A brief introduction to optimization methods

Finn Aakre Haugen

April 29th, 2018

Contents

Preface

These notes give a brief introduction to optimization methods. A section about various applications of optimization is under construction.

Finn Aakre Haugen, PhD
finnhaugen@hotmail.com

Skien, Norway
April 2018

Chapter 1

OPTIMIZATION

1.1 The optimization problem

Optimization is about to find the best solution, for example:

- Which model parameter values makes a mathematical model represent a given real system most accurately?
- Which PI controller settings gives the best performance of a given control system?
- Which are the best future control signals – or control moves – by a model-predictive controller?
- Which are the best estimates calculated by a state estimator?
- Which is the feed rate to a biogas reactor that maximizes the biogas production?

Typically, optimization problems are stated as *minimization* problems:

Find the value of the optimization variable x that
minimizes the *objective function* $f(x)$,
taking into account any constraints on x or on some functions of x.
The solution is denoted the *optimal solution,* x_{opt}.

Figure 1.1 illustrates a minimization problem. Here, the optimization variable, x, is a vector of two elements. The problem is to calculate the combined values of $x(1)$ and $x(2)$ so that f is minimized.

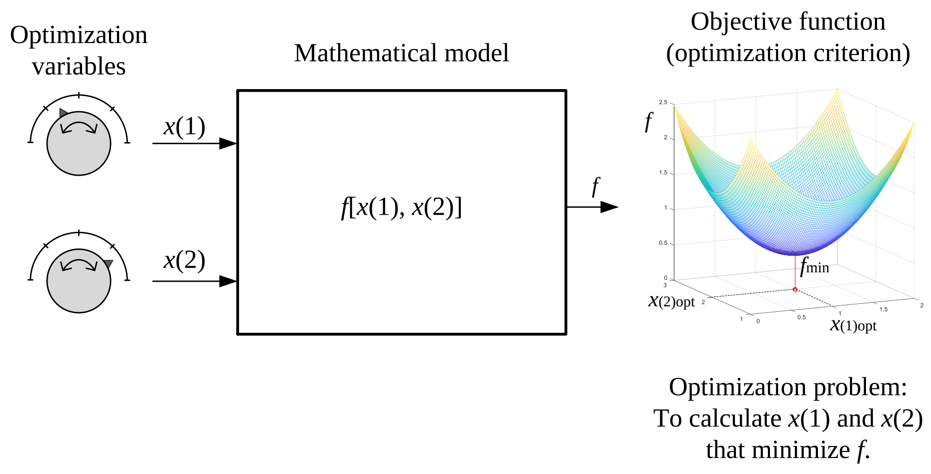

Optimization variables
Mathematical model
Objective function (optimization criterion)
x(1)
x(2)
f[x(1), x(2)]
f
f
fmin
X(2)opt
X(1)opt
Optimization problem: To calculate x(1) and x(2) that minimize f.

Figure 1.1: The optimization problem.

Mathematical formulation of the optimization problem

There are many different ways of formulating mathematically optimization (minimization) problems. The following formulation is quite general. (It complies with the formulation required by the optimization function fmincon in Matlab.)

For a given mathematical model M, find the value of x that minimizes some objective function $f(x)$, that is,

$$\min_x f(x) \tag{1.1}$$

subject to (often denoted "s.t.") *constraints*, which may be in the form of:

- *Inequality constraints*:
$$g(x) \leq 0 \tag{1.2}$$
where g is a linear or nonlinear function.

- *Equality constraints*:
$$h(x) = 0 \tag{1.3}$$
where h is a linear or nonlinear function of x.

- *Lower bounds and upper bounds*:
$$x_{\text{lb}} \leq x \leq x_{\text{ub}} \tag{1.4}$$

(1.3) and (1.2) define constraints on the relation between the optimization variables, while (1.4) define constraints or bounds on the ranges of the optimization variables.

The ranges of the optimization variables in which the optimal solution can be found constitute the *feasible region* of the optimization problem. The constraints define the feasible region.

Some characteristics of the optimal solution

To illustrate some characteristics the optimal solution, we will study a minimization problem with a scalar optimizaton variable.[1] Figure 1.2 shows f as a function of x. The optimization problem is to find the value of x that minimizes f over the shown interval, which is here $2 \leq x \leq 22$. In Example 1.1 the (global) minimum is calculated using the grid search method, in Example 1.5 the Newton search method is used, in Example 1.3 the steepest descent method is used, and in Example 1.7 the steepest descent method and Newton's methods are combined. With any method, the result is

$$f_{\text{min}} = 4.77$$

$$x_{\text{opt}} = 18.8$$

As illustrated in the upper plot of Figure 1.2, f may have *local minima* (or maxima) which are different from the *global minimum* which is the minimum that you want to find. When searching for the global minimum, you must start the search for the minimum sufficiently close to the global minimum. Otherwise, you may get stuck at the local minimum. We will address this challenge in Section 1.2.7.

Some important mathematical characteristics of the optimal solution are:

- f is flat, that is, $f'(x) = 0$ at $x = x_{\text{opt}}$.
- f is *convex*, that is, $f''(x) = 0$ at $x = x_{\text{opt}}$.

Above, the optimization variable, x, is a scalar. In general, the optimization variable is a vector: $x = [x(1), x(2),..., x(n)]^{\text{T}}$.

What about maximization problems?

Suppose the optimization problem is to *maximize* a function, f. You can turn that maximization problem into a minimization problem by

[1]This example is "home-made" for illustration purposes.

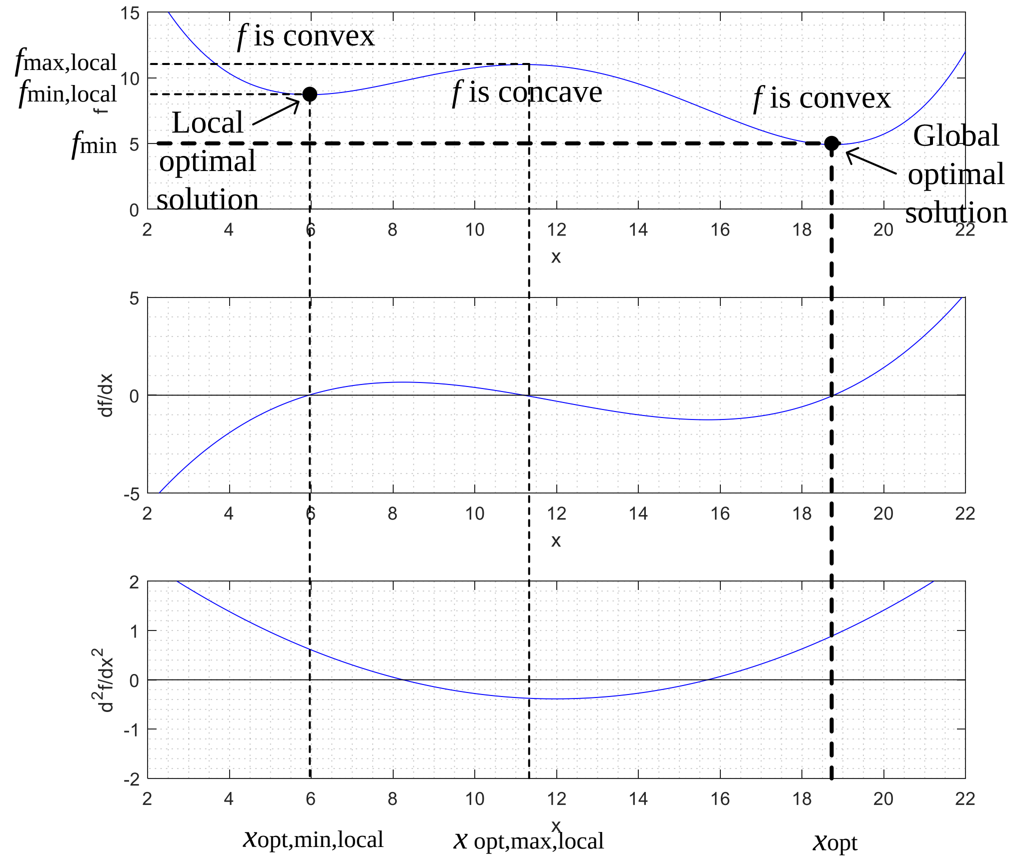


Figure 1.2: Optimization problem: Find $x = x_{\text{opt}}$ that minimizes $f(x)$.

multiplying f by -1. Both problems will have the same solution, x_{opt}, see Figure 1.3. So, if you can solve *minimization* problems, you can also solve maximization problems.

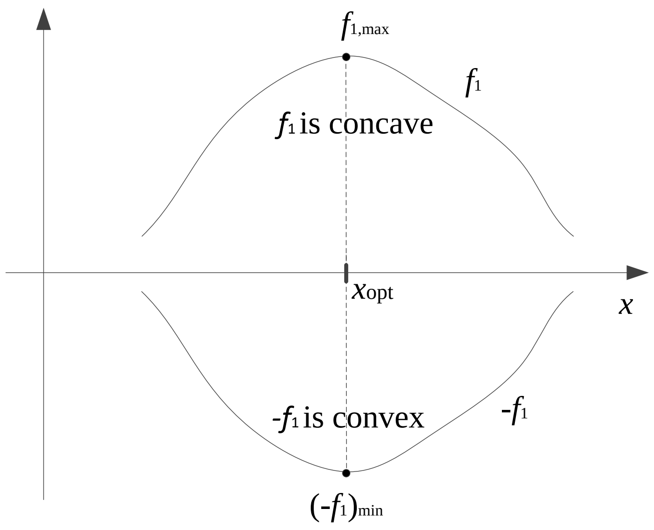


Figure 1.3: The same x_{opt} maximizes f_1 and minimizes $-f_1$.

1.2 How to solve optimization problems

1.2.1 Introduction

There several forms of optimization problems, and there are several methods for finding the optimal solution [Edgar *et al.*, 2001], [Nocedal & Wright, 2006]. The following optimization methods are presented briefly in the following sections:

- The *grid search method*, which is also denoted the brute force method.
- The *steepest descent search method.*
- The *Newton search method.*

- A *combination* of the steepest descent and Newton's method (to exploit their benefits, while avoiding their drawbacks).
- Two professional optimization functions, namely *fmincon* (Matlab) and *sqp* (Octave).

1.2.2 The grid search for optimum

Many optimization problems have just a few optimization variables, e.g. four or less. For such problems, an approximate global optimal solution may be easily obtained with the grid search method. This is a simple, straightforward method which can be implemented in any computer language, and no special optimization tools (algorithms) are needed. The basic principle is calculating the objective function, f, for all possible combinations of the optimization variables, $x(1)$, $x(2)$,... , $x(n)$ within their ranges. Each of these ranges spans a proper number of equally spaced values of the pertinent optimization variable, for example 100 values or grid points of each variable. The optimal solution, x_{opt}, is the x-value which corresponds to the minimum (or maximum) value of f.

The grid search method is simple and easy to implement, but have some drawbacks, too:

- A limited accuracy of the optimal solution.
- A large computational burden if the number of optimization variables is large and/or the objective function is computational demanding. Imagine 5 with 100 grid points for each. The total number of grid points, combinations, and function calls are then $100^5 = 10^{10}$ which is a large number.

With a scalar optimization variable, x, the grid search can be implemented by calculating f for each value of x in one For Loop.

Figure 1.4 illustrates the grid for the case of two optimization variables, $x(1)$ and $x(2)$ with $N_{x_1} \times N_{x_2} = 20 \times 20 = 400$ grid points. The grid search can be implemented with *nested For Loops*, which in this example means that for each value of $x(1)$, the objective function f is calculated for all of the values of $x(2)$, see Figure 1.5.

Example 1.1 ***Grid search with scalar x***

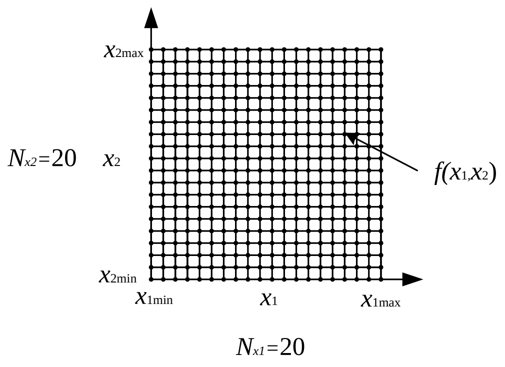


Figure 1.4: Grid with two optimization variables, $x(1)$ and $x(2)$ with $20 \times 20 = 400$ grid points.

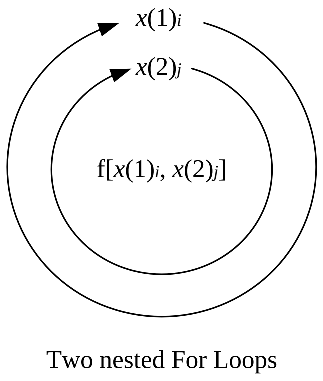


Figure 1.5: Two nested For Loops. f is calculated for every combination of the values of $x(1)$ and $x(2)$. Specifically, for each value of $x(1)$, f is calculated for all of the values of $x(2)$.

The grid search method is here applied to find the global optimal solution of the function plotted in Figure 1.2. The function is

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \tag{1.5}$$

with

$$a_4 = 0.00232,\ a_3 = -0.111,\ a_2 = 1.80,\ a_1 = -11.6,\ a_0 = 34.4 \tag{1.6}$$

The range of the optimization variable is

$$2 \le x \le 22$$

The Matlab script below implements a grid search to find the global optimal solution. The number of x-values is selected as $N = 100$, giving a resolution of $r_x = (22 - 2)/100 = 0.2$. (100 is selected for illustration purposes.) The computational load is still small even if N was increased to e.g. 10000 in this simple application. Figure 1.6 shows $f(x)$ and the optimal solution, which is

$$f_{\text{min}} = 4.77$$

$$x_{\text{opt}} = 18.8$$

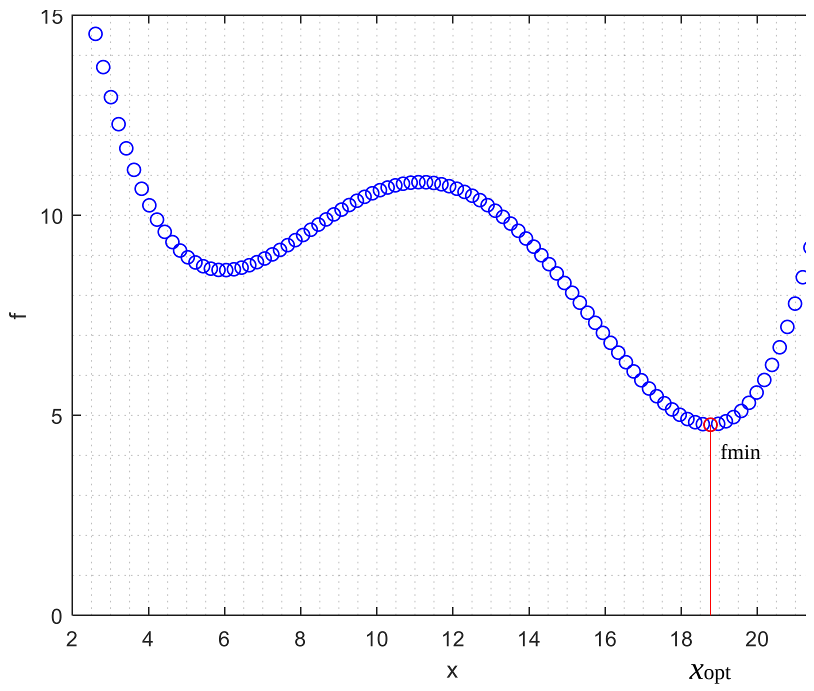


Figure 1.6: Example 1.1: The optimal result with the grid search method.

Script name: matlab_script_init_grid_search.m.

```
x_min=2;
x_max=22;
N_x=100;
x_array=linspace(x_min,x_max,N_x);%Preallocation for plotting
f_array=x_array*0;%Preallocation for plotting

%Initialization:
f_min=inf;
x_opt=-inf;

%Creating anonymous function for the objective function:
```

```
f_obj=@(x) x^4*0.00232-x^3*0.111+x^2*1.80-x*11.6+34.4;

%For loop than runs through all x-values:
for
  k_x=1:length(x_array)
  x=x_array(k_x);
  %Objective function:
  f=f_obj(x);
  %Possibly improving the previous solution:
  if f <= f_min f_min=f;
    x_opt=x;
  end %If

  %Storing objective function values for plotting:
  f_array(k_x)=f;
  x_array(k_x)=x;
end %For

disp('Optimal solution:')
f_min
x_opt
```

The result shown in Matlab is:

```
Optimal solution:
f_min = 4.7664
x_opt = 18.7677
```

[End of Example 1.1]

The following example describes a grid search with two optimization variables, in the first case without any constraint about the relation between the optimization variables, and in the second case, with such a constraint.

Example 1.2 ***Grid search with vectorial x - without and with a constraint***

First case: No constraints on the relation between the optimization variables

The optimization problem is:

$$\min_x f(x) \tag{1.7}$$

with

$$f(x) = (x_1 - 1)^2 + (x_2 - 2)^2 + 0.5 \tag{1.8}$$

The bounds on the optimization variables are:

$$0 \le x_1 \le 2 \tag{1.9}$$

$$1 \le x_2 \le 3 \tag{1.10}$$

It is apparent from (1.8) that the optimal solution is:

$$f_{\min} = 0.5 \tag{1.11}$$

$$x_{1_{\text{opt}}} = 1 \tag{1.12}$$

$$x_{2_{\text{opt}}} = 2 \tag{1.13}$$

Figure 1.7 shows a plot of f and the optimal solution. A Matlab script that implements the grid search is presented below.

Second case: Constraint on the relation between the optimization variables

Assume the following constraints on the *relation* between the optimization variables:

$$x(1) - x(2) + 1.5 \le 0 \tag{1.14}$$

which is on the form of (1.2). The feasible region of the optimization problem is now given by this inequality together with the ranges (1.9) – (1.10).

Figure 1.8 shows a plot of f given by (1.8) in the feasible region. As is often the case when there are constraints on the relation between the optimization variables, the optimal solution is on the border of the feasible region.

