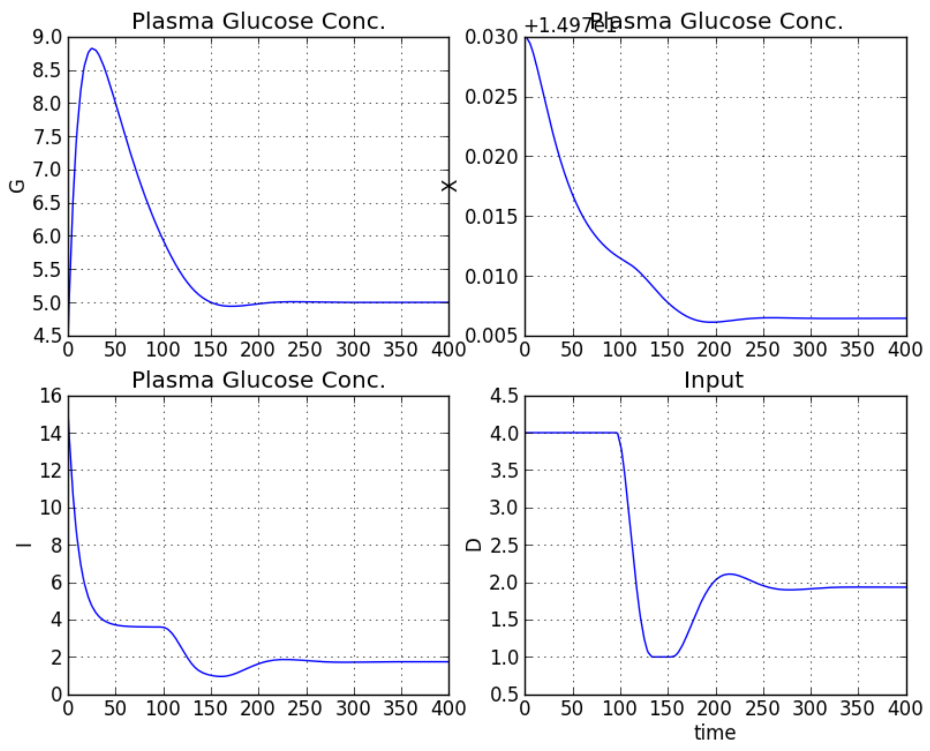


During half of the time the input is highly oscillating between the constraints before it stabilises at 2 mmol/L. The graph for G looks smooth and convenient.

To prevent oscillations for the input a common method is to add a penalty function to the objective. The sum of changes in the input function should be decreased and consequentially the penalty function is assembled of the sum of the squared derivatives of D. Therefore it is necessary to define D as a state variable and set dD as an input with der(D) = dD. The optimization source code is then:

```
optimization BloodGlucose_optconstraint(objective=(cost(finalTime)),
                      startTime=0,
                      finalTime=400)

  input Real u=bc.dD;
  JMExamples.BloodGlucose.BloodGlucoseconstraint bc(G,X,I,D);

  Real cost(start=0,fixed=true);
  parameter Real G_ref = 5;

equation
  der(cost) = (G_ref-bc.G)^2 + u^2;

constraint
  0<=bc.G;
  bc.G<=10;
  bc.D>=1;
  bc.D<=4;
end BloodGlucose_optconstraint;
```

Figure 4.11. BloodGlucose (reference value for G, constraints for D, penalty function)



Now the graph for G as well as for D is smooth and the variation of the input is passable. I achieved a reasonable solution. Depending on additional demands the constraints and penalty function could be adapted.

Chapter 5. Review

1. Internship experience

For me it was a new experience to work in a company for a longer time. At the beginning I was quiet uncertain what to expect and what was expected from me. I got an introduction in the corporate philosophy and the different programs I was supposed to work with. My task was to implement examples in Modelica, Optimica and Python for the example library. Apart from that I was totally free in the choice of my models, in my approach and my time management. This freedom was quiet unexpected and inconvenient for me at the beginning but turned out to be very comfortable. I had regularly meetings with Johan were I was given a leitmotif for my work but I also got the chance to develop ideas in different directions and away from the starting point.

Getting along with new programs (Python, Modelica, JModelica, Dymola, Subversion,...) was very challenging but I could make efforts within a short period of time and learn a lot. I worked independently but there was always somebody to ask when I stuck (Tove+Jesper for the technical support, other students concerning Dymola + Modelica, Hubertus, Johann).

My university knowledge provided a solid base for my work. I could apply different topics such as numerics, matrix theory, thermodynamics and first of all programming languages; more precisely Dymola, Modelica and Python, which were introduced in the "Simulation Tools" course, and Matlab.

Adapted to the fact that the internship was supposed to be a practical experience, I focused more on the performance of the models and disregarded numerical considerations. At the same time it was very interesting for me to get an insight in different technical applications. The physical models were descended from different fields of engineering like thermodynamics, biotechnology, mechanics and chemistry.

2. Approach

The first step when starting with a new topic was always to familiarize with the required programs. Therefore it was helpful and necessary to get through some provided examples. When setting up a new model I acted on the following composition:

- at the beginning collect as much information as possible, use given references
- orientate on equivalent examples or the JModelica UserGuide
- if I totally stuck, ask for help
- try and error
- search for mistakes systematically
- solve other tasks in between and come back to a problem the next day
- set up a similar but less complex model
- motivation (solve easier tasks in between, make a pause, drink a coffee)

I always tried to apply ideas for different methods and models crosswise, for example converting the Penicillin Plant problem to a minimum time problem.

3. Forecast

One of the examples presented in this documentation should be published in the JModelica User Guide. Furthermore the development and enhancements of the model should be represented in a short video documentation.

Chapter 6. Reference list

- Dymola Introduction course 1
- PROPT- Matlab Optimal Control Software (DAE, ODE)
- JModelica User Guide
- AP Monitor Documentation
- Peter Fritzson: Object-oriented modeling and simulation with Modelica 2.1
- Brian J. Driessen, Nader Sadegh: Minimmum- Time Control of Systems With Column Friction: Global Optima Via Mixed Integer Linear Programming
- Elizabeth D. Dolan, Jorge J. More and Todd S. Munson: Benchmarking Optimization Software with COPS 3.0, ARGONNE NATIONAL LABORATORY, Mathematics and Computer Science Division, Technical Report ANL/MCS-TM-273, February 2004
- Dipl. Phys. Moritz Mathias Diehl: Real- Time Optimization for Large Scale Nonlinear Processes, Juni 2001
- B. Srinivasan, D. Bonvin, E. Visser and S. Palanki: Dynamic optimization of batch processes II. Role of measurements in handling uncertainty, July 2000
- Health library, University of Maryland Medical Center home page
- Wikipedia (Definition Simulation, Picture Distillation Column)