As found with the Matlab script below, the optimal solution is now

$$f_{\min} = 0.628 \tag{1.15}$$

$$x(1)_{\text{opt}} = 0.748 \tag{1.16}$$

$$x(2)_{\text{opt}} = 2.25 \tag{1.17}$$

Below is a Matlab script that implements the grid search method for the second case presented above. This case has the constraint (1.14). The script also applies for the first case presented above. That case has no constraint on the relation between $x(1)$ and $x(2)$. The script implements that case if the following part of the script is removed:

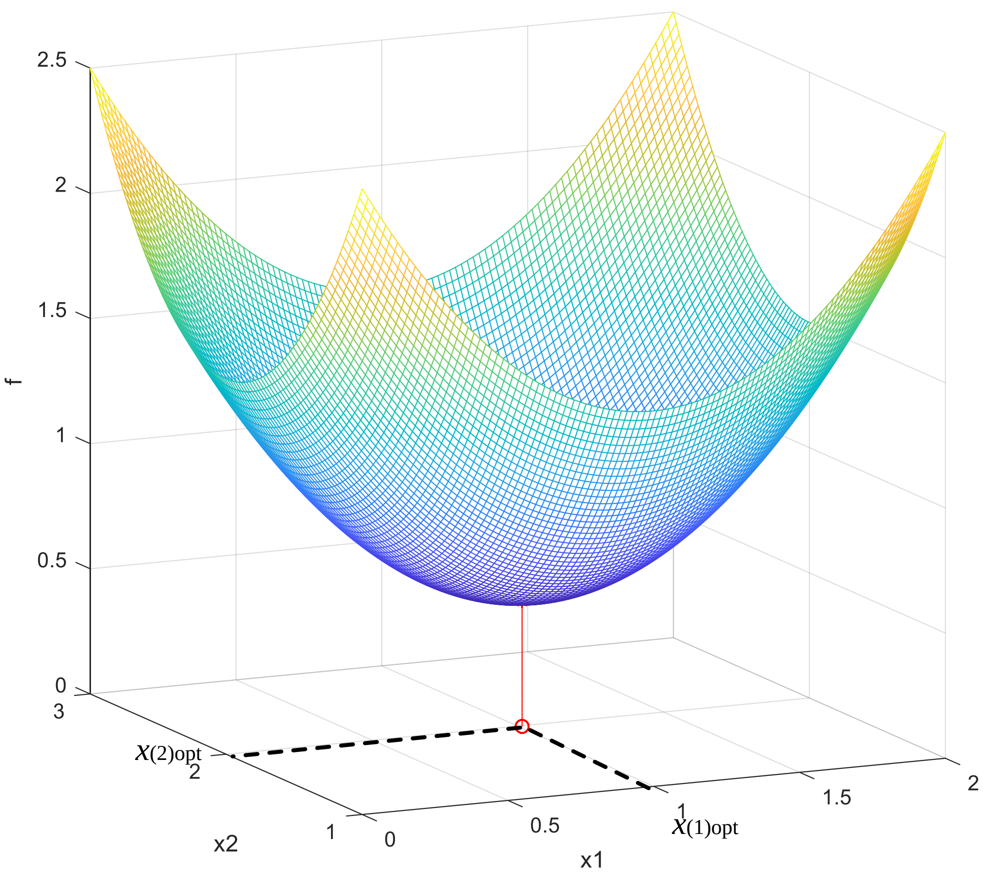


Figure 1.7: Example 1.2: A plot of f and the optimal solution as found with the grid method.

```
%Constraint:
if not(x1-x2+1.5 <= 0)
  f=inf;
end %if
```

Script name: matlab_script_grid_search.m.

```
clear all; close all; format compact;

%Initialization:
x1_min=0;x1_max=2;N_x1=100;
x1_array=linspace(x1_min,x1_max,N_x1);
```

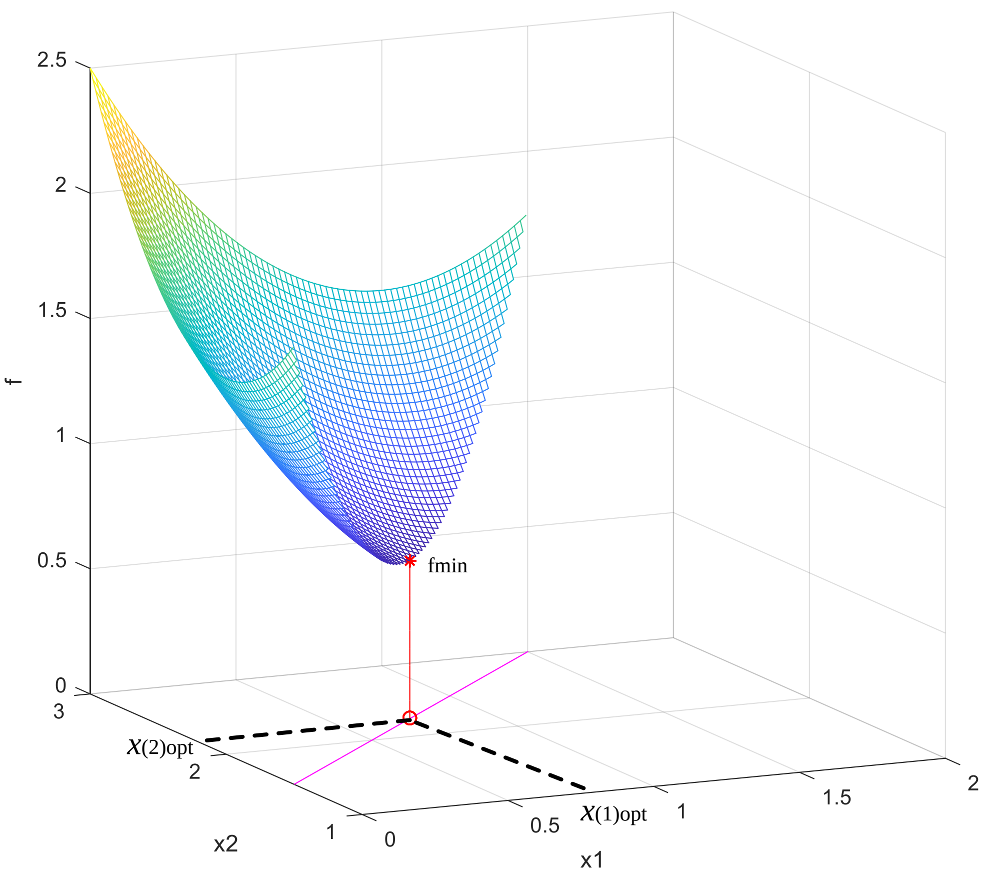


Figure 1.8: The grid method. The horizontal line represents the inequality constraint (1.14).

```
x2_min=1;x2_max=3;N_x2=100;
x2_array=linspace(x2_min,x2_max,N_x2);
f_min=inf;
x1_opt=-inf;
x2_opt=-inf;

%Objective function defined as 'anonymous' function:
fun_obj=@(x1,x2) (x1-1)^2+(x2-2)^2+0.5;

for k_x1=1:length(x1_array)
x1=x1_array(k_x1);
  for k_x2=1:length(x2_array)
```

```
    x2=x2_array(k_x2);
    %Value of objective function:
    f=fun_obj(x1,x2);
    %Constraint:
    if not(x1-x2+1.5 <= 0)
      f=inf;
    end %if
    %Improving the previous solution:
    if f <= f_min
      f_min=f;
      x1_opt=x1;
      x2_opt=x2;
    end%if
    %Storing objective function values for later plotting:
    f_matrix(k_x1,k_x2)=f;
  end %for loop of x2
end %for loop of x1

disp('Optimal solution:')
f_min x1_opt x2_opt
```

[End of Example 1.2]

1.2.3 Steepest decent search method

The steepest descent method is for solving *unconstrained* optimization problems. In the steepest descend method, the optimization variable, x, is moved so that the value of the objective function is the largest reduction possible. It is like trying to get to the bottom of a valley by always walking down as steeply as possible.

Scalar x

In the scalar case (x scalar), the steepest descend iteration is:

$$x_{k+1} = x_k + \Delta x_k \tag{1.18}$$

where the step is

$$\Delta x_k = -K f'(x_k) \tag{1.19}$$

where K is a factor which can be used to determine the size or length of the step, and $f'(x_k)$ is the derivative, or the gradient, of the objective function. In the standard steepest descend search, the numerical value of K is 1.

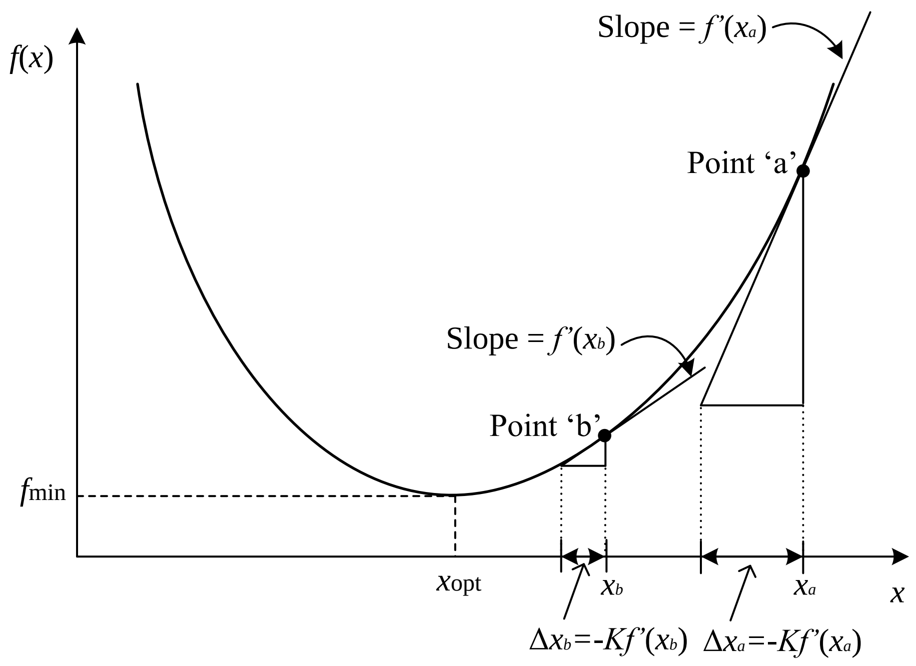


Figure 1.9: In the steepest descent method the search step size, Δx_k is proportional to the derivative of the objective function, $f'(x_k)$.

Δx_k is proportional to $f'(x_k)$. This is illustrated in Figure 1.9 where two different values of x are considered. This proportionality implies that the closer to the optimum (minimum of f), the smaller the step. This sounds like the minimum will be found. However, it may happen that Δx_k becomes too large, so that x_{k+1} will pass x_{opt}, causing the search to "jump" to other side of the "valley". Consequently, there may be oscillations in the search. There are several methods to improve (optimize) the step size, e.g. conjugate gradient methods [Edgar *et al.*, 2001], and variants of Newton's method. It is also possible to just manually set K to a value less than the default of 1. These are methods that apply also for the vectorial case (x a vector).

Vectorial x

In the vectorial case, the steepest descend iteration is:

$$x_{k+1} = x_k + \Delta x_k \tag{1.20}$$

where the search step is

$$\Delta x_k = -K \nabla f(x_k) \tag{1.21}$$

where $\nabla f(x_k)$ is the gradient of f, calculated at x_k. So, the step, Δx_k, is taken in the negative direction of the gradient, or – in other words – along the steepest descent. In the standard steepest descend search, the numerical value of K is 1, or, which gives the equivalent result, $K = I$, the identity matrix. There are several methods to improve (optimize) the step size and the direction, but we will not discuss these improvements here, except we will look at Newton's method in Section 1.2.4.

Numerical calculation of the derivative

The gradient in (1.19) may be calculated analytically or numerically. For numerical calculation, the center difference approximation is probably appropriate:

$$\nabla f(x_k) \approx \begin{bmatrix} \frac{f[x(1)_k+h,x(2)_k,\ldots,x(n)_k]-f[x(1)_k-h,x(2)_k,\ldots,x(n)_k]}{2h} \\ \frac{f[x(1)_k,x(2)_k+h,\ldots,x(n)_k]-f[x(1)_k,x(2)_k-h,\ldots,x(n)_k]}{2h} \\ \vdots \\ \frac{f[x(1)_k,x(2)_k,\ldots,x(n)_k+h]-f[x(1)_k,x(2)_k,\ldots,x(n)_k-h]}{2h} \end{bmatrix} \tag{1.22}$$

where the increment size h can be selected as a very small number, for example 10^{-4} (independently of the search step size).

To summarize:

Steepest descent search method for minimization:

1. Make a good guess, x_{guess}, of the optimal solution, and set

$$x_0 = x_{\text{guess}} \tag{1.23}$$

2. Iterate with

$$x_{k+1} = x_k + \mathit{\Delta} x_k \tag{1.24}$$

where the increment $\mathit{\Delta} x(x_k)$ is

$$\mathit{\Delta} x_k = -K\nabla f(x_k) \tag{1.25}$$

The standard value of K is 1, but is can be reduced to obtain a smoother, but slower, search.

Continue the iterations until an appropriate stop condition is satisfied, e.g.

$$|f(x_{k+1}) - f(x_k)| \leqslant \mathrm{d}f \tag{1.26}$$

When the stop condition is met,

$$x_{\text{opt}} = x_{k+1} \tag{1.27}$$

Example 1.3 ***Steepest descent search - scalar*** x

We will use Newton search to find the global optimal solution of the function plotted in Figure 1.2. The function is given in Example 1.1, but is repeated here for convenience:

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \tag{1.28}$$

where

$$a_4 = 0.00232,\ a_3 = -0.111,\ a_2 = 1.80,\ a_1 = -11.6,\ a_0 = 34.4 \tag{1.29}$$

The guessed value is selected as

$$x_{\text{guess}} = 12$$

Below is a Matlab script that implements the search.

Figure 1.10 shows f, x_{guess}, and x_{opt}.

The results as shown in Matlab are:

```
x_guess = 12
x_opt = 18.7476
f_min = 4.7662
abs_df = 1.7186e-05

[k,grad,dx,x,abs_df]
1.0000 -0.3162 0.3162 12.3162 0.1191
2.0000 -0.4369 0.4369 12.7530 0.2267
3.0000 -0.6000 0.6000 13.3530 0.4240
4.0000 -0.8095 0.8095 14.1625 0.7552
5.0000 -1.0456 1.0456 15.2081 1.2028
6.0000 -1.2275 1.2275 16.4356 1.5132
7.0000 -1.1841 1.1841 17.6196 1.2064
8.0000 -0.7877 0.7877 18.4073 0.4362
9.0000 -0.2850 0.2850 18.6923 0.0484
10.0000 -0.0497 0.0497 18.7420 0.0014
11.0000 -0.0056 0.0056 18.7476 0.0000
```

Note that $x_{\text{guess}} = 12$ is to the right of the maximum point, see Figure 1.10. Since f' is monotonically decreasing between the starting point of the search and the global minimum, the search will approach that minimum.

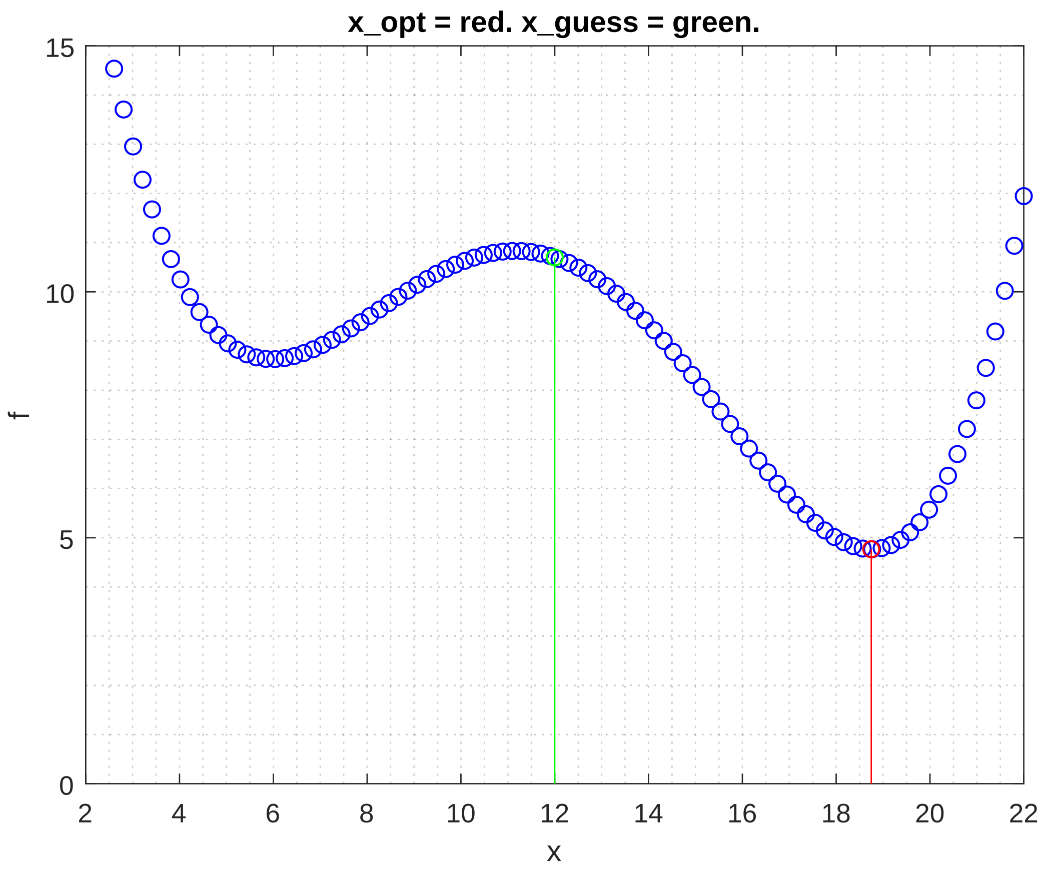


Figure 1.10: Example 1.3: Steepest descent search with $x_{\text{guess}} = 12$. x_{opt} is at global minimum.

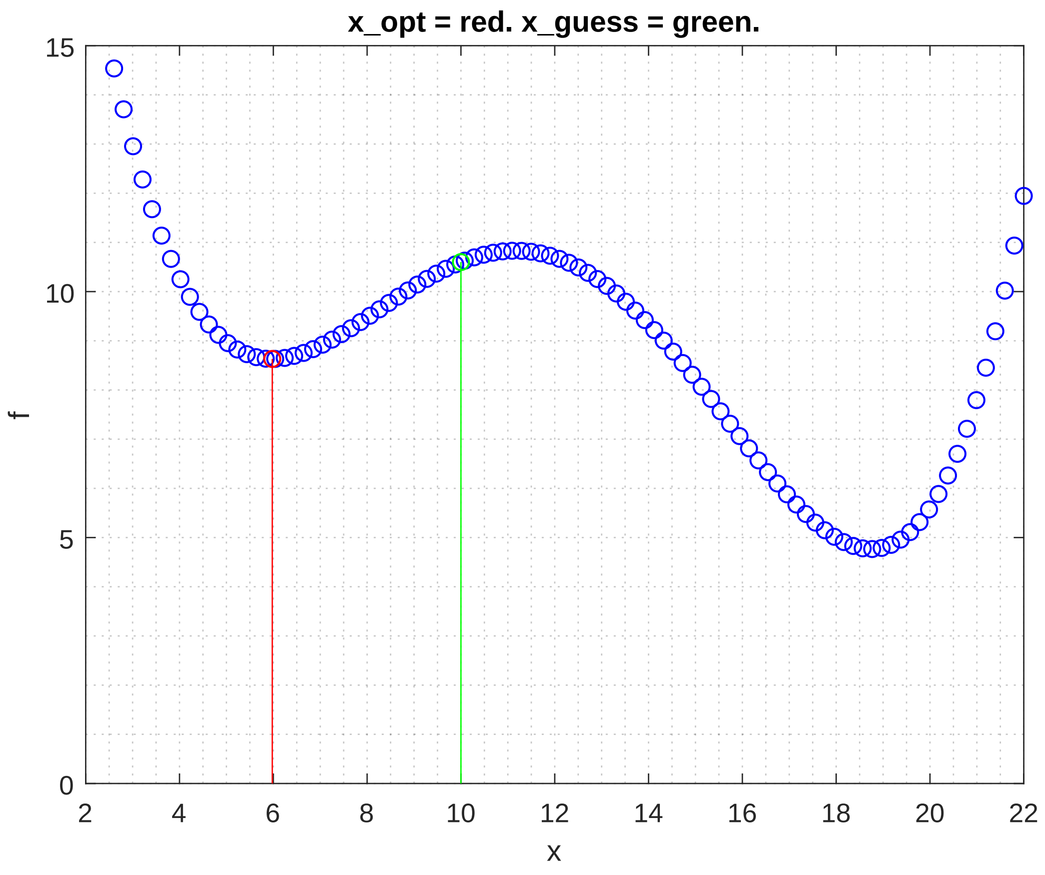


Figure 1.11: Example 1.3: Steepest descent search with $x_{\text{guess}} = 10$. x_{opt} is at a *local* minimum.

To demonstrate that the starting point (the guessed value) is crucial for the result, let us select

$$x_{\text{guess}} = 10$$

which is to the left of the maximum point, see Figure 1.10. Now, the result is:

```
x_guess = 10
x_opt = 5.9775
f_min = 8.6305
```

which is a *local* minimum, different from the global minimum. So, the starting point is crucial for the result of the search.

Script name: matlab_script_steepest_descent_scalar.m.

```
clear all, close all, format compact
a4=0.00232; a3=-0.111; a2=1.80; a1=-11.6; a0=34.4;
%Creating anonymous function for the objective function:
f_obj=@(x) a4*x.^4 + a3*x.^3 + a2*x.^2 + a1*x + a0;
x_guess=[10]
x_k=x_guess;
h=1e-4; %Step size in center difference method
N=1000;%Preset max number of iterations
abs_df_spec=1e-4;%Stopping criterion
for k=1:N-1
  gradient_num_k=(f_obj(x_k+h)-f_obj(x_k-h))/(2*h);
  dx_k=-gradient_num_k;
  x_kp1=x_k+dx_k;
  f_k=f_obj(x_k);
  f_kp1=f_obj(x_kp1);
  abs_df=abs(f_kp1-f_k);
  x_k=x_kp1;
  if abs_df < abs_df_spec
    break
  end %if
end %for loop
disp('Result:')
x_opt=x_kp1
f_min=f_obj(x_opt)
abs_df
```

[End of Example 1.3]

In the following example, x is vectorial.

Example 1.4 ***Steepest descent search - vectorial*** x

We will now make a steepest descent search to solve the optimization problem already presented in Example 1.2. For convenience, the problem formulation is repeated here:

$$\min_x f(x) \tag{1.30}$$

where

$$f(x) = [x(1) - 1]^2 + [x(2) - 2]^2 + 0.5 \tag{1.31}$$

The stop criterion is selected as

$$|f(x_{k+1}) - f(x_k)| \leq \mathrm{d}f = 10^{-4}$$

The gradient is:

$$\nabla f(x_k) = \begin{bmatrix} 2x(1)_{\mathrm{k}} - 2 \\ 2x(2)_k - 4 \end{bmatrix}$$

Obviously, the optimal solution is

$$f_{\min} = 0.5$$

at

$$x_{\mathrm{opt}} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

We calculate the first iteration:

$$x_1 = x_0 - \nabla f(x_0) = x_0 - \begin{bmatrix} 2x(1)_0 - 2 \\ 2x(2)_0 - 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} -2 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Checking the stop condition:

$$|f(x_1) - f(x_0)| = \left| f\left(\begin{bmatrix} 2 \\ 3 \end{bmatrix} \right) - f\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) \right| = |2.5 - 2.5| = 0 \leqslant 10^{-4} \text{ (Yes!)}$$

So, the stop condition is met after the first iteration since $f(x_1) = f(x_0)$. However, optimum is not found! One explanation is that the step size is too large, as a consequence of the gradient being relatively large.

To approach (much) closer to the optimum, the step size must be reduced. Although rigorous methods exist (as pointed out in the beginning of this section), we will here just observe the effect of a reduction of the step size. Let's take the iteration as

$$x_{k+1} = x_k - K\nabla f(x_k)$$

with

$$K = 0.1$$

The result, as calculated with the Matlab script shown below, is now:

$$f_{\min} = 0.5002$$

$$x_{\mathrm{opt}} = \begin{bmatrix} 0.9908 \\ 1.9908 \end{bmatrix}$$

which is very close to the correct (analytical) result presented above.

The number of iteration until the stop condition is met, is 21.

A Matlab script implementing the above is shown below.

Script name: matlab_script_steepest_descent_2_vars.m.

```
%Creating anonymous function for the objective function:
f_obj=@(x) (x(1)-1)^2+(x(2)-2)^2+0.5;
x_guess=[0,1]';
%Guess x_k=x_guess;
N=10000;%Preset max number of iterations
abs_df=1e-4;%Stopping criterion
for k=1:N-1
  G_k=[2*(x_k(1)-1), 2*(x_k(2)-2)]'; %Gradient:
  K=0.1;
  dx_k=-K*G_k
  x_kp1=x_k+dx_k;
  f_k=f_obj(x_k);
  f_kp1=f_obj(x_kp1);
  df=f_kp1-f_k
  x_k=x_kp1;
  if abs(df) < abs_df
    break
  end %if
end %for loop
disp('Result:')
k
x_opt=x_kp1
f_min=f_obj(x_opt)
```

The result is as shown above.

[End of Example 1.4]

1.2.4 The Newton search method

The Newton search method is an iterative method for solving *unconstrained* optimization problems. The method does not take into account any constraints on the form of (1.3) and (1.2). Optimization methods that do take constraints into account are often denoted Nonlinear Programming (NLP) methods. Section 1.2.6 introduces two professional NLP solvers, namely fmincon in Matlab and sqp in Octave. Concepts that are central in this method are central also in solvers for constrained problems (NLP solvers).

The Newton search method will here be introduced assuming only a scalar optimization variable, x. Thereafter, the Newton search method with a vectorial optimization variable, $x = [x(1), x(2),..., x(n)]^{\mathrm{T}}$, is presented.

Scalar x

See Figure 1.2. At the minimum of f, f is flat, that is,

$$f'(x_{\text{opt}}) = 0 \tag{1.32}$$

Generally, Newton's method is an iterative method for solving equations on the form $F(x) = 0$. In the context of optimization, Newton's method is used to solve (1.32) for x. But since minimization points as well as maximizing points are characterized by $f' = 0$, as illustrated in Figure 1.2, we can not just take the solution of (1.32) as the minimizing solution. By solving $f'(x) = 0$ for x, we only have a *candidate* of the optimal x, say x_{cand}. To ensure that x_{cand} is actually the optimal (minimizing) solution, we must check that f is *convex* where $f'(x_{\text{cand}}) = 0$. f is convex at x_{cand} if the second order derivative of f is strictly positive at x_{cand}, that is, if

$$f''(x_{\text{cand}}) > 0 \tag{1.33}$$

x_{cand} is the results of a number of Newton iterations. Figure 1.12 illustrates one iteration.

We assume that an estimate, or a candidate, of the optimal x exists at iteration number k, namely x_k, and that both $f'(x_k)$ and $f''(x_k)$ are known (at x_k). An improved estimate can be obtained graphically as shown in Figure 1.12. We will now find the formula of x_{k+1}. From Figure 1.12 we find that the slope at x_k is

$$f''(x_k) = \frac{f'(x_k) - 0}{x_k - x_{k+1}} \tag{1.34}$$

Solving for x_{k+1} gives the Newton iteration expressed as a formula:

$$x_{k+1} = x_k - \left[f''(x_k)\right]^{-1} f'(x_k) \tag{1.35}$$

How many Newton iterations should be calculated? In a computer program, the Newton iteration can be implemented in a While Loop with the stop condition of the loop being, for example,

$$|f(x_{k+1}) - f(x_k)| \leqslant \mathrm{d}f \tag{1.36}$$

where $\mathrm{d}f$ is a constant of an appropriate value. Alternatively, a For Loop can be used with a fixed, maximum number of iterations (Newton

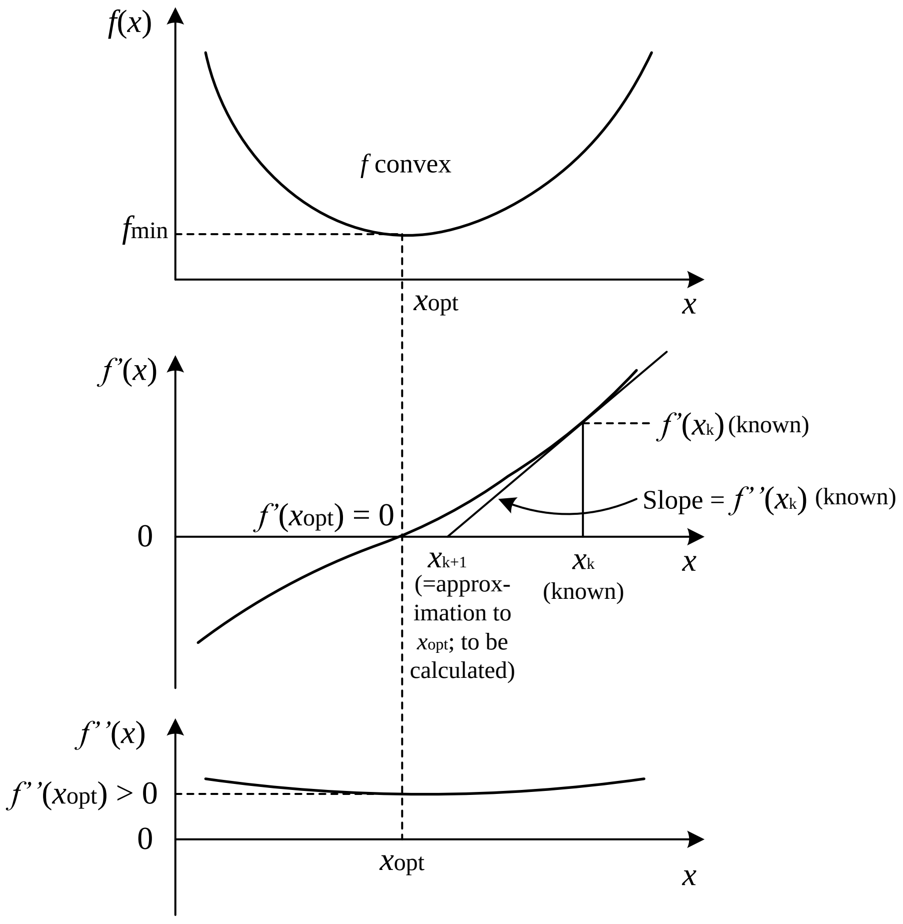


Figure 1.12: One iteration in Newton's method for searching for x_{opt}.

iterations) and with the possibility to break the loop if (1.36) is satisfied before the maximum number of iterations is reached.

Once we have found a candidate, $x_{\text{cand}} = x_{k+1}$, of the optimal solution, we must check that $f''(x_{\text{cand}}) > 0$. If this test is passed,

$$x_{\text{opt}} = x_{\text{cand}} \tag{1.37}$$

Note that the Newton's search method is a *local optimizer (minimizer)*. It is only if x_{guess} is sufficiently close to the global optimum, that the Newton search will arrive at the *global* optimum. As an example, see 1.2. If $x_{\text{guess}} = 4$, a Newton search will arrive at a local optimum. If $x_{\text{guess}} = 16$, the Newton search will arrive at the global optimum.

In the Newton's method, x will be moved towards a point where

$$f'(x_{\max}) = 0$$

However, this point may be a (local) maximum if the initial value of x in the search is where f is concave. Figure 1.13 illustrates this situation. The Newton iteration, (1.35), will move x towards $x_{\max}$. So, it is crucial that the guessed value of x is where f is not concave, i.e., is convex. This situation can be avoided by using the steepest descent method in stead of Newton's method where f is not concave, cf. Section 1.2.5.

An explanation of why the Newton search is fast

Let's consider the steepest descent algorithm for the scalar case, (1.18) – (1.19), which is repeated here for convenience:

$$x_{k+1} = x_k - Kf'(x_k) \tag{1.38}$$

where the factor K is included. K can be regarded as the step size. In the standard steepest descent method, $K = 1$ (fixed). Now, let's open for other values of K. (1.38) can be regarded as a nonlinear discrete-time difference equation or model with x as the state-variable. It is nonlinear because it can be assumed that the derivative, or gradient, $f'(x)$, is a nonlinear function of x. To analyze the dynamic properties of this model, we can consider the linearized model. Let's define dx as the deviation variable corresponding to x. Linearization of (1.38), which is based on a first order Taylor series of the nonlinear term, gives the following linear model:

$$dx_{k+1} = dx_k - Kf''(x_k)dx = \left[1 - Kf''(x_k)\right] dx_k \tag{1.39}$$

According to systems theory of discrete-time systems, the fastest dynamic response in dx is obtained with the next state, dx_{k+1}, is assumed zero,

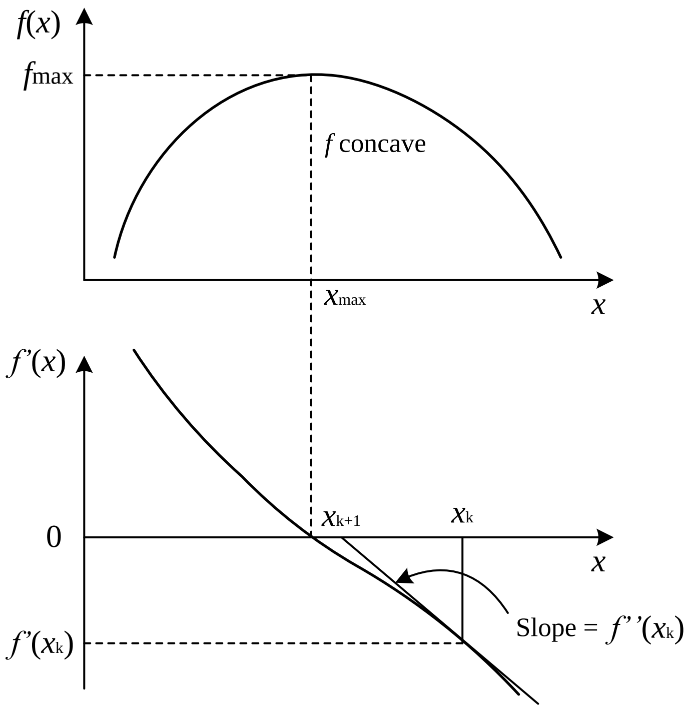


Figure 1.13: One iteration in Newton's method when f is concave.

independent of the present state, dx_k.[2] This dynamics is denoted dead-beat response. So, dead-beat dynamics is obtained with

$$1 - Kf''(x_k) = 0$$

which gives

$$K = \left[f''(x_k)\right]^{-1}$$

Inserting this K into (1.39), gives the "dead-beat steepest descent" algorithm:

$$x_{k+1} = x_k - \left[f''(x_k)\right]^{-1} f'(x_k)$$

which is the Newton search algorithm, (1.35)!

For a quadratic f, the optimum is found in one search iteration!

Assume that $f(x)$ is quadratic, say

$$f(x) = a_1 + a_2 x^2$$

[2]An alternative design based on z-plane theory is as follows: The fastest dynamics of a discrete-time system is when the z-eigenvalue(s) of the system being zero, i.e. in the origin of the complex z-plane. The z-eigenvalue is $z = 1 - Kf''(x_k)$, which is set to 0.

This implies:

$$f'(x) = 2a_2x$$

and

$$f''(x) = 2a_2$$

Inserting these functions in the Newton algorithm, gives:

$$x_{k+1} = x_k - \left[f''(x_k)\right]^{-1} f'(x_k) = x_k - [2a_2]^{-1} 2a_2x_k = 0x_k$$

Thus, the Newton algorithm exhibits dead-beat dynamics, and this result is obtained directly, without any approximate analysis involving linearization. This implies that for quadratic functions, the search will arrive at the minimum in just one iteration, whatever is selected as the starting or guessed value of x of the search. This holds also for optimization problems where x is vectorial. Example 1.6 gives a demonstration.

Vectorial x

Assume x a vector:

$$x = \begin{bmatrix} x(1) \\ x(2) \\ \vdots \\ x(n) \end{bmatrix}$$

It can be shown [Edgar *et al.*, 2001] that the Newton iteration, (1.35), now takes the form

$$x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \tag{1.40}$$

In (1.40),

$$\nabla f(x_k) = \begin{bmatrix} \frac{\partial f(x_k)}{\partial x(1)} \\ \frac{\partial f(x_k)}{\partial x(2)} \\ \vdots \\ \frac{\partial f(x_k)}{\partial x(n)} \end{bmatrix} \tag{1.41}$$

which is the *gradient* of f with respect to x. In words, the gradient of f is a vector of first order partial derivatives of f. In (1.41), $\partial f(x_k)/\partial x(1)$ means the partial derivative of f with respect to $x(1)$, calculated at $x = x_k$.

Furthermore in (1.40),

$$\nabla^2 f(x_k) \equiv H(x_k) = \begin{bmatrix} \frac{\partial^2 f}{\partial x(1)^2} & \frac{\partial^2 f}{\partial x(1)\partial x(2)} & \cdots & \frac{\partial^2 f}{\partial x(1)\partial x(n)} \\ \frac{\partial^2 f}{\partial x(2)\partial x(1)} & \frac{\partial^2 f}{\partial x(2)^2} & \cdots & \frac{\partial^2 f}{\partial x(2)\partial x(n)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x(n)\partial x(1)} & \frac{\partial^2 f}{\partial x(n)\partial x(2)} & \cdots & \frac{\partial^2 f}{\partial x(n)^2} \end{bmatrix} \tag{1.42}$$

which is the *Hessian* (matrix) of f with respect to x. In words, the Hessian of f is a matrix of second order partial derivatives of f.

As for the scalar case discussed above, the Newton iteration can be implemented in a While Loop or For Loop (the latter with a preset maximum number of iterations) with the stop or break condition of the loop being

$$|f(x_{k+1}) - f(x_k)| \leqslant \mathrm{d}f \tag{1.43}$$

where $\mathrm{d}f$ is a constant of an appropriate value.

To verify that x_{opt} minimizes and not maximizes f, we must check that f is convex at x_{opt}. f is convex if the Hessian is positive definite, which is ensured if all the eigenvalues of the Hessian are strictly positive [Edgar *et al.*, 2001]. Hence, x_{opt} minimizes f if

$$\forall\, \mathrm{eig}\, \nabla^2 f(x_{\text{cand}}) > 0 \tag{1.44}$$

(The symbol $\forall$ means "for each".)

Avoiding the inversion of the Hessian

The Newton iteration, 1.40, can be written as

$$x_{k+1} = x_k + \mathit{\Delta} x_k \tag{1.45}$$

where

$$\mathit{\Delta} x_k = -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \tag{1.46}$$

Generally, the inverse of matrices should not be calculated because the inverse may be mathematically ill-conditioned, i.e. sensitive to numerical errors. If possible, you should instead express the unknown as the solution of an equivalent systems of linear equations, and solve for the unknown, which in our case is $\mathit{\Delta} x_k$, using robust algorithms or functions for solving systems of linear equations, for example using the "\" operator in Matlab. By premultiplying (1.46) by the $-\nabla^2 f(x_k)$ (minus one times the Hessian), we get the following system of linear equations with $\mathit{\Delta} x_k$ as the unknown:

$$-\nabla^2 f(x_k)\mathit{\Delta} x_k = \nabla f(x_k) \tag{1.47}$$

In Matlab, we can solve (1.47) for $\mathit{\Delta} x(x_k)$ with code like

```
dx_k = -H_k\grad_k
```

which is better numerically than

```
dx_k = -inv(H_k)*grad_k
```

Numerical calculation of the gradient and the Hessian

In Example 1.6, both the gradient and the Hessian of f was calculated analytically. Alternatively, they can be calculated numerically from calculations of the objective function f only. The center difference approximation to the derivative, (1.22), may be appropriate.

To sum it up:

Newton's search method method for minimization:

1. Make a good guess, x_{guess}, of the optimal solution, and set

$$x_0 = x_{\text{guess}} \tag{1.48}$$

2. Iterate with

$$x_{k+1} = x_k + \Delta x_k$$

where the increment $\Delta x(x_k)$ may be calculated directly by

$$\Delta x_k = -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \tag{1.49}$$

or, preferably, indirectly by solving the following system of linear equation for $\Delta x(x_k)$:

$$-\nabla^2 f(x_k) \Delta x_k = \nabla f(x_k) \tag{1.50}$$

Continue the iterations until an appropriate stop condition is satisfied, e.g.

$$|f(x_{k+1}) - f(x_k)| \leqslant \mathrm{d}f \tag{1.51}$$

The candidate of the optimal solution is then

$$x_{\text{cand}} = x_{k+1} \tag{1.52}$$

3. If f is convex at x_{cand}, the optimal solution has been found, that is,

$$x_{\text{opt}} = x_{\text{cand}} = x_{k+1} \tag{1.53}$$

To check for convexity: f is convex at x_{cand} if the Hessian $\nabla^2 f(x_{\text{cand}})$ is positive definite, that is, if

$$\forall \text{eig}\, \nabla^2 f(x_{\text{cand}}) > 0 \tag{1.54}$$

Example 1.5 ***Newton search - scalar*** x

We will use Newton search to find the global optimal solution of the function plotted in Figure 1.2. The function is given in Example 1.1, but is repeated here for convenience:

$$f(x) = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \tag{1.55}$$

where

$$a_4 = 0.00232,\ a_3 = -0.111,\ a_2 = 1.80,\ a_1 = -11.6,\ a_0 = 34.4 \tag{1.56}$$

The Matlab script below implements a Newton search to find the global optimal solution.

Let us at first try the guessed value as

$$x_{\text{guess}} = 12$$

where f is *concave*, see Figure 1.14. The figure also shows the result of the search. The result as shown in Matlab is:

$$f_{\text{max}} = 10.8309$$

at

$$x_{\text{max}} = 11.1624$$

So, the Newton search has – unfortunately – arrived where f is at maximum. This illustrates that Newton's method fails if the search starts where f is concave.

We will now set

$$x_{\text{guess}} = 16$$

where f is *con*vex, see Figure 1.15. The figure also shows the result of the search. The result as shown in Matlab is:

$$f_{\text{min}} = 4.7662$$

at

$$x_{\text{opt}} = 18.7483$$

Now, the Newton search has – correctly – arrived at f_{min}. This illustrates that Newton's method succeeds (only) if the search starts where f is convex.

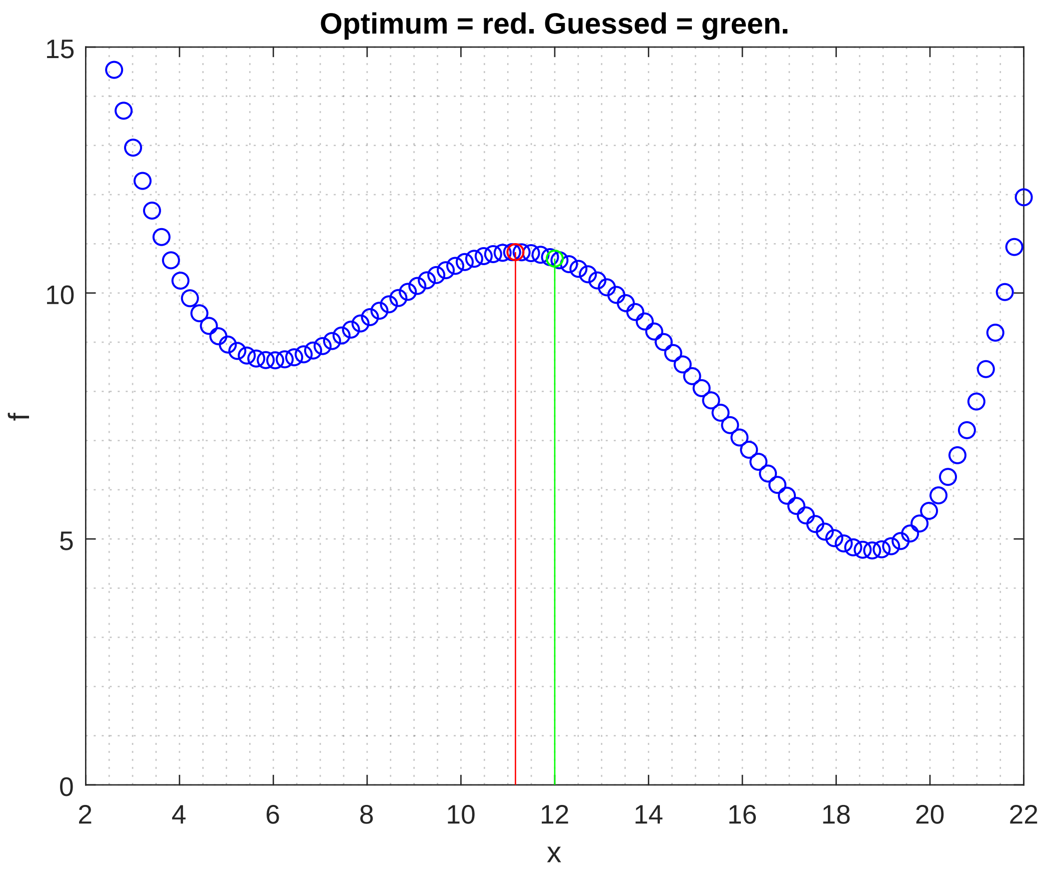


Figure 1.14: Example 1.5: Newton search with $x_{\text{guess}} = 12$. x_{opt} arrives (unfortunately) at $f_{\max}$.

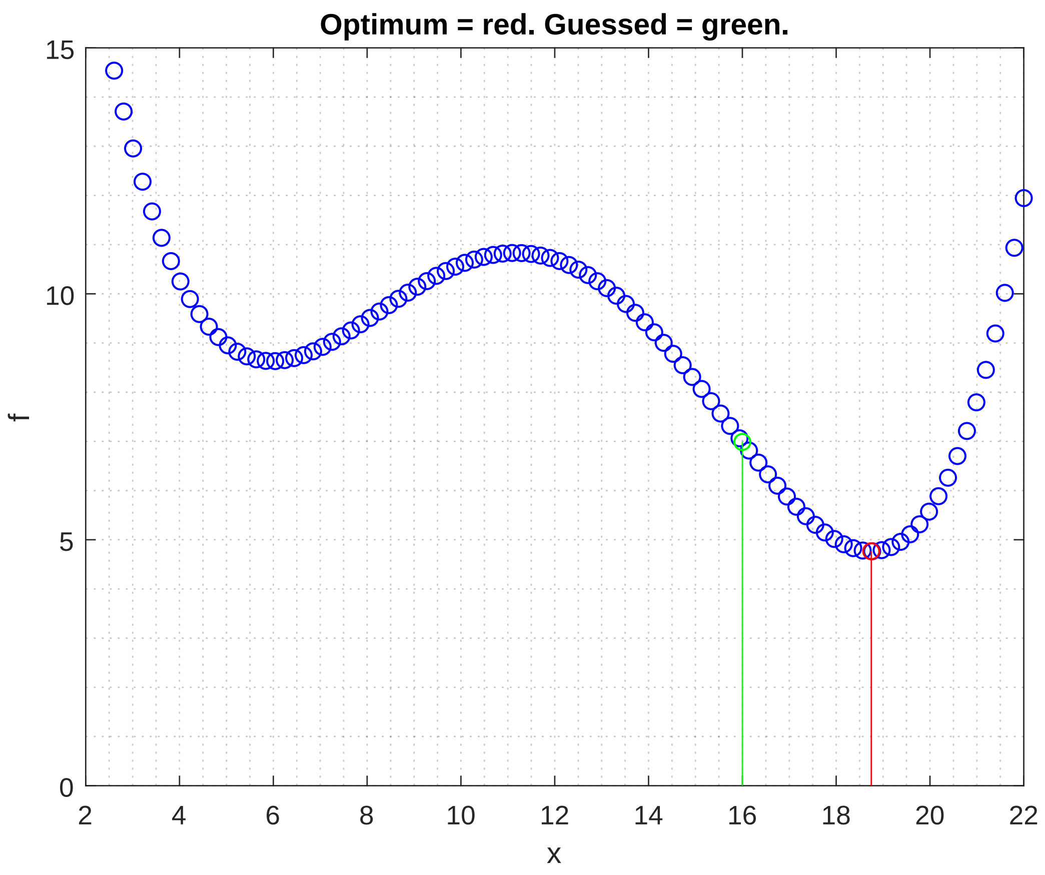


Figure 1.15: Example 1.5: Newton search with $x_{\text{guess}} = 16$. x_{opt} arrives at $f_{\min}$.

Script name: matlab_script_newton_search_scalar.m.

```
clear all, close all, format compact
a4=0.00232; a3=-0.111; a2=1.80; a1=-11.6; a0=34.4;
%Creating anonymous function for the objective function:
f_obj=@(x) a4*x.^4 + a3*x.^3 + a2*x.^2 + a1*x + a0;
%Anonymous function for gradient:
grad=@(x) 4*a4*x.^3 + 3*a3*x.^2 + 2*a2*x + a1;
%Anonymous function for Hessian:
hessian=@(x) 12*a4*x.^2 + 6*a3*x + 2*a2;
x_guess=12;
x_k=x_guess;
N=1000;%Preset max number of iterations
abs_df_spec=1e-4;%Stopping criterion
for k=1:N-1
  dx_k=-inv(hessian(x_k))*grad(x_k);
  %A numerically better alternative to calc dx_k:
  %dx_k=-hessian(x_k)\grad(x_k);
  x_kp1=x_k+dx_k;
  f_k=f_obj(x_k);
  f_kp1=f_obj(x_kp1);
  abs_df=abs(f_kp1-f_k);
  x_k=x_kp1;
  if abs_df < abs_df_spec
    break
  end %if
end %for loop

disp('Result:')
k
x_opt=x_kp1
f_min=f_obj(x_opt)
abs_df
```

[End of Example 1.5]

Example 1.6 ***Newton search - vectorial*** x

We will now make a Newton search to solve the optimization problem already presented in Section 1.2.2:

$$\min_x f(x) \tag{1.57}$$

where

$$f(x) = [x(1) - 1]^2 + [x(2) - 2]^2 + 0.5 \tag{1.58}$$

which is a quadratic objective function.

Let the stop criterion be

$$|f(x_{k+1}) - f(x_k)| \leq \mathrm{d}f = 10^{-4}$$

We follow the procedure given above:

1. A guess:
$$x_0 = x_{\text{guess}} = \left[\begin{array}{c} x(1)_{\text{guess}} \\ x(2)_{\text{guess}} \end{array} \right] = \left[\begin{array}{c} 0 \\ 1 \end{array} \right]$$

2. The first Newton iteration:
$$x_1 = x_0 + \mathit{\Delta} x_0$$
where $\mathit{\Delta} x_0$ for simplicity is calculated directly from Eq, (1.49)
$$\mathit{\Delta} x_0 = -\left[\nabla^2 f(x_0)\right]^{-1} \nabla f(x_0)$$
Here:
$$\nabla f(x_0) = \left[\begin{array}{c} \frac{\partial f(x_0)}{\partial x(1)} \\ \frac{\partial f(x_0)}{\partial x(2)} \end{array} \right] = \left[\begin{array}{c} 2(x(1)_0 - 1) \\ 2(x(2)_0 - 2) \end{array} \right] = \left[\begin{array}{c} -2 \\ -2 \end{array} \right]$$
and
$$\nabla^2 f(x_0) = \left[\begin{array}{cc} \frac{\partial^2 f(x_0)}{\partial x(1)^2} & \frac{\partial^2 f(x_0)}{\partial x(1)\partial x(2)} \\ \frac{\partial^2 f(x_0)}{\partial x(2)\partial x(1)} & \frac{\partial^2 f(x_0)}{\partial x(2)^2} \end{array} \right] = \left[\begin{array}{cc} 2 & 0 \\ 0 & 2 \end{array} \right]$$
giving
$$\mathit{\Delta} x_0 = -\left[\begin{array}{cc} 2 & 0 \\ 0 & 2 \end{array} \right]^{-1} \left[\begin{array}{c} -2 \\ -2 \end{array} \right] = \left[\begin{array}{c} 1 \\ 1 \end{array} \right]$$
So, the first Newton iteration is
$$x_1 = x_0 + \mathit{\Delta} x_0 = \left[\begin{array}{c} 0 \\ 1 \end{array} \right] + \left[\begin{array}{c} 1 \\ 1 \end{array} \right] = \left[\begin{array}{c} 1 \\ 2 \end{array} \right]$$
Checking the stop condition:
$$|f(x_1) - f(x_0)| = \left| f\left(\left[\begin{array}{c} 1 \\ 2 \end{array} \right] \right) - f\left(\left[\begin{array}{c} 0 \\ 1 \end{array} \right] \right) \right| = |0.5 - 2.5| = 2 \leqslant 10^{-4} (\text{No!})$$
The stop condition is not satisfied, so we make the second Newton iteration:
$$x_1 = x_0 + \mathit{\Delta} x_0$$

where

$$\varDelta x_1 = -\left[\nabla^2 f(x_1)\right]^{-1} \nabla f(x_1)$$

where

$$\nabla f(x_1) = \left[\begin{array}{c} 2(x(1)_1 - 1) \\ 2(x(2)_1 - 2) \end{array}\right] = \left[\begin{array}{c} 0 \\ 0 \end{array}\right]$$

and

$$\nabla^2 f(x_1) = \left[\begin{array}{cc} 2 & 0 \\ 0 & 2 \end{array}\right]$$

giving

$$\varDelta x_1 = -\left[\begin{array}{cc} 2 & 0 \\ 0 & 2 \end{array}\right]^{-1} \left[\begin{array}{c} 0 \\ 0 \end{array}\right] = \left[\begin{array}{c} 0 \\ 0 \end{array}\right]$$

Thus, the second Newton iteration is

$$x_2 = x_1 + \varDelta x_1 = \left[\begin{array}{c} 1 \\ 2 \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \end{array}\right] = \left[\begin{array}{c} 1 \\ 2 \end{array}\right]$$

Checking the stop condition:

$$|f(x_2) - f(x_1)| = \left| f\left(\left[\begin{array}{c} 1 \\ 2 \end{array}\right]\right) - f\left(\left[\begin{array}{c} 1 \\ 2 \end{array}\right]\right)\right| = |0.5 - 0.5| = 0 \leqslant 10^{-4}$$

Now, the stop condition is satisfied, so the candidate of the optimal solution is

$$x_{\text{cand}} = x_2 = \left[\begin{array}{c} 1 \\ 2 \end{array}\right]$$

3. Checking for convexity:

$$\operatorname{eig} \nabla^2 f(x_{\text{cand}}) = \operatorname{eig} \left[\begin{array}{cc} 2 & 0 \\ 0 & 2 \end{array}\right] = \left[\begin{array}{c} 2 \\ 2 \end{array}\right]$$

Both eigenvalues are strictly positive, so f is convex at x_{cand}. Consequently, the optimal solution is

$$x_{\text{opt}} = x_{\text{cand}} = x_2 = \left[\begin{array}{c} 1 \\ 2 \end{array}\right]$$

giving

$$f_{\min}(x_{\text{opt}}) = 0.5$$

Actually, the optimum was found in just one Newton iteration. This confirms that the Newton algorithm is exactly a “dead-beat” algorithm for quadratic objective function, as was explained earlier in this section.

A Matlab script implementing the above is shown below.

Script name: matlab_script_newton_search.m.

```
clear all, close all, format compact

%Creating anonymous function for the objective function:
f_obj=@(x) (x(1)-1)^2+(x(2)-2)^2+0.5;

x_k=[0,1]';
%Init N=100; %Preset max number of iterations
abs_df=1e-4;%Stopping criterion
for k=1:N-1
  %Gradient:  G_k=[2*(x_k(1)-1); 2*(x_k(2)-2)];
  %Hessian:  H_k=[2,0; 0,2];
  %Calculation of increment of x (used in x_kp1=x_k+dx_k):
  dx_k=-inv(H_k)*G_k;
  %A numerically better alternative to calc dx_k:
  %dx_k=-H_k\G_k;
  x_kp1=x_k+dx_k;
  f_k=f_obj(x_k);
  f_kp1=f_obj(x_kp1);
  df=f_kp1-f_k;
  x_k=x_kp1;
  if abs(df) < abs_df
    break
  end %if
end %for loop
disp('Result:')
k x_opt=x_kp1
f_min=f_obj(x_opt)
```

The result as shown in Matlab is:

```
k = 2
x_opt = 1 2
f_min = 0.5000
```

[End of Example 1.6]

1.2.5 Combining steepest descent with Newton for robust search

The steepest descent method, cf. Section 1.2.3, has these typical benefits and drawbacks:

- Benefit: It always moves towards a minimum of f, even if f is concave ($f'' > 0$) at the starting point of the search.
- Drawback: It may behave poorly as the search approaches the minimum, with oscillations and/or slow convergence.

The Newton's method, cf. Section 1.2.4, has these typical benefits and drawbacks:

- Benefit: It converges fast and accurately to the minimum when f is close to the minimum.
- Drawback: It moves towards a maximum (away from the desired minimum) if f is concave ($f'' < 0$) at the starting point of the search.

By combining the steepest descent method with Newton's method, the benefits of both methods are retained while the drawbacks are omitted. Theis is summarized in the following.[3]

Combined steepest descent search and Newton search:

1. Start the search with the steepest descent method:

$$x_{k+1} = x_k - \nabla f(x_k)$$

which should eventually bring f into a region where it is convex (if it is not already convex from the start of the search).

2. When f has become convex, continue the search with Newton's method to arrive quickly and smoothly at $f_{\min}$:

$$x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k)$$

To check for f being convex, calculate the eigenvalues of the Hessian, $\nabla^2 f(x_k)$. If all of these eigenvalues have strictly positive real parts, the Hessian is positive definite, and f is convex.

Example 1.7 ***Combined steepest descend search and Newton search***

[3]This is one of two strategies to overcome the problem about the Newton method in a concave region suggested in e.g. https://courses.maths.ox.ac.uk/node/view_material/18818. The second strategy suggested is to reverse the sign of the Newton step, from minus to plus, when f is concave.

The combined method is here applied to the optimization problem presented in Example 1.3. x_{guess} is set as 12, which is an a concave region of f. The Matlab script below shows an implementation of the method.

Script name: matlab_script_combined_steepest_and_newton_search.m.

```
clear all, close all, format compact
a4=0.00232; a3=-0.111; a2=1.80; a1=-11.6; a0=34.4;
%Creating anonymous function for the objective function:
f_obj=@(x) a4*x.^4 + a3*x.^3 + a2*x.^2 + a1*x + a0;
%Anonymous function for gradient:
grad=@(x) 4*a4*x.^3 + 3*a3*x.^2 + 2*a2*x + a1;
%Anonymous function for Hessian:
hessian=@(x) 12*a4*x.^2 + 6*a3*x + 2*a2;
x_guess=12;
x_k=x_guess;
N=1000;%Preset max number of iterations
abs_df_spec=1e-4;%Stopping criterion
for k=1:N-1
   if sum((eig(hessian(x_k)))<=0)>0,
      %convex=0
      dx_k=-grad(x_k)
   else
      %convex=1
      dx_k=-inv(hessian(x_k))*grad(x_k);
      %A numerically better alternative to calc dx_k:
      %dx_k=-hessian(x_k)\grad(x_k);
   end %if
   x_kp1=x_k+dx_k;
   f_k=f_obj(x_k);
   f_kp1=f_obj(x_kp1);
   abs_df=abs(f_kp1-f_k);
   x_k=x_kp1;
   if abs_df < abs_df_spec
      break
   end %if
end %for loop

disp('Result:')
k
x_opt=x_kp1
f_min=f_obj(x_opt)
abs_df
```

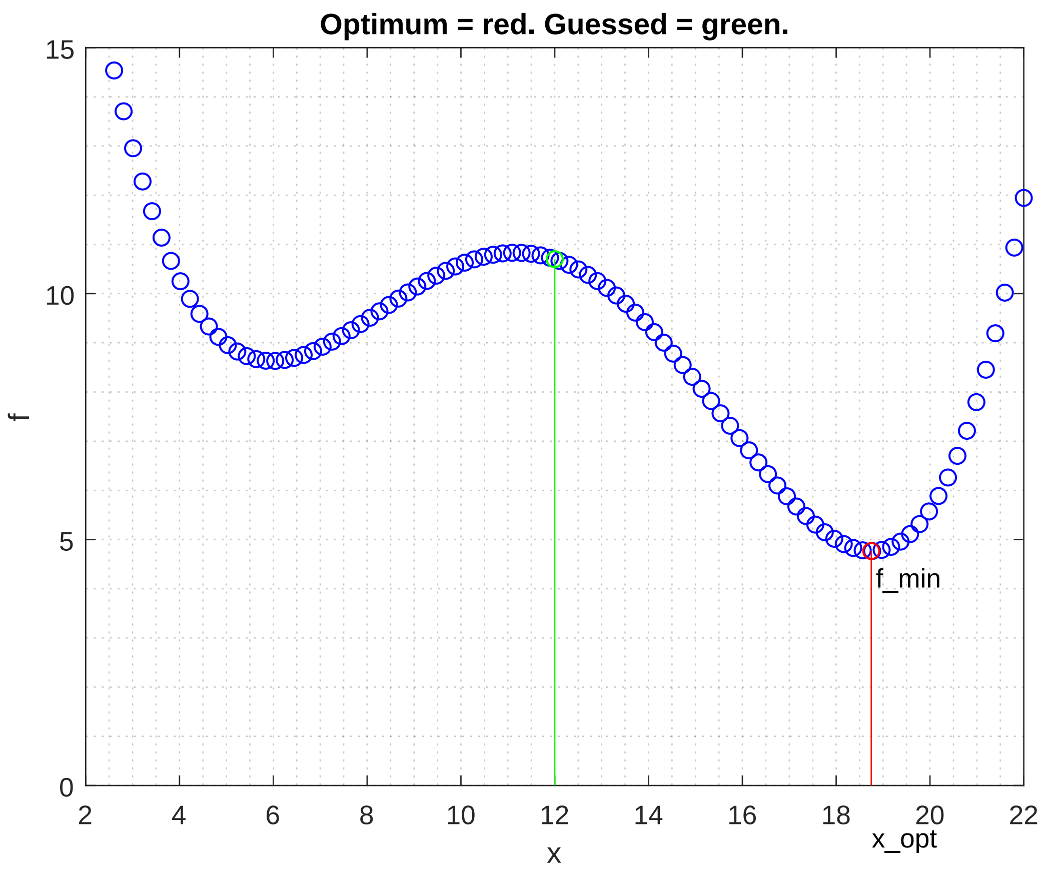


Figure 1.16: Example 1.7: Steepest descent search with $x_{\text{guess}} = 12$. x_{opt} is at global minimum.

The result as shown in Matlab, is:

```
k = 12
x_opt = 18.7483
f_min = 4.7662
abs_df = 1.7086e-08
```

With the steepest descent search in Example, `abs_df = 1.7186e-05`. So, the combined method gives here a more accurate final result which illustrates the benefit of using Newton search in the final stage (that is, after f has become convex) of the search.

Figure 1.16 illustrates the result.

[End of Example 1.7]

1.2.6 Two professional NLP optimizers: fmincon and sqp

1.2.6.1 Introduction

In many applications, you need more powerful and flexible optimizers than one you can implement yourself with e.g. the grid search method and/or the Newton search method. Many optimization problems can be solved with Nonlinear Programming (NLP) optimizers, for example, parameter estimation of nonlinear models, model-predictive control, etc.[4] In these optimizers, constraints on the form of (1.3) and (1.2) are, in principle, included as "penalty" terms in a modified objective function [Edgar *et al.*, 2001]:

$$L(x,\lambda,u) = f(x) + \sum_{j=1}^{r} u_j g_j(x) + \sum_{i=1}^{m} \lambda_i h_i(x) \tag{1.59}$$

where L is denoted the *Lagrangian*, and λ_i and u_j are the Lagrange multipliers of the inequality and equality functions, respectively. The modified optimization problem is solved from $\nabla_x L = 0$.

In the following sections, the practical use of the following two professional Nonlinear Programming (NLP) optimizers are presented:

- fmincon in Matlab's Optimization Toolbox. (fmincon = "finds a constrained minimum of a function of several variables", cf. Matlab documentation.)
- sqp in Octave. (Octave[5] is free Matlab clone. sqp = "sequential or successive quadratic programming".)

Complete examples with Matlab and Octave code are given at the end of the respective sections.

1.2.6.2 fmincon (Matlab)

The information below is compiled from the documentation about fmincon in Matlab.[6]

[4] "Nonlinear programming" is a traditional term used on optimization algorithms for solving nonlinear optimization problems.

[5] http://octave.org

[6] Where appropriate, I have modified the nomenclature.

fmincon attempts to solve optimization problems on the form of

$$\min_x f(x)$$

subject to:

- Linear constraints[7]:
$$A \cdot x \leq B$$
$$Aeq \cdot x = Beq$$
- Nonlinear inequality (less-than-or-equal) constraints:
$$g_{\text{leq}}(x) \leq 0$$
- Nonlinear equality constraints:
$$h_{\text{eq}}(x) = 0$$
- Bounds on the optimization variables:
$$lb \leq x \leq ub$$

fmincon offers a number of alternative algorithms: interior point, sqp, active set (default), and trust region reflective. You can choose one different from the default selection via the option input parameter, see below.

There are several optional input arguments and output arguments of fmincon, and they can be omitted in the function call.[8] Below is the function call with a complete set of arguments:

```
[x,fval,exitflag,output,lambda,grad,hessian] = ...
    fmincon(fun,x_guess,A,B,Aeq,Beq,lb,ub,nonlcon,options)
```

The input arguments of fmincon are:

- fun, a user-defined function with x as input argument and the scalar value of the objective function, $f(x)$, as output. You can pass any model parameters to fun while fun is invoked by fmincon by using an anonymous function call in fmincon. This is demonstrated in the example below.

[7]Alternatively, these linear constraints may be defined with the functions g_{leq} and h_{eq}.

[8]Personally, I use the complete list of arguments, but typically setting some of the arguments as empty, that is, '[]'.

- x_guess, your guessed value of x_{opt} (the optimal solution).
- A, the matrix in the linear equality constraints.
- B, the vector in the linear equality constraints.
- Aeq, the matrix in the linear inequality constraints.
- Beq, the vector in the linear equality constraints.
- lb, the lower bound on x. For example, in the case of three optimization variables: lb = [x1_lb, x2_lb, x3_lb] (assuming numerical values of x1_lb etc. are already set). Inf (infinity) can be used as a value of a bound.
- ub, the upper bound on x.
- nonlcon, a user-defined function with x as input argument and vectors g_leq and h_eq as output vectors calculated (in the function) as the left-hand part of the nonlinear inequalities and equalities, respectively. fmincon calculates the minimum such that each of the elements of the vector g_leq is ≤ 0, and similarly h_eq $= 0$. If no bounds exists, set g_leq = [ ] and/or h_eq = [ ].)
  You can pass any model parameters to nonlcon by using an anonymous function call in fmincon. This is demonstrated in the example below.
- options: Various options can be set as pairs of properties and values. Example: To set the solver algorithm to sqp and the maximum number of iterations to 500:
  options =
  optimoptions(@fmincon,'Algorithm','sqp','MaxIterations',500);
  The default settings of the above two properties corresponds to:
  options = optimoptions(@fmincon,'Algorithm','interior-point','MaxIterations',1000);
  Default settings apply if the options input argument is omitted in the fmincon call, or if options is set as:
  options = optimoptions(@fmincon);

The output arguments of fmincon are:

- x = x_{opt} (the optimal solution).
- fval = $f(x_{\text{opt}})$.
- exitflag, which is an integer expressing various exit conditions.

- output which is a Matlab struct with information about the number of iterations, the number of function calls, etc.
- lambda, which are the Langrange multipliers at x_{opt}.
- grad = $\nabla f(x_{\text{opt}})$, which is a vector of zeros at (the exact) optimum.
- hessian = $\nabla^2 f(x_{\text{opt}})$, which is positive definite matrix at optimum.

Example 1.8 ***fmincon (Matlab)***

We will see how fmincon can be used to solve the following optimization problem, which is the same problem that we solved with the grid search method in Section 1.2.2.

$$\min_x f(x) \tag{1.60}$$

where

$$f(x) = (x_1 - p_1)^2 + (x_2 - p_2)^2 + p_3 \tag{1.61}$$

where $p_1 = 1$, $p_2 = 2$, $p_3 = 0.5$ are model parameters.

Inequality (less-than-or-equal) constraint:

$$g_{\text{leq}}(x) = x(1) - x(2) + 1.5 \leq 0 \tag{1.62}$$

Bounds on the optimization variables:

$$0 \leq x_1 \leq 2 \tag{1.63}$$

$$1 \leq x_2 \leq 3 \tag{1.64}$$

The guess of the optimal solution is

$$x_{\text{guess}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Below is a Matlab script that finds the optimum. Note: The objective function and the constraints function are defined as local functions within the script, and they exist only within the script. Local functions are supported from Matlab version R2016b.

Script name: script_fmincon_matlab.m.

```
clear all;
format compact;
```

```
%Model params:
p1 = 1;
p2 = 2;
p3 = 0.5;
params_model.p1 = p1;%params_model is a struct.  p1 is a field.
params_model.p2 = p2;
params_model.p3 = p3;

%Inputs to fmincon:
x_guess = [0,1]';
Aineq = [];
Bineq = [];
Aeq = [];
Beq = [];
x_lb = [0,2]';
x_ub = [1,3]';

fun_objective_handle = @(x)fun_objective(x,params_model);
  %fun_objective is def as a local function at end of script.
fun_constraints_handle = @(x)fun_constraints(x,params_model);
  %fun_constraints is def as a local function at end of
script.
my_optim_options = optimoptions('fmincon');
options.display = 'off';
%options.algorithm = 'sqp';

%Executing fmincon:
[x_opt,fval,exitflag,output,lambda,grad,hessian] =...

fmincon(fun_objective_handle,x_guess,Aineq,Bineq,Aeq,Beq,x_lb,x_ub,...
  fun_constraints_handle,my_optim_options);

%Results:
disp('Results:')
fval
x_opt

%Defining local functions:

function f = fun_objective(x,params_model)
p1 = params_model.p1;
p2 = params_model.p2;
p3 = params_model.p3;
f = (x(1)-p1)^2+(x(2)-p2)^2+p3;
```

```
end %function

function [g_leq,h_eq] = fun_constraints(x,params_model)
p1 = params_model.p1;
p2 = params_model.p2;
p3 = params_model.p3;
g_leq = x(1)-x(2)+1.5; %Left side of "less than or equal"
nonlin in
h_eq = []; %Left side of nonlinear equalities.
end %function
```

The result shown in Matlab is:

```
Local minimum found that satisfies the constraints.

Optimization completed because the objective function is
non-decreasing in feasible directions, to within the default
value of the optimality tolerance, and constraints are
satisfied to within the default value of the constraint
tolerance.

<stopping criteria details>

Optimal solution:
fval =
  0.6250
x_opt =
  0.7500
  2.2500
```

which is very similar to the result as found with the grid search method in Section 1.2.2. (With a finer grid in the grid method, the grid solution will become even closer to the fmincon solution.)

[End of Example 1.8]

1.2.6.3 sqp (Octave)

The information below is compiled from Octave's documentation about the sqp function.[9]

The sqp function implements an NLP (Nonlinear Programming) optimizer using a sequential (or successive) quadratic programming algorithm.

[9]Where appropriate, I have modified the nomenclature.

The optimization problem to be solved is:

$$\min_x f(x)$$

subject to:

- Nonlinear inequality (greater-than-or-equal) constraints:[10]

  $$g_{\text{geq}}(x) \geq 0$$

- Nonlinear equality constraints:

  $$h_{\text{eq}}(x) = 0$$

- Bounds on the optimization variables:

  $$lb \leq x \leq ub$$

Example 1.9 ***sqp (Octave)***

Below is an Octave script that solves the same optimization problem as was solved with Matlab's fmincon optimizer in Example 1.8. In the script, the sqp function invokes three functions, namely fun_objective, fun_constr_h_eq, and fun_constr_g_g These functions are defined in separate function files (m-files).[11] The function files are shown below the script.

Script name: octave_script_sqp.m.

```
clear all, close all, format compact

% Model params:
p1 = 1;
p2 = 2;
p3 = 0.5;
params_model.p1 = p1;%params_model is a struct.  p1 is a field.
params_model.p2 = p2;
params_model.p3 = p3;

%Inputs to sqp:
```

[10]Note that in the sqp function, the nonlinear inequality is a greater-than-or-equal inequality, while in fmincon, the nonlinear inequality is a less-than-or-equal inequality.

[11]Local functions like in Matlab are not supported in Octave.

```
x_guess = [0,1]';

fun_objective_handle = @(x)fun_objective(x,params_model);
fun_constr_h_eq_handle = @(x)fun_constr_h_eq(x,params_model);
fun_constr_g_geq_handle = @(x)fun_constr_g_geq(x,params_model);

x_lb = [0,2]';
x_ub = [1,3]';

%Executing sqp:
[x_opt,fval,info,output,lambda] =...
   sqp(x_guess,fun_objective_handle,fun_constr_h_eq_handle,...
   fun_constr_g_geq_handle,x_lb,x_ub);

%Results:
fval
x_opt
```

Contents of function files (m-files):

fun_objective.m:

```
function f = fun_objective(x,params_model)
p1 = params_model.p1;
p2 = params_model.p2;
p3 = params_model.p3;
f = (x(1)-p1)^2+(x(2)-p2)^2+p3;
endfunction
```

fun_constr_h_m:

```
function h_eq = fun_constr_h_eq(x,params_model)
h_eq = [];
endfunction
```

fun_constr_g_gm:

```
function g_geq = fun_constr_g_geq(x,params_model)
p1 = params_model.p1;
p2 = params_model.p2;
p3 = params_model.p3;
g_geq = -x(1)+x(2)-1.5; %Left side of nonlinear in:  g_leq >=
0.
endfunction
```

The result of running the script script_sqp_octave.m is:

```
fval = 0.62500
x_opt =
   0.75000
   2.25000
```

which is the same as with fmincon in Matlab.

[End of Example 1.9]

1.2.7 Global optimization

The grid search method with a sufficiently small resolution (search step) is a global optimizer, but the method may arrive at an inaccurate value of the global optimum. The Newton search method will arrive at an accurate value of the local optimum. These two methods can be combined into an accurate global optimizer as follows:

1. Do a grid search. The optimal solution from this search is here denoted x_{grid}.
2. Use the optimum from the grid search as the optimal guess in the Newton search, that is,
$$x_{\text{guess}} = x_{\text{grid}}$$

As an alternative to the second item above (a Newton search), a new grid search can be made around the solution from the first grid search and with a (much) smaller resolution. For example, the range of $x(1)$ can be set to

$$[x(1)_{\text{grid}} - \Delta x(1),\ x(1)x_{\text{grid}} + \Delta x(1)]$$

where $x(1)_{\text{grid}}$ is the solution for x_1 from the first grid search, and $\Delta x(1)$ is the resolution. The same applies to $x(2)$.

Dedicated global optimization methods exist, for example the so-called genetic algorithms, cf. the Global Optimization Toolbox for Matlab, and/or [Edgar *et al.*, 2001].

1.3 Some applications of optimization

1.3.1 Introduction

Optimization methods can be used to solve different kinds of problems, as:

- Controller tuning
- Parameter estimation
- State estimation
- Model-based control
- Process optimization

The clue is, of course, to state the problem as an optimization – typically a minimization – problem,

$$\min_x f(x)$$

and then solve this problem with one of the methods described in Section 1.1, or with other methods. Some typical problems of the kinds listed above are presented in the following sections.

1.3.2 PI controller tuning

To appear.

1.3.3 Parameter estimation of static and dynamic models using nonlinear optimization

You can use optimization methods for estimation of parameters in a mathematical model. The model can be static or dynamic. Dynamic models may be in the form of:

- Differential equations (continuous-time models), linear or nonlinear, possibly in the form of a state space model
- Difference equations (discrete-time models), linear or nonlinear, possibly in the form of a state space model
- Transfer functions - Laplace-transform based (continuous-time transfer functions) or Z-transform based (discrete-time transfer functions)

The formulation of the parameter estimation problem as an optimization problem can be done as follows:

- The objective function to be minimized is the sum of squared prediction errors:
$$\text{SSPE} = \sum_{k=1}^{N} e(k)^2 \tag{1.65}$$
where e is the prediction error which is the difference between the (real, observed) measurements and the model-based predicted or calculated measurements (there are N measurement samples):
$$e = y_{\text{meas}} - y_{\text{pred}} \tag{1.66}$$
y_{pred} is calculated in *simulations*, using the model. Therefore, y_{pred} is a function of the parameters to be estimated, and SSPE is a also a function of the parameters.
- The parameters to be estimated are used as optimization variables. All the parameters may be collected in a parameter vector:
$$P = [p(1),\, p(2), ...,\, p(r)]^T \tag{1.67}$$
- In each iteration, the optimizer runs a simulation with parameter values that are adjusted based on previous iterations (simulations). The iterations stops when when the parameter values that minimizes the SSPE, are found. Those values are the ultimate, estimated parameter values.

Mathematically, the optimization problem can be stated as:

$$\min_{P=[p(1),p(2),...,p(r)]} \text{SSPE}$$

s.t. (subject to) the given mathematical model.

Figure 1.17 illustrates the principle of optimization-based parameter estimation.

Any nonlinear optimizer can be used to implement the parameter estimation. In the following example, the fmincon optimizer in Matlab is used.

Model validation. Roughly said, even a model with wrong structure – i.e. a poorly structured model – can be fitted to given data in a least square sense (mimimizing SSPE). How can you check if the adapted model is actually representing the real system well, or accurately? Such a check is denoted *model validation.* It may be done as follows. Divide the original data series in two parts, typically of equal sizes:

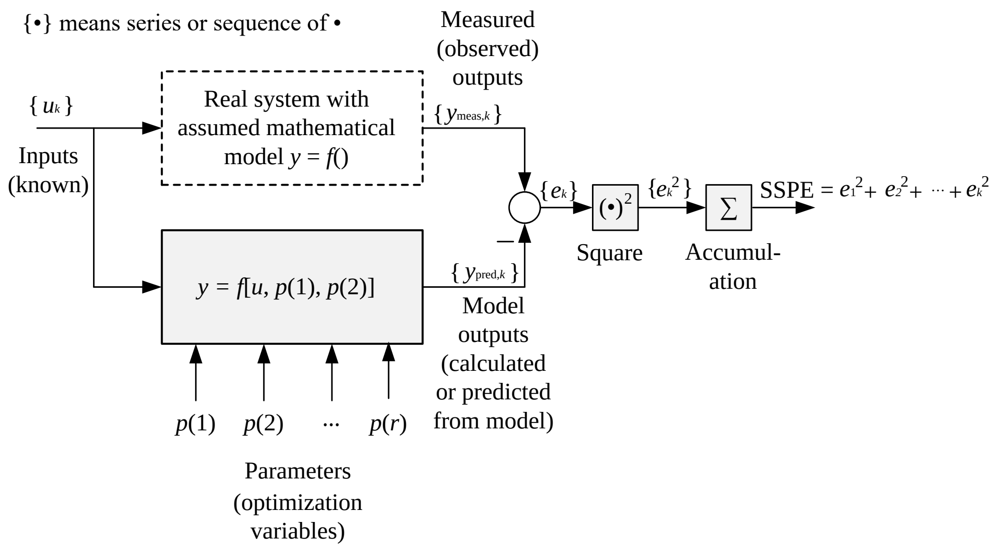

{•} means series or sequence of •
Measured (observed) outputs
$\{u_k\}$
Inputs (known)
Real system with assumed mathematical model $y = f()$
$\{y_{\text{meas},k}\}$
$\{e_k\}$
$(\bullet)^2$
$\{e_k^2\}$
Σ
$\text{SSPE} = e_1^2 + e_2^2 + \cdots + e_k^2$
–
Square
Accumulation
$y = f[u, p(1), p(2)]$
$\{y_{\text{pred},k}\}$
Model outputs (calculated or predicted from model)
$p(1)$ $p(2)$ ··· $p(r)$
Parameters (optimization variables)

Figure 1.17: The principle of parameter estimation using optimization. The ultimate (estimated) parameter values are those that minimize SSPE. (SSPE = sum of squared prediction errors.)

- *Model adaptation part*: This part is used for the parameter estimation.
- *Model validation part*: This part is used for checking if the model is valid or accurate.

So, do not use the data used for model adaptation also for model validation. A new, different data set must be used for the validation.

The model validation can be realized by comparing a simulated response with the real response. In that simulation, the model with the estimated parameters is used, and the simulation is driven by the input data series in the validation data set. If there is only one model (only one model candidate), the model validation is just a visual check to convince yourself that the model reasonably well fits the real motor, as represented by the given data series.

In some applications there are several candidates of models, for example, a first order differential equation and a second order differential equation that are both assumed to represent the real system. You can select the best model as the one that minimizes a numerical measure (and not just a visual check). A typical measure to be minimized is the SSPE index, cf. Eq. (1.65), based on the validation data set. Then, the ultimate model is the one with the smallest SSPE.

Recursive (online) parameter estimation. The parameter estimation described in this section is a *batch* implementation since it operates on the whole data series available. An alternative term is *full information estimation.* If the parameters may change continuously, you may consider *online parameter estimation* where the estimates are updated continuously based on the most recent process measurement available.

Two methods for online parameter estimation are:

- Moving horizon estimation (MHE) which is presented in Section 1.3.4.
- Kalman filtering which is presented in Chapter ... (ref to be added).

In both methods, the parameters are estimated as state variables. The original state vector is augmented with parameter states. Therefore, the Kalman filter used for parameter estimation is denoted *augmented* Kalman filter, and the MHE may similarly be denoted augmented MHE.

Among those two alternatives, I generally recommend Kalman filtering since it is easier to implement, executes faster, and the performance is typically comparable with that of a successful implementation of MHE.[12]

Example 1.10 ***Parameter estimation of a DC motor using fmincom (Matlab)***

In this example, two parameters of a mathematical model of a real DC motor, see Figure 1.18, are estimated (with batch estimation) with the fmincon optimizer in Matlab. There is only one model candidate, so the model validation is in the form of just a visual check.

The control signal, u [V], is the input signal. The rotational speed of the motor is measured with a tachometer, which produces a voltage that is proportional to the speed. The measurement signal, S [V], is regarded as the output signal. A mathematical model of the motor can be developed from electrical and mechanical laws. However, under reasonable assumptions, this model can be approximated with the following time constant model which represents the dynamics of the motor:

$$T\dot{S} = -S + Ku + L \tag{1.68}$$

[12]My own experience is that MHE may actually fail to estimate parameters (expect for simple models) while the Kalman filter works well.

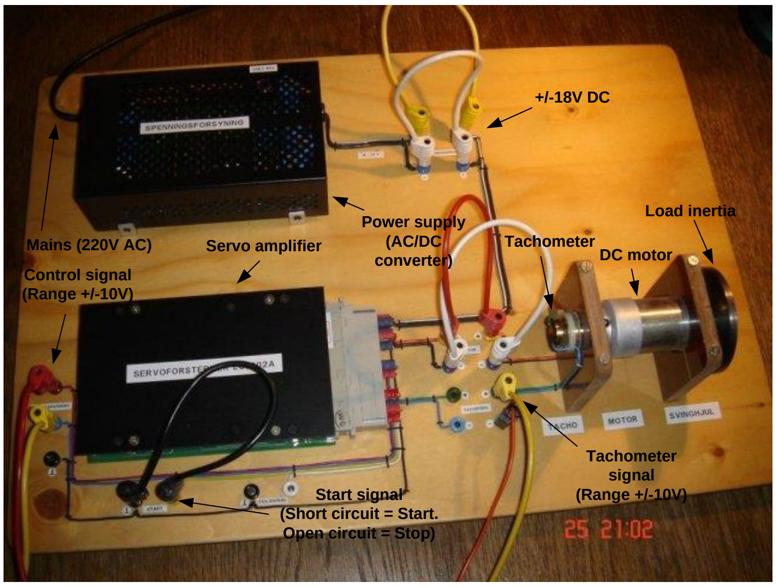


Figure 1.18: DC motor. (http://techteach.no/tekdok/dcmotor)

where K [V/V] is the gain and T [s] is the time constant. K and T are to be estimated from experimental data. L [V] represents the load torque on the motor. In this example it is assumed that L can be neglected.

A series of data of u and S have been recorded from an experiment. These data are available at http://techteach.no/tekdok/dcmotor/.

Below is a Matlab script that implements the parameter estimation of K and T. Comments are included throughout the script.

The original data series is cut in two of equal sizes:

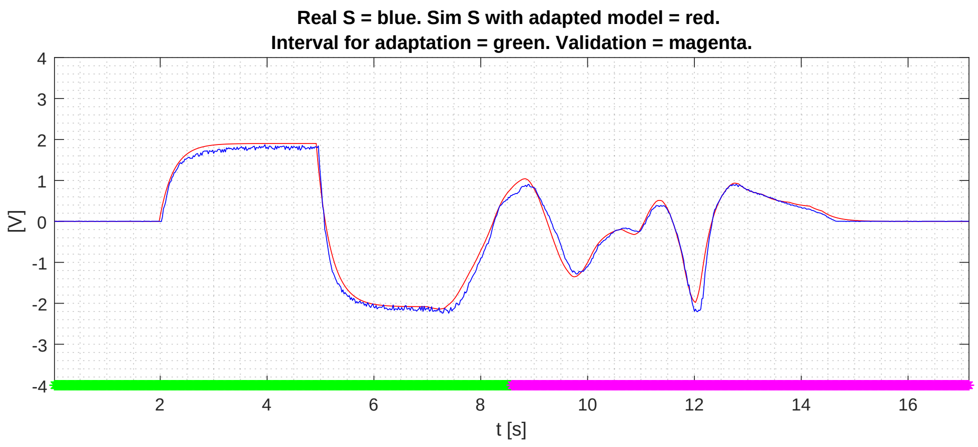
- The first part (first half of the time interval) is used for the parameter estimation, or in more general terms: model adaptation.
- The second part is used for model validation.

The result of the estimation is

$$K_{\text{est}} = 0.865 \tag{1.69}$$

$$T_{\text{est}} = 0.270 \tag{1.70}$$

Figure 1.19 shows the real u and the real S and the simulated S with estimated K and T. The simulation is run with both adaptation (estimation) data and validation data. Since the simulated S fits well with



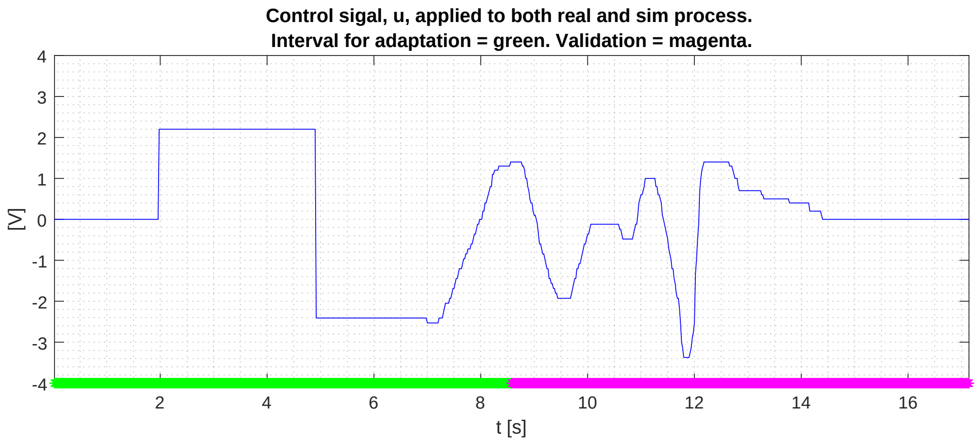


Figure 1.19: Example: 1.10: Real and simulated data with estimated model parameters.

the real S from the validation data, we can conclude that the model is valid (accurate).

Script name: script_dc_motor_model_adapt_fmincon.m.

```
%---------------------------------------------------------

%Parameter estimation of a gain & time-constant model of DC motor

%from experimental data using nonlinear optimation

%with fmincon() in Matlab.

%---------------------------------------------------------
```

```
%Model (gain & time-constant model):

%

%dSdt(t) = (1/T)*[K*u(t)-S(t)]

%

%where:

%S [V] is rotational speed measurement.

%u [V] is control signal.

%K [V/V] is gain (to be estimated).

%T [s] is time-constant (to be estimated).

%--------------------------------------------------

clear all

close all

format compact

commandwindow

%--------------------------------------------------

%Loading logfile, and assigning the data to a matrix:

data_series = load('logfile1.txt');

Ts = 0.02; %Sampling interval

L = length(data_series);

%First fraction (portion) of logfile

%to be used for model adaptation:

F = 0.5;

N = floor(L*F);

%Extracts data for estimation (model adaptation):

t_real_estim = Ts*[1:N]; %Time array

u_real_estim = data_series(1:N,2);
```

```
S_real_estim = data_series(1:N,3);

%Data for model validation:

t_real_valid = Ts*[N+1:L];

u_real_valid = data_series(N+1:L,2);

S_real_valid = data_series(N+1:L,3);

%Total data:

t_real_total = Ts*[1:L];

u_real_total = data_series(1:L,2);

S_real_total = data_series(1:L,3);

%------------------------------------------------

%Upper and lower bounds of optim variables (parameters):

K_max = 2; %[V/V]

T_max = 1; % [s]

K_min = 0.2;

T_min = 0.05;

p_ub = [K_max, T_max];

p_lb = [K_min, T_min];

%------------------------------------------------

%Guessed values of optim vars (params):

p_guess = [0.1, 0.2];

%------------------------------------------------

%Known parameter values (if any):

p_known = [];

%------------------------------------------------

%Initialization of simulation:

S_sim_init = S_real_total(1);
```

```
S_sim_k = S_sim_init;

%-----------------------------------------------

%Preallocation of array used in plotting:

S_sim_plot_array = t_real_total*0;

%-----------------------------------------------

%Optimization (calculating optim param values) with fmincon:

%Linear inequalities and equalities,

%set empty since not used in this application:

Aineq = []; Bineq = []; Aeq = []; Beq = [];

%Defining function handle:

fun_objective_handle = ...

@(p)fun_objective(p,p_known,t_real_estim,...

u_real_estim,S_real_estim,S_sim_init,Ts);

%Defining function handle:

fun_constraints_handle = ...

@(p)fun_constraints(p,p_known,t_real_estim,...

u_real_estim,S_real_estim,S_sim_init,Ts);

my_optimoptions = optimoptions(@fmincon); %Using default options

%my_optimoptions = optimoptions(@fmincon,'Algorithm','sqp');

[p_opt,fval,exitflag,output,lambda,grad,hessian] = ...

fmincon(fun_objective_handle,p_guess,Aineq,Bineq,Aeq,Beq,...

p_lb,p_ub,fun_constraints_handle,my_optimoptions);

%-----------------------------------------------

%Displaying the optimal solution:

disp('-------------------')

disp('Optimal parameter estimates:')
```

```
K_estim = p_opt(1)

T_estim = p_opt(2)

%-------------------------------------------------

%Simulation with adapted model over whole interval,

%and comparing with real data.

%Using estiated param values:

K = K_estim;

T = T_estim;

%Initialization:

S_sim_init = S_real_total(1);

S_sim_k = S_sim_init;

%Simulation loop:

for k_sim = 1:length(t_real_total)

%Integration with Euler forward:

dS_sim_dt_k = (1/T)*(-S_sim_k + K*u_real_total(k_sim));

S_sim_kp1 = S_sim_k + Ts*dS_sim_dt_k;

%Updating array used in plotting:

S_sim_plot_array(k_sim) = S_sim_k;

%Time shift:

S_sim_k = S_sim_kp1;

end

%-------------------------------------------------

%Plotting:

h = figure; %Getting figure handle

fig_posleft = 8;fig_posbottom = 1.5;fig_width = 24;fig_height = 20;

fig_pos_size_1 = [fig_posleft,fig_posbottom,fig_width,fig_height];
```

```
set(gcf,'Units','centimeters','Position',fig_pos_size_1);

figtext = 'Estimation of model params of DC motor';

set(gcf,'Name',figtext,'NumberTitle','on')

figure(1)

subplot(2,1,1)

plot(t_real_total,S_sim_plot_array,'r',t_real_total,S_real_total,'b',...

t_real_estim,t_real_estim*0-4,'g*',...

t_real_valid,t_real_valid*0-4,'m*')

axis([t_real_total(1),t_real_total(end),-4,4])

grid minor

title({'Real S = blue.  Sim S with adapted model = red.  ';...

'Interval for adaptation = green.  Validation = magenta.'})

xlabel('t [s]')

ylabel('[V]')

subplot(2,1,2)

plot(t_real_total,u_real_total,'b',...

t_real_estim,t_real_estim*0-4,'g*',...

t_real_valid,t_real_valid*0-4,'m*')

axis([t_real_total(1),t_real_total(end),-4,4])

grid minor

title({'Control sigal, u, applied to both real and sim process.
';...

'Interval for adaptation = green.  Validation = magenta.'})

xlabel('t [s]')

ylabel('[V]')

%-----------------------------------------------
```

```
%Saving plot figure as pdf file (for inclusion in document):

saveas(h,'example_dcmotor_param_estim','pdf')

%------------------------------------------------

%Defining local functions used by fmincon:

%(local functions must be defined at the end of the script)

%------------------------------------------------

function f = fun_objective(p,p_known,t_real_estim,...

u_real_estim,S_real_estim,S_sim_init,Ts)

%Parameters (optim variables) used by fmincon

K = p(1);

T = p(2);

%Initialization:

S_sim_k = S_sim_init;

sspe_km1 = 0;

%Simulation loop:

for k_sim = 1:length(t_real_estim)

%Integration with Euler forward:

dS_sim_dt_k = (1/T)*(-S_sim_k + K*u_real_estim(k_sim));

S_sim_kp1 = S_sim_k + Ts*dS_sim_dt_k;

%Updating sspe:

d_sspe_k = (S_real_estim(k_sim) - S_sim_k)^2;

sspe_k = sspe_km1 + d_sspe_k;

%Time shift in sim loop:

S_sim_k = S_sim_kp1;

sspe_km1 = sspe_k;

end
```

```
%Objective function (to be minimized):

f = sspe_k;

end

%-----------------------------------------------

%fun_constraints (this function is actually not used):

function [cineq, ceq] = fun_constraints(p,p_known,t_real_estim,...

u_real_estim,y_real_estim,y_sim_init,Ts)

cineq = []; % Compute nonlinear inequalities.

ceq = []; % Compute nonlinear equalities.

end
```

[End of Example 1.10]

1.3.4 Moving horizon estimation

Moving horizon estimation (MHE) [Robertson *et al.*, 1996][13] is a method for state estimation of dynamic systems that can be regarded as an alternative to Kalman Filtering. The principle is to continuously use the present and the previous measurements of the system and known inputs to the system over a fixed-length *historical time horizon*, together with an assumed mathematical model of the system to calculate the optimal (best) present state of the system. This is illustrated in Figure 1.20.

It is assumed that the model of the pertinent system is given as a discrete time state space model on the following form:

$$x_{k+1} = f(x_k, \cdot) + w_k \tag{1.71}$$

$$y_k = g(x_k, \cdot) + v_k \tag{1.72}$$

where:

- k is the discrete time index, so that the actual time is $t_k = kT_s$ where T_s is the time-step or sampling time.

[13]A good presentation of MHE and several other topics in state estimation is given in [Boegli, 2014].

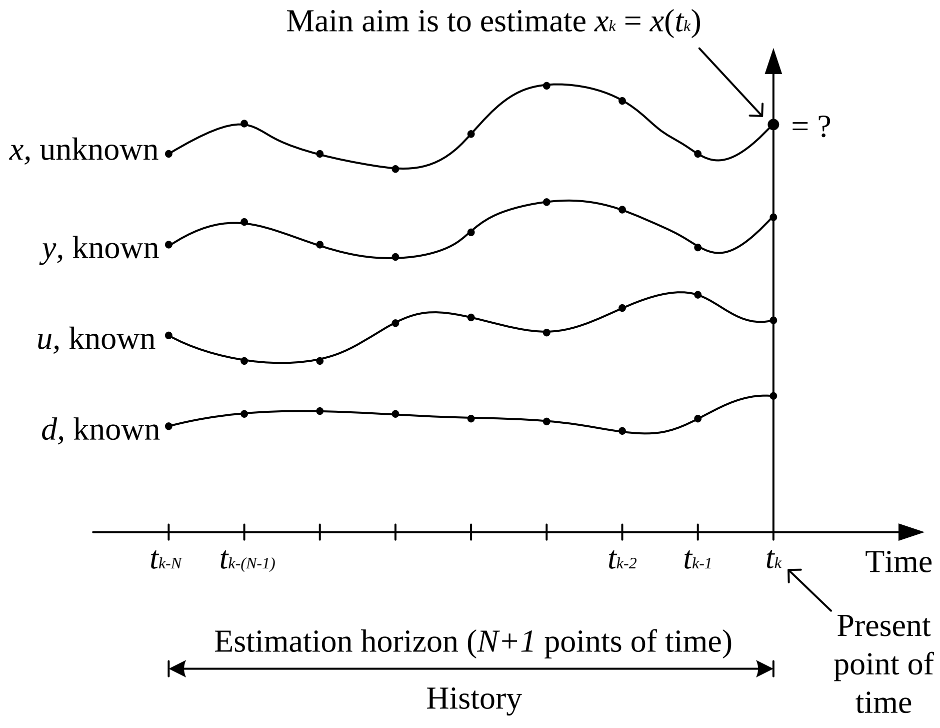


Figure 1.20: Principle of Moving horizon estimation (MHE). x is state. y is measurement. u is control signal. d is disturbance.

- x is the state vector to be estimated. More precisely, it is the value of x at time (index) k that we want to estimate. x is a vector of n scalar state variables:

$$x = \begin{bmatrix} x(1) \\ x(2) \\ \vdots \\ x(n) \end{bmatrix} \tag{1.73}$$

- f is a vectorial function. It is a vector of n nonlinear, or linear, functions of x_k:

$$f = \begin{bmatrix} f_1(x_k, \cdot) \\ f_2(x_k, \cdot) \\ \vdots \\ f_n(x_k, \cdot) \end{bmatrix} \tag{1.74}$$

  There may be additional arguments of f, as the control variable (vector) u_k, the process disturbance (vector), d_k, and parameters (vector), p. These additional arguments are represented by the dots in Eq. (1.71), and we assume they have *known* values. (Estimation of unknown disturbances or parameters is described in Section 1.3.4.)

- w_k represents the *non-modeled or unknown process disturbance*, acting on the state, at time index k. w_k may alternatively be interpreted as a *model error* since w_k represents the error of the prediction of the state at the next time-step. Any known disturbance is represented by d_k, cf. the above item. It is not necessary to assume any paricular statistical properties of w_k, however, it may be reasonable to assume it is a random signal – "white noise" – with a specified covariance matrix, say Q, just as in the Kalman Filter. We may then use Q in the objective function of MHE, as described below.

- y is the system output vector of m elements assumed being measured and therefore having known value.

- g is a vectorial function. It is a vector of m nonlinear, or linear, functions of x and possibly of additional variables and parameters, represented by dots:

$$g = \begin{bmatrix} g_1(x_k, \cdot) \\ g_2(x_k, \cdot) \\ \vdots \\ g_m(x_k, \cdot) \end{bmatrix} \tag{1.75}$$

- v_k is the *measurement error*. It is not necessary to assume any paricular statistical properties of v_k, however, it may be reasonable to assume it is a random signal – "white noise" – with a specified covariance matrix, say R, just as in the Kalman Filter. We may then use R in the objective function of MHE, as described below.

The optimization problem in MHE to be solved continuously, to continuously calculate the state estimate, is:

$$\min_X J \tag{1.76}$$

J is the objective function. It is defined below. X is a matrix containing the state at each point of time of the estimation horizon:

$$\begin{aligned} X &= \left[x_{k-N}, x_{k-(N-1)}, x_{k-1}, x_k\right] \\ &= \begin{bmatrix} x(1)_{k-N} & x(1)_{k-(N-1)} & \cdots & x(1)_{k-1} & x(1)_k \\ x(2)_{k-N} & x(2)_{k-(N-1)} & \cdots & x(2)_{k-1} & x(2)_k \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ x(n)_{k-N} & x(n)_{k-(N-1)} & \cdots & x(n)_{k-1} & x(n)_k \end{bmatrix} \end{aligned} \tag{1.77}$$

X can be denoted the total state matrix. X will be the solution of the MHE optimization problem. From this X,

$$x_k = \begin{bmatrix} x(1)_k \\ x(2)_k \\ \vdots \\ x(n)_k \end{bmatrix}$$

is used as the applied present state estimate. As an example, assume the model has $n = 3$ state variables and the horizon length is 5, the number of optimization variables $n(N+1) = 3 \cdot 6 = 18$.

Before we look at the details of J, the optimization function to be minimized, let's review the mathematical term *norms* since J contain norms.

About norms. A norm is a measure of the length of a vector. There are various kinds of norms, but the most common one is the quadratic norm and variations of such. For a given vector z, the expression $\|z\|_M^2$ is the square of the M-quadratic norm of z.[14] In detail, $\|z\|_M^2$ is

[14]Quadratic norms are also denoted l^2-norms or Euclidian distance.

$$\begin{aligned}
\|z\|_M^2 &= z^T M z \\
&= \begin{bmatrix} z(1), & \cdots, & z(r) \end{bmatrix} \begin{bmatrix} M_{11} & & \\ & \ddots & \\ & & M_{rr} \end{bmatrix} \begin{bmatrix} z(1) \\ \vdots \\ z(r) \end{bmatrix} \\
&= M_{11} z(1)^2 + \cdots + M_{rr} z(r)^2
\end{aligned}$$

If M=I , the identity matrix, the M-quadratic norm is the square of the well-known length of the vector, since the length is
$\sqrt{z^T z} = \sqrt{z(1)^2 + \cdots + z(r)^2}$.

The optimization function. The optimization function to be minimized, cf. Eq. (1.76), is

$$J = \sum_{i=k-N}^{k-1} \|x_{i+1} - f(x_i, \cdot)\|_{Q^{-1}}^2 + \sum_{i=k-N}^{k} \|y_i - g(x_i, \cdot)\|_{R^{-1}}^2 \tag{1.78}$$

Based on Eqs. (1.71) and (1.72), J can also be written as

$$J = \sum_{i=k-N}^{k-1} \|w_i\|_{Q^{-1}}^2 + \sum_{i=k-N}^{k} \|v_i\|_{R^{-1}}^2 \tag{1.79}$$

or

$$J = \sum_{i=k-N}^{k-1} w_i^T Q^{-1} w_i + \sum_{i=k-N}^{k} v_i^T R^{-1} v_i \tag{1.80}$$

where:

The process disturbance vector:

$$w_i = \begin{bmatrix} w(1)_i \\ \vdots \\ w(n)_i \end{bmatrix}$$

The measurement vector:

$$v_i = \begin{bmatrix} v(1)_i \\ \vdots \\ v(m)_i \end{bmatrix}$$

Q^{-1} is a weight matrix:

$$Q^{-1} = \begin{bmatrix} \frac{1}{Q_{11}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{Q_{nn}} \end{bmatrix}$$

Assuming that w is random, Q can be interpreted as the inverse of the process disturbance covariance matrix.

R^{-1} is a weight matrix:

$$R^{-1} = \begin{bmatrix} \frac{1}{R_{11}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{R_{mm}} \end{bmatrix}$$

Assuming that v is random, R can be interpreted as the inverse of the measurement error covariance matrix.

Now, Eq. (1.80) can be written in detail as

$$J = \sum_{i=k-N}^{k-1} \left[\frac{w(1)_i^2}{Q_{11}} + \cdots + \frac{w(n)_i^2}{Q_{nn}} \right] + \sum_{i=k-N}^{k} \left[\frac{v(1)_i^2}{R_{11}} + \cdots + \frac{v(m)_i^2}{R_{mm}} \right]$$

Roughly said, MHE minimizes, in a least squares sense, the measurement errors and the non-measured process disturbances (or model errors) over the estimation horizon. In other words, MHE utilizes *the measurements* (by minimizing the influence of the measurement errors) and *the model* (by minimizing the model errors) to calculate the state estimate.

Typically, the objective function is presented with an additive term denoted the arrival cost[15], a:

$$J = \sum_{i=k-N}^{k-1} \|w_i\|_{Q^{-1}}^2 + \sum_{i=k-N}^{k} \|v_i\|_{R^{-1}}^2 + a \tag{1.81}$$

where

$$a = \|x_{k-N} - \overline{x}_{k-N}\|_{P^{-1}}^2 \tag{1.82}$$

where $\overline{x}_{k-N}$ is the actual state at the start of the horizon, and P^{-1} is the corresponding weight or cost matrix. The actual state is not known, so it is problematic to give it a value. This uncertainty can be expressed with a very large P, causing a to vanish. Furthermore, an estimate of x_{k-N} will actually be provided since x_{k-N} is included in the first term of Eq. (1.78). Consequently, it is probably ok to omit the arrival cost term from the objective function, i.e. a can be set to zero in Eq. (1.81).

[15]Maybe a better name would have been the departure cost?

Constraints. The MHE optimization problem can also include constraints on X (the optimization variables). For example, if the liquid level in a tank is one state variable, it is natural to define the maximum possible level as an upper bound and the minimum possible level as the lower bound on X.

Guessed value of X. When solving the optimization problem, it is necessary that the optimizer is supplied with a good guess of the optimization variable, X. As a good value of X_{guess} at time index k, here denoted $X_{\text{guess}_\text{k}}$, we can use the pertinent portion of $X_{\text{opt}_{k-1}}$, the optimal solution found at time index $k-1$ (the previous point of time). However, in $X_{\text{opt}_{k-1}}$, we must leave out the state $x_{\text{opt}_{k-1}}$ while we insert a guessed value for time index k. Lets us denote the latter guessed state by x_{guess_k}. How to select x_{guess_k}? It is here suggested that a model-predicted value of x based on $x_{\text{opt}_{\text{k}-1}}$ is used to calculate x_{guess_k}, as follows:

$$x_{\text{guess}_k} = x_{\text{pred}_k} = f(x_{\text{opt}_{k-1}}, u_k, d_k)$$

To summarize, X_{guess_k} can be selected as

$$X_{\text{guess}_k} = \left[X_{\text{opt}_{(2:k-1)}}, x_{\text{pred}_k} \right] \tag{1.83}$$

where (2:k-1) is Matlab-like notation.

No linearization. In MHE, no linearization of the state space model is needed. Hence, it is a “nonlinear” state estimation method. This is contrary to Extended Kalman Filtering, which requires linearization of the state space model to obtain the Kalman gain.

Estimation of model parameters and state disturbances. In state estimation, it is often a wish to estimate model parameters and/or process disturbances in addition to the “ordinary” states that stem from the principles of mechanistic modeling, i.e. material balances, energy balances, impulse balances (laws of motion) etc. Such parameters or disturbances can be estimated in a straighforward way with MHE (as with a Kalman Filter) if we model them as state variables. Remember that a state variable is represented by its time derivative, i.e, a differential equation, in a continuous-time state space model. It is common to assume that a parameter or a disturbance to be estimated is constant. What is the time derivative of a constant? Zero! Therefore, a model parameter p can be represented by the differential equation

$$\dot{p} = 0 \tag{1.84}$$

The corresponding difference equation to be included in the discrete time state space model of MHE, is

$$p_{k+1} = p_k + w_p \tag{1.85}$$

where w is model error pertinent to this parameter state variable. The original state space model has now been *augmented* with this difference equation. p is an *augmentation* state variable. The original state vector has been augmented, to become:

$$x_{\text{aug}} = \begin{bmatrix} x_{\text{original}} \\ p \end{bmatrix} \tag{1.86}$$

Similary, a state disturbance can be estimated by augmenting the original model with the following difference equation

$$d_{k+1} = d_k + w_d \tag{1.87}$$

Tuning factors of MHE. The main tuning factors of MHE are:

- *The estimation horizon length, N.* The larger N, the “safer” optimal solution can be expected, but on the expense of more computational demand. A typical value of N seems to be between 5 and 20, assuming an appropriate time step length (which may be e.g. 1/5 of smallest time-constant-like dynamics represented by the model).
- *The covariance matrix, R, of the measurement error.* Increasing the value of R_{jj}, which is the variance of measurement error of variable $y(j)$, implies higher influence of $y(j)$ on the state estimate, but at the expense of measurement noise $v(j)$ also influencing more on the estimate. Although you may use R as a tuning factor, it may be reasonable to fix it to the variance of an appropriate measurement series of the real process measurement. If you do not have any measurement series at hand, you may set it to the square of 1/100 of an assumed value of the measurement (which corresponds to the standard deviation of the measurement error assumed as 1/100 of the measurement value), i.e.

$$R(j,j) = \left[\frac{y(j)_{\text{assumed}}}{100}\right]^2 \tag{1.88}$$

- *The covariance matrix, Q, of the non-modelled process disturbance.* It is not easy to set a proper value of Q, but the following starting value may be selected for $Q(j,j)$ – the covariance of the non-modelled process disturbance acting on state $x(j)$:

$$Q(j,j) = \left[\frac{x(j)_{\text{assumed}}}{1000}\right]^2 \tag{1.89}$$

If you increase $Q(j,j)$, the estimate of $x(j)$ may approach to its "true" value faster, but at the expense of the estimate becoming more noisy. This can be seen from Eq. (1.79): If element $Q(j,j)$ is increased, it is implicitly assumed that the pertinent non-modelled process disturbance w_j is increased, which calls for a more "aggressive" update of the state estimate by the MHE.

Example 1.11 ***Moving Horizon Estimation with fmincon (Matlab)***

Below is a Matlab script that implements MHE for the following continuous-time model (which will be discretized):

$$\begin{aligned}
\dot{x}(1) &= x(2) \\
\dot{x}(2) &= \left[-x(2) + Ku\right]/T + d \\
y &= x(1)
\end{aligned}$$

where K is the gain and T is the time constant. d is a process disturbance which is to be estimated along with $x(1)$ and $x(2)$. The measurement is $y = x(1)$. u is the control signal. Figure 1.21 shows the results of a simulation. The MHE is started when the simulation has run for as long as the length of the estimation horizon, which is here 10 time-steps corresponding to 5 s. We see that the MHE estimates the states, including the disturbance, well.

Below is a Matlab script, including comments, that implements the MHE using the fmincon optimizer. fmincon is described in Section 1.2.6.2.

Notes about the Matlab implementation:

- The optimization variable in the MHE optimization problem is the total state matrix, X, cf. Eq. (1.77). You can represent X with a Matlab *matrix* in fmincon! Thus, it is not necessary to transform this matrix to an array for fmincon.
- The objective function and the constraints function are defined as local functions within the script, and they exist only within the script. Local functions are supported from Matlab version R2016b.

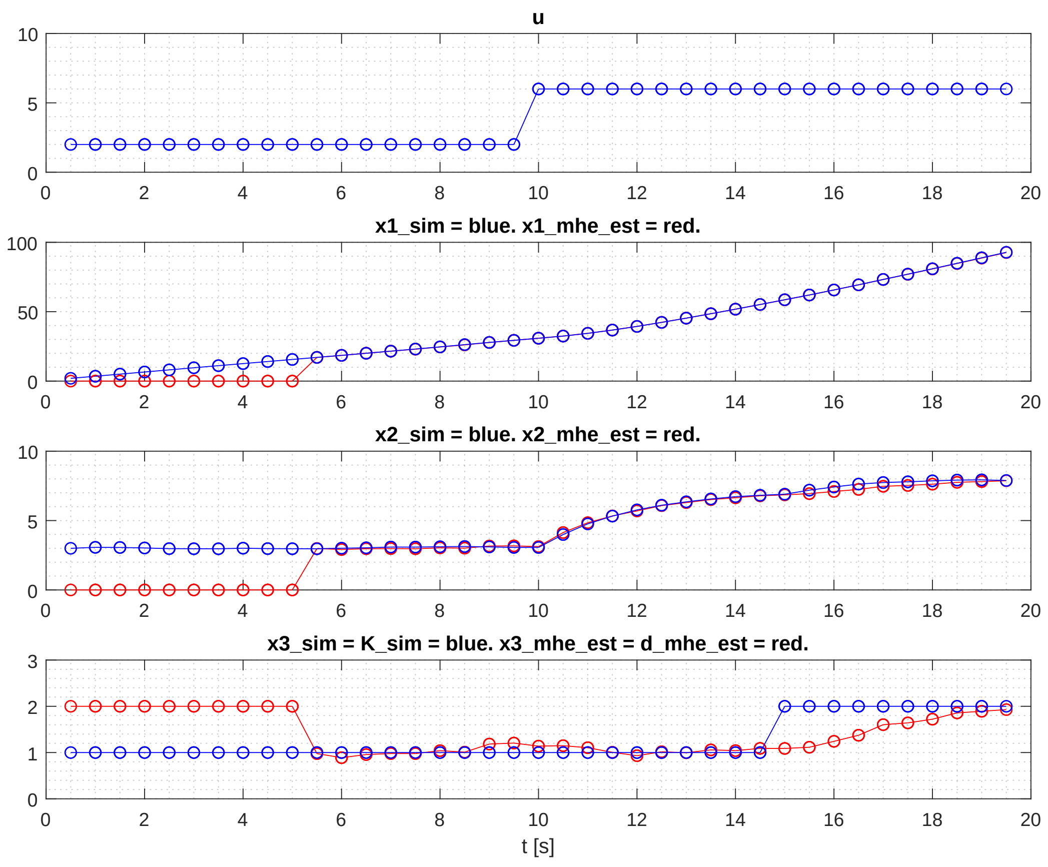


Figure 1.21: Example 1.11: MHE estimation.

Script name: script_mhe_fmincon.m.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Finn Aakre Haugen

%15 04 2018

%MHE with model with 2 state variables and 1 disturbance as
augmented state:

%dx1_dt = x2 + w1

%dx2/dt = (-x2 + K*u + d)/T + w2

%y = x1 + v

%where d = x3 is to be estimated.

%

%Note 1:  Using matrix as optim variable.

%Note 2:  Objective and constraints functions for fmincon are
defined

%as local functions at the end of this script.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('-------------------------------');

disp('Moving Horizon Estimator for:')

disp('dx1_dt=x2; dx2/dt=(-x2+K*u)/T;');

disp('x1, x2, x3 = d are estimated.');

disp('y=x1 is measured.');

disp('');

%-------------------------------------------

close all

clear all

format compact

commandwindow
```

```
%-------------------------------------------

%Model params:

K = 1; %Gain

T = 2;%s Time constant

n = 3;%Number of state variables

model_params.K = K;

model_params.T = T;%model_params is struct.  K is field.

cov_process_disturb_w1 = .001;

cov_process_disturb_w2 = .001;

cov_process_disturb_w3 = .001;

cov_process_disturb_w = ...

diag([cov_process_disturb_w1,cov_process_disturb_w2,...

cov_process_disturb_w3]);

cov_meas_noise_v1 = .01;

cov_meas_noise_v = diag([cov_meas_noise_v1]);

%-------------------------

%Time settings:

Ts = 0.5;%s

t_start = 0;%s

t_stop = 20;%s

t_array = [t_start:Ts:(t_stop-Ts)];%Array for storage

N = length(t_array);

t_mhe = 5

N_mhe = floor(t_mhe/Ts)

number_optim_vars = n*N_mhe

%-----------------------------------
```

```
%Preallocation of arrays for storage:

u_sim_array = t_array*0;

x1_sim_array = t_array*0;

x2_sim_array = t_array*0;

x3_sim_array = t_array*0;

y1_sim_array = t_array*0;

x1_est_optim_plot_array = t_array*0;

x2_est_optim_plot_array = t_array*0;

x3_est_optim_plot_array = t_array*0;

%-----------------------------------

%Sim initialization:

x1_sim_init = 2;

x2_sim_init = 3;

x1_sim_k = x1_sim_init;

x2_sim_k = x2_sim_init;

%-----------------------------------

%MHE initialization:

mhe_array = zeros(1,N_mhe);

x1_est_init_guess = 0;

x2_est_init_guess = 0;

x3_est_init_guess = 2;

x1_est_optim_array = zeros(1,N_mhe) + x1_est_init_guess;

x2_est_optim_array = zeros(1,N_mhe) + x2_est_init_guess;

x3_est_optim_array = zeros(1,N_mhe) + x3_est_init_guess;

x_est_guess_matrix = ...

[x1_est_optim_array;x2_est_optim_array;x3_est_optim_array];
```

```
u_mhe_array = mhe_array*0;

y1_meas_mhe_array = mhe_array*0;

%-----------------------------------

%Figure size etc.:

fig_posleft=8;fig_posbottom=2;fig_width=24;fig_height=18;

fig_pos_size_1=[fig_posleft,fig_posbottom,fig_width,fig_height];

h = figure(1);

set(gcf,'Units','centimeters','Position',fig_pos_size_1);

figtext='Moving Horizon Estimator';

set(gcf,'Name',figtext,'NumberTitle','on')

%-----------------------------------

%Sim loop:

for k = 1:N

t_k = k*Ts;

%-----------------------------

%Process simulator:

if t_k < 2

u_k = 2;

d_k = 1;

end

if t_k >= 2 %Change of u

u_k = 2;

end

if t_k >= 8 %Change of d

d_k = 1;

end
```

```
if t_k >= 10 %Change of u

u_k = 6;

end

if t_k >= 15 %Change of d

d_k = 2;

end

%Derivatives:

dx1_sim_dt_k = x2_sim_k;

dx2_sim_dt_k = (-x2_sim_k + K*u_k + d_k)/T;

f1_sim_k = x1_sim_k + Ts*dx1_sim_dt_k;

f2_sim_k = x2_sim_k + Ts*dx2_sim_dt_k;

%Integration and adding disturbance:

w1_sim_k = sqrt(cov_process_disturb_w1)*randn;

w2_sim_k = sqrt(cov_process_disturb_w2)*randn;

x1_sim_kp1 = f1_sim_k + w1_sim_k;

x2_sim_kp1 = f2_sim_k + w2_sim_k;

%Calculating output and adding meas noise:

v1_sim_k = sqrt(cov_meas_noise_v1)*randn;

y1_sim_k = x1_sim_k + v1_sim_k;

%Storage:

t_array(k) = t_k;

u_sim_array(k) = u_k;

x1_sim_array(k) = x1_sim_k;

x2_sim_array(k) = x2_sim_k;

x3_sim_array(k) = d_k;

y1_sim_array(k) = y1_sim_k;
```

```
%Preparing for time shift:

x1_sim_k = x1_sim_kp1;

x2_sim_k = x2_sim_kp1;

%Updating u and y for use in MHE:

u_mhe_array = [u_mhe_array(2:N_mhe),u_k];

y1_meas_mhe_array = [y1_meas_mhe_array(2:N_mhe),y1_sim_k];

y_meas_mhe_array = [y1_meas_mhe_array];

%-------------------------------------------------------------------

if k > N_mhe

Q = cov_process_disturb_w;

R = cov_meas_noise_v;

covars.Q = Q;

covars.R = R;

%Matrices defining linear constraints for use in fmincon:

A_ineq = []; B_ineq = []; A_eq = []; B_eq = [];

%fmincon initialization:

x1_est_init_error = 0;

x2_est_init_error = 0;

x3_est_init_error = 0;

x_est_init_error=[x1_est_init_error;x2_est_init_error;x3_est_init_error];

%Guessed optim states:

%Guessed present state (x_k) is needed to calculate optimal present
meas

%(y_k).  Model is used in prediction:

x1_km1 = x1_est_optim_array(N_mhe);

x2_km1 = x2_est_optim_array(N_mhe);
```

```
x3_km1 = x3_est_optim_array(N_mhe);

dx1_dt_km1 = x2_km1;

dx2_dt_km1 = (-x2_km1 + K*u_k + x3_km1)/T;

dx3_dt_km1 = 0;

x1_pred_k = x1_km1 + Ts*dx1_dt_km1;

x2_pred_k = x2_km1 + Ts*dx2_dt_km1;

x3_pred_k = x3_km1 + Ts*dx3_dt_km1;

%Now, guessed optimal states are:

x1_est_guess_array = ...

[x1_est_optim_array(2:N_mhe),x1_pred_k];

x2_est_guess_array = ...

[x2_est_optim_array(2:N_mhe),x2_pred_k];

x3_est_guess_array = ...

[x3_est_optim_array(2:N_mhe),x3_pred_k];

x_est_guess_matrix = ...

[x1_est_guess_array;x2_est_guess_array;x3_est_guess_array];

%Lower and upper limits of optim variables:

x1_est_max = 100;

x2_est_max = 10;

x3_est_max = 10;

x1_est_max_array = zeros(1,N_mhe) + x1_est_max;

x2_est_max_array = zeros(1,N_mhe) + x2_est_max;

x3_est_max_array = zeros(1,N_mhe) + x3_est_max;

x1_est_min = -100;

x2_est_min = -10;

x3_est_min = -10;
```

```
x1_est_min_array = zeros(1,N_mhe) + x1_est_min;

x2_est_min_array = zeros(1,N_mhe) + x2_est_min;

x3_est_min_array = zeros(1,N_mhe) + x3_est_min;

x_est_ub_matrix = [x1_est_max_array;x2_est_max_array;x3_est_max_array];

x_est_lb_matrix = [x1_est_min_array;x2_est_min_array;x3_est_min_array];

%Creating function handles:

fun_objective_handle = ...

@(x_est_matrix) fun_objective_mhe(x_est_matrix,...

y_meas_mhe_array,u_mhe_array,model_params,covars,x_est_init_error,n,N_mhe,Ts);

fun_constraints_handle = ...

@(x_est_matrix) fun_constraints_mhe(x_est_matrix,...

y_meas_mhe_array,u_mhe_array,model_params,covars,x_est_init_error,n,N_mhe,Ts);

%Calculating MHE estimate using fmincon:

%fmincon_options = optimoptions(@fmincon);

fmincon_options = optimoptions(@fmincon,'display','none');

% fmincon_options =
optimoptions(@fmincon,'algorithm','sqp','display','none');

[x_est_optim_matrix,fval,exitflag,output,lambda,grad,hessian] = ...

fmincon(fun_objective_handle,x_est_guess_matrix,A_ineq,...

B_ineq,A_eq,B_eq,x_est_lb_matrix,x_est_ub_matrix,...

fun_constraints_handle,fmincon_options);

x1_est_optim_array = x_est_optim_matrix(1,:);

x2_est_optim_array = x_est_optim_matrix(2,:);

x3_est_optim_array = x_est_optim_matrix(3,:);

% fval

end %if
```

```
x1_est_optim_plot_array(k) = x1_est_optim_array(end);

x2_est_optim_plot_array(k) = x2_est_optim_array(end);

x3_est_optim_plot_array(k) = x3_est_optim_array(end);

%Continuous plotting:

x_lim_array=[t_start,t_stop];

if (k>1 & k<N)

if k < N_mhe

pause(1);

else

pause(0);

end

subplot(4,1,1)

plot([t_array(k-1),t_array(k)],...

[u_sim_array(k-1),u_sim_array(k)],'b-o');

if k==2

hold on

grid minor

xlim(x_lim_array);

ylim([0 10]);

title('u')

%ylabel('[m]')

%xlabel('t [s]')

end

subplot(4,1,2)

plot([t_array(k-1),t_array(k)],...

[x1_est_optim_plot_array(k-1),x1_est_optim_plot_array(k)],'r-o',...
[t_array(k-1),t_array(k)],...

[x1_sim_array(k-1),x1_sim_array(k)],'b-o');

if k==2

hold on

grid minor

xlim(x_lim_array);

ylim([0 100]);

title('x1\_sim = blue.  x1\_mhe\_est = red.')

%ylabel('[m]')

%xlabel('t [s]')

end

subplot(4,1,3)

plot([t_array(k-1),t_array(k)],...

[x2_est_optim_plot_array(k-1),x2_est_optim_plot_array(k)],'r-o',...

[t_array(k-1),t_array(k)],...

[x2_sim_array(k-1),x2_sim_array(k)],'b-o');

if k==2

hold on

grid minor

xlim(x_lim_array);

ylim([0 10]);

title('x2\_sim = blue.  x2\_mhe\_est = red.')

%ylabel('[m]')

%xlabel('t [s]')

end

subplot(4,1,4)
```

```
plot([t_array(k-1),t_array(k)],...

[x3_est_optim_plot_array(k-1),x3_est_optim_plot_array(k)],'r-o',...

[t_array(k-1),t_array(k)],...

[x3_sim_array(k-1),x3_sim_array(k)],'b-o');

if k==2

hold on

grid minor

xlim(x_lim_array);

ylim([0 3]);

title('x3\_sim = K\_sim = blue.  x3\_mhe\_est = d\_mhe\_est = red.')

%ylabel('[m]')

xlabel('t [s]')

end

end %if (k>1 & k<N)

end %sim loop

%-----------------------------------------------------

%Printing figure as pdf file:

saveas(h,'example_mhe','pdf')

%-----------------------------------------------------

%Defining local functions:

%-----------------------------------------------------

function f = fun_objective_mhe(x_est_matrix,...

y_meas_mhe_array,u_mhe_array,model_params,...

covars,x_est_init_error,n,N_mhe,Ts)

K = model_params.K;

T = model_params.T;
```

```
Q = covars.Q;

R = covars.R;

J_km1 = 0;

y1_meas_mhe_array = y_meas_mhe_array(1,:);

for k = 1:N_mhe

u_k = u_mhe_array(1,k);

x_k = x_est_matrix(:,k);

x1_k = x_k(1);

x2_k = x_k(2);

x3_k = x_k(3);

h1_k = x1_k;

y1_meas_k = y1_meas_mhe_array(1,k);

v1_k = y1_meas_k - h1_k;

v_k = [v1_k];

if k <= N_mhe-1

x_kp1 = x_est_matrix(:,k+1);

x1_kp1 = x_kp1(1);

x2_kp1 = x_kp1(2);

x3_kp1 = x_kp1(3);

dx1_dt_k = x2_k;

dx2_dt_k = (-x2_k + K*u_k + x3_k)/T;

dx3_dt_k = 0;

f1_k = x1_k + Ts*dx1_dt_k;

f2_k = x2_k + Ts*dx2_dt_k;

f3_k = x3_k + Ts*dx3_dt_k;

w1_k = x1_kp1 - f1_k;
```

```
w2_k = x2_kp1 - f2_k;

w3_k = x3_kp1 - f3_k;

w_k = [w1_k,w2_k,w3_k]';

end

dJ_k = w_k'*inv(Q)*w_k + v_k'*inv(R)*v_k;

J_k = J_km1 + dJ_k;

%Time shift:

J_km1 = J_k;

end %for k=1:N_mhe

f = J_k;

end

%------------------------------------------------

function [cineq,ceq]=fun_constraints_mhe(x_est_matrix,...

y_meas_mhe_array,u_mhe_array,model_params,covars,x_est_init_error,n,N,Ts)

cineq = []; % Compute nonlinear inequalities.  Calculated below.

ceq = []; % Compute nonlinear equalities.

end

%------------------------------------------------
```

[End of Example 1.11]

1.3.5 Model-predictive control

Model-predictive control (MPC) is the dominant model-based control method. In [Maciejowski, 2002], it is argued that "MPC is the only advanced control technique that is more advanced than standard PID to have a significant and widespread impact on industrial process control". The history of MPC may be traced back to Dynamic Matrix Control (DMC) method implemented by Cutler and Ramaker at Shell Oil in 1973 [Cutler & Ramaker, 1980]. A standard overview over MPC technology is given in [Qin & Badgwell, 2003]. A more recent overview is given in

[Lee, 2011].

MPC is available in various professional and industrial software tools, e.g. DeltaV Predict (Emerson Process), 800xA APC (ABB), PCS7 (Siemens), MPC Toolbox of Matlab and Simulink (Mathworks), and Control Design Toolkit of LabVIEW (National Instruments).

As a mathematical problem, MPC and MHE are almost identical: Both exploits a mathematical model which is run (basically simulated) over a time horizon. However, they can also be regarded as opposites of each other: MPC looks into the future, while MHE looks into the past. And, in MPC, the process measurement is an input (to the MPC), and the control signal is an output, while in MHE, the situation is opposite – the process measurement is an output (from the MHE), and the control signal is an input.

MPC exists in different versions. Here, nonlinear MPC is presented. The term “nonlinear” is used because the underlying mathematical model of the process to be controlled, is a *nonlinear* state space model. The model may be multivariable and may contain time delays. Nonlinear MPC can of course be applied to linear models, too, since linear state space models are just a special case of nonlinear state space models.

The principle of MPC is continuously calculation of the optimal (“best”) control signal sequence over a future or prediction time horizon using the following information:

- A process model. The model is used by the optimizer to simulate the process over the prediction horizon.
- The current process state as obtained from measurements and/or state estimates from a state estimator which typically is in the form of a Kalman filter.
- Setpoint values and process disturbance values known over the prediction horizon.
- Constraints (maximum and minimum values) on the control signal, the process variable, and state variables.

It can be claimed that MPC resembles closely how a human controls a process, like driving a car: The driver looks ahead to take into account future disturbances like other cars, pedestrians and other obstacles, and future speed setpoints as shown on the signs ahead, while manipulating the various actuators (throttle, break, steering wheel, gear).

The predictions/simulations in the MPC can be based on any model that is representative of the process to be controlled. A particularly flexible model form is a discrete time nonlinear state space model:

$$x_{k+1} = f(x_k, u_{k,}, d_k, \cdot) \tag{1.90}$$

$$y_k = g(x_k, \cdot) \tag{1.91}$$

where x is the state vector, y is the process output variable vector, u is the control signal vector, and d is the process disturbance vector. f and g are nonlinear (or linear) vectorial functions. This model form is described in more detail in Section 1.3.4.

Figure 1.22 illustrates the principle of MPC. The predicted values are found by successive simulations over the prediction horizon performed by the optimizer until the optimal solution (optimal control signal sequence) has been found. The optimal control sequence or array (or matrix in the multivariable case), u_{opt} is calculated as the solution of an optimization problem where typically the future (predicted) control errors and control signal changes are minimized in a least squares sense. And from this optimal future control sequence, the first element is picked out and applied as control signal to the process, i.e.

$$u(t_k) = u(k) = u_{\text{opt}}(1) \tag{1.92}$$

The optimization function to be minimized in MPC may be stated as follows:

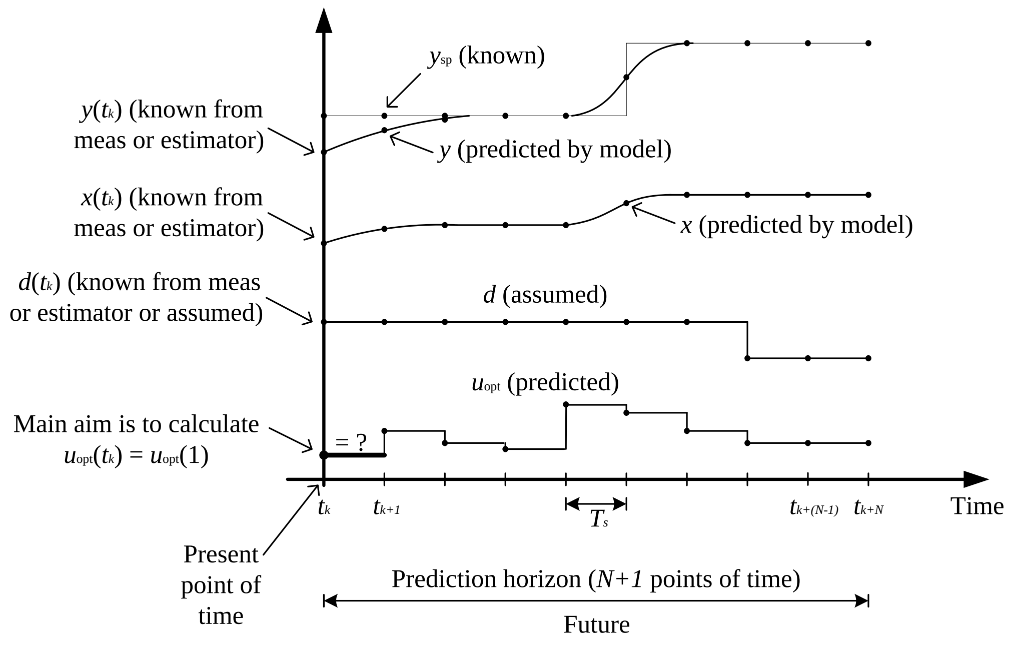
$$\min_U J \tag{1.93}$$

J is the objective function. It is defined below. U is a matrix containing the r control signals at each point of time of the prediction horizon:

$$\begin{aligned} U &= \left[u_k, u_{k+1}, \cdots, u_{k+(N-1)}, u_N\right] \\ &= \begin{bmatrix} u(1)_k & u(1)_{k+1} & \cdots & u(1)_{k+(N-1)} & u(1)_{k+N} \\ u(2)_k & u(2)_{k+1} & \cdots & u(2)_{k+(N-1)} & u(2)_{k+N} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ u(r)_k & u(r)_{k+1} & \cdots & u(r)_{k+(N-1)} & u(r)_{k+N} \end{bmatrix} \end{aligned} \tag{1.94}$$

The number of optimization variables is the number of elements of U. The number is $r(N+1)$.

U can be denoted the total control signal matrix. U will be the solution of



y_{sp} (known)
$y(t_k)$ (known from meas or estimator)
y (predicted by model)
$x(t_k)$ (known from meas or estimator)
x (predicted by model)
$d(t_k)$ (known from meas or estimator or assumed)
d (assumed)
u_{opt} (predicted)
Main aim is to calculate $u_{opt}(t_k) = u_{opt}(1)$
= ?
t_k
t_{k+1}
T_s
$t_{k+(N-1)}$
t_{k+N}
Time
Present point of time
Prediction horizon ($N+1$ points of time)
Future

Figure 1.22: The principle of MPC.

the MPC optimization problem. From this U,

$$u_k = \begin{bmatrix} u(1)_k \\ u(2)_k \\ \vdots \\ u(r)_k \end{bmatrix}$$

is used as the control signal applied to the process actuator.[16]

The optimization function to be minimized, cf. Eq. (1.93), is

$$J = \sum_{i=k}^{k+N} \left(\|e\|_{C_e}^2 + \|du\|_{C_{du}}^2 \right) \tag{1.95}$$

where expressions like $\|\cdot\|_M$ means M-quadratic norm, see Page 66. In more detail, Eq. (1.95) is

$$J = \sum_{i=k}^{k+N} \left(e_i^T C_e e_i + du_i^T C_{du} du_i \right) \tag{1.96}$$

The symbols in Eq. (1.96) are described below.

[16]You can represent U with a Matlab *matrix* in fmincon. So it is not necessary to transform this matrix to an array for fmincon.

The control error vector:

$$e_i = \begin{bmatrix} e(1)_i \\ \vdots \\ e(m)_i \end{bmatrix} \tag{1.97}$$

where $e(j)_i$ is the control error related to process output no. j at time-index i:

$$e(j)_i = y(j)_{\mathrm{sp}_i} - y(j)_i \tag{1.98}$$

The control signal change vector:

$$du_i = \begin{bmatrix} du(1)_i \\ \vdots \\ du(r)_i \end{bmatrix} \tag{1.99}$$

where $du(j)_i$ is the control signal change relative to the control signal at the previous point of time:

$$du(j)_i = u(j)_i - u(j)_{i-1} \tag{1.100}$$

The matrixes C_e and C_{du} in Eq. (1.95) are cost (or weight) matrixes which typically are set as constant matrixes:

$$C_e = \begin{bmatrix} C_e(1,1) & & 0 \\ & \ddots & \\ 0 & & C_e(m,m) \end{bmatrix} \tag{1.101}$$

$$C_{du} = \begin{bmatrix} C_{du}(1,1) & & 0 \\ & \ddots & \\ 0 & & C_{du}(r,r) \end{bmatrix} \tag{1.102}$$

C_e and C_{du} are tuning factors in MPC.

Now, Eq. (1.96) can be written in detail as

$$J = \sum_{i=k}^{k+N} \left[C_e(1,1)e(1)_i^2 + \cdots + C_e(m,m)e(m)_i^2\right] + \left[C_{du}(1,1)du(1)_i^2 + \cdots + C_{du}(r,r)du(r)_i^2\right] \tag{1.103}$$

Roughly said, MPC produces the control signal that gives the optimal comprimise between control errors and control signal changes. It is not possible to obtain both very small control errors and very small control signal changes. Hence, a comprimise will always exist.

Constraints. You may include constraints in the MPC optimization problem: Typically, upper and lower bounds are set for the control signal. Furthermore, you can set limits on the process output variable and on certain state variables. For example, if the liquid level in a tank is one state variable, it is natural to define a maximum level limit and a minimum level limit.

Guessed value of U. When solving the optimization problem, it is necessary that the optimizer is supplied with a good guess of the optimization variable, U. As a good value of U_{guess} at time index k, here denoted U_{guess_k}, we can use the optimal solution found at time index $k-1$ (the previous point of time):

$$U_{\text{guess}_k} = U_{\text{opt}_{k-1}} \tag{1.104}$$

No linearization. In MPC as described above, no linearization of the model is needed. This is contrary to the traditional Linear Quadratic Regulator (LQR), which has several similarities with MPC, as LQR assumes a linear state space model.

The need for a state estimator. The optimizer in MPC uses successive simulations for the prediction. For the simulations to become accurate, it is necessary that the initial state of the simulations are close to the present state, x_k, of the process to be controlled. Typically, not all the states are measured, and if so, a state estimator – most often a Kalman Filter – is used to provide an estimate of x_k. Even if all the states are estimated, state estimator can be useful for several reasons:

- The estimates are typically less noisy than the (raw) measurements.
- If a process sensor fails, and this failure is detected, the state estimator may be configures to continue providing a representative state estimate despite the lack of measurement-based update or correction of the estimate. This enhances the robustness of the MPC.
- A state estimator may be used to estimate disturbances and/or model parameters. This may increases the robustness of the MPC as the process model underlying the MPC becomes more accurate. Disturbances and model parameters can be estimated as augmented state variables modelled as constants, i.e. as state variables having time-derivatives equal to zero but with an additive random

disturbance/noise. This augmentation is explained in detail on Page 91.

Tuning factors of MPC. The main tuning factors of MPC are:

- *The prediction horizon length, N.* The larger N, the better ability to take into account future setpoints and disturbances. A drawback of selecting a large N, is the increase of the computational demand which is due to the more challenging optimization problem (more optimization variables to be optimized) and longer simulations. A typical value of N for simple applications seems to be between 5 and 50, assuming an appropriate time step length (which may be e.g. 1/5 of smallest time-constant-like dynamics represented by the model). To reduce the computional burden, control signal blocking can be considered, as explained earlier in this section.

- *The control error cost matrix, C_e.* Increasing the value of $C_e(j,j)$, forces the pertinent control error, $e(j)$, to become smaller[17], but at the expense of larger variation in the control signal. Initially, you may try setting $C_e(j,j)$ equal to the square of inverse of the maximum expected absolute value of the control error:

$$C_e(j,j) = \frac{1}{[|e(j)|_{\max}]^2} \tag{1.105}$$

This implies a normalization of the error terms in the objective function. For example, the first error term in Eq. (1.103) becomes

$$C_e(1,1)e(1)_i^2 = \frac{e(1)_i^2}{[|e(1)|_{\max}]^2}$$

Then, you may try using only C_{du} as a tuning parameter (cf. next item). In the scalar case, i.e. $m = 1$ and $r = 1$, you may simply set $C_e = 1$ since it is only the ratio between C_e and C_{du} that counts for the tuning.

- *The control signal change cost matrix, C_{du},.* Also the terms in C_{du} may be normalized before the tuning. Initially, you may try setting $C_{du}(j,j)$ equal to the square of inverse of the maximum change (between two time steps) of the control signal:

$$C_{du}(j,j) = \frac{1}{[|du(j)|_{\max}]^2} \tag{1.106}$$

[17]The higher cost of something, the less of it is bought/used.

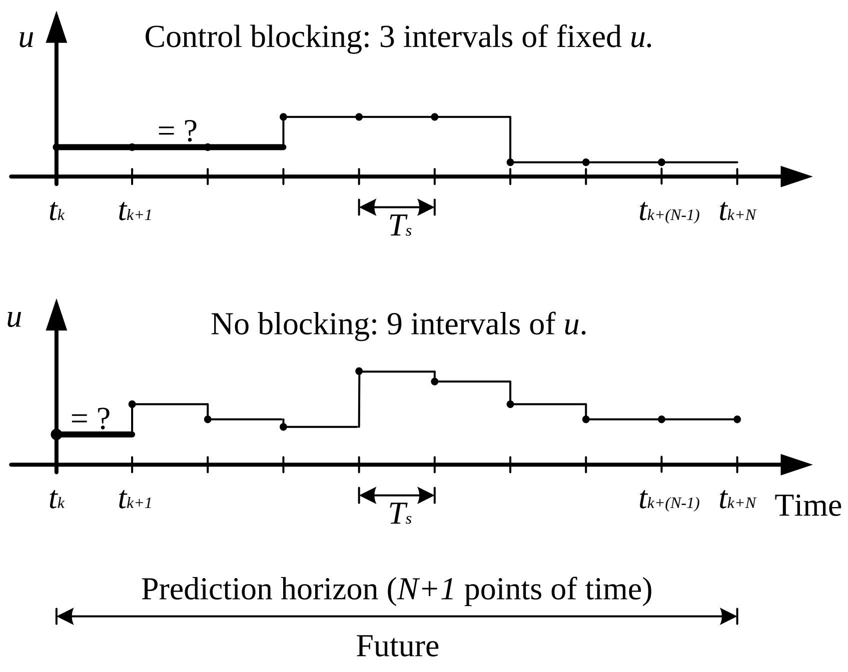

u
Control blocking: 3 intervals of fixed u.
= ?
tk
tk+1
Ts
tk+(N-1)
tk+N
u
No blocking: 9 intervals of u.
= ?
tk
tk+1
Ts
tk+(N-1)
tk+N
Time
Prediction horizon (N+1 points of time)
Future

Figure 1.23: Control blocking.

For example, the first control signal change term in Eq. (1.103) becomes

$$C_{du}(1,1)du(1)_i^2 = k_{du(1,1)}\frac{du(1)_i^2}{[|du(1)|_{\max}]^2}$$

where the factor $k_{du(1,1)}$ with the default value of 1 has been included for tuning purposes. Then, you can try tuning $k_{du(1,1)}$: Increasing it gives smoother control signal (less change). Decreasing gives more abrupt changes in the control signal.

Control blocking. Control blocking can be used to reduce the number of optimization variables. Control blocking is to fix the control signal in time-blocks in the predition horizon, see Figure 1.23. My experience from is that using as small number as 3 intervals may not detoriate the performance of the MPC. I have even tried using only one interval (i.e. constant u throughout the horizon), with acceptable performance. Control blocking, as other settings, should be tested in simulations before being applied to a real process.

Example 1.12 ***Model-predictive control with fmincon (Matlab)***

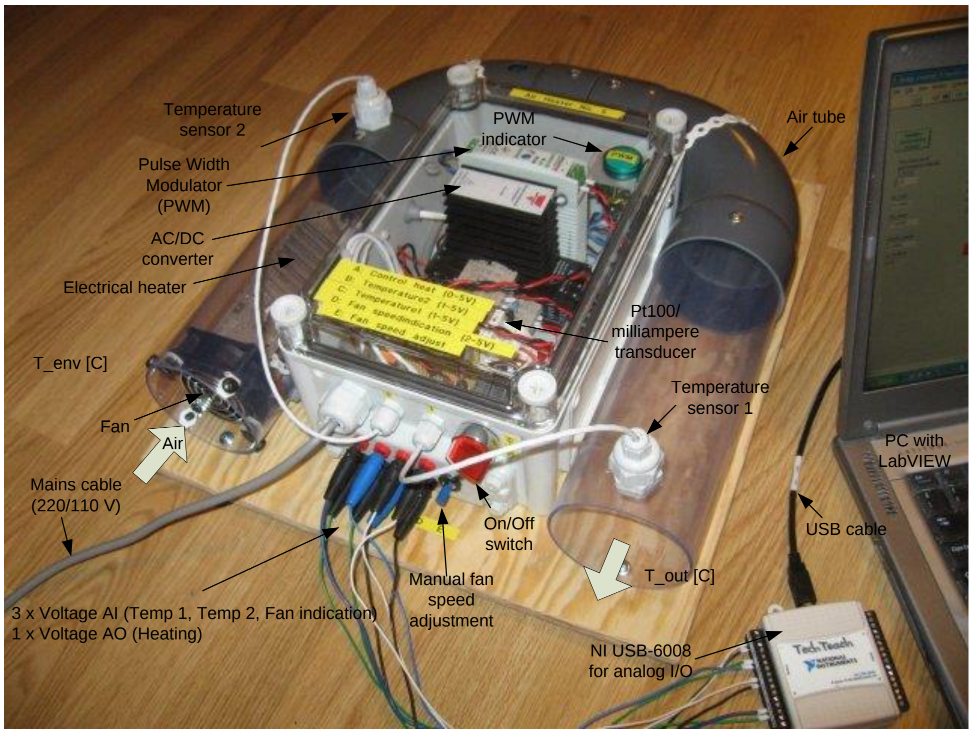


Figure 1.24: Air heater.

This example is about MPC for a simulated air heater lab station. The physical air heater is described on http://home-usn.no/finnh/air_heater. Figure 1.24 shows the air heater.[18]

The continuous-time model assumed representing the air heater is:

$$\theta_t \dot{T}_{\text{heat}}(t) = -T_{\text{heat}}(t) + K_h \left[u(t - \theta_d) + d \right] \tag{1.107}$$

$$T_{\text{out}}(t) = T_{\text{heat}}(t) + T_{\text{env}} \tag{1.108}$$

Variables and parameters and assumed parameter values are defined in Table 1.1. The model (1.107)-(1.108) can be characterized as a "time-constant with time-delay" model.

Figure 1.25 shows the results with the MPC applied to a simulated air heater. The simulator and the MPC, including a state estimator in the form of an augmented Kalman Filter for estimation of T_{heat}, d, and T_{out}, using the (simulated) measurement of T_{out} as process measurement. MPC

[18]University College of Southeast Norway, Porsgrunn, has 26 of identical units of this lab station, being used in several control courses in both bachelor and master programmes in technology.

Table 1.1: Nomenclature of the mathematical model of the air heater.

Symbol	**Unit**	**Value (default)**	**Description**
K_h	[°C/V]	3.5	Heater gain.
T_{env}	[°C]	25	The environmental, or ambient, temperature. It is the temperature in the outlet air of the air tube when the control signal to the heater has been set to zero for relatively long time (some minutes).
T_{heat}	[°C]	-	The additive contribution to the total temperature Tout due to the heater.
T_{out}	[°C]	-	Temperature of the air flowing out of tube. Measured by a sensor.
u	[V]	-	Control signal to heater.
d	[V]	-0.5	Input disturbance (added to the control signal).
θ_d	[s]	3.0	Time-delay representing air transportation and sluggishness of heater.
θ_t	[s]	23.0	Time-constant representing sluggishness of heater.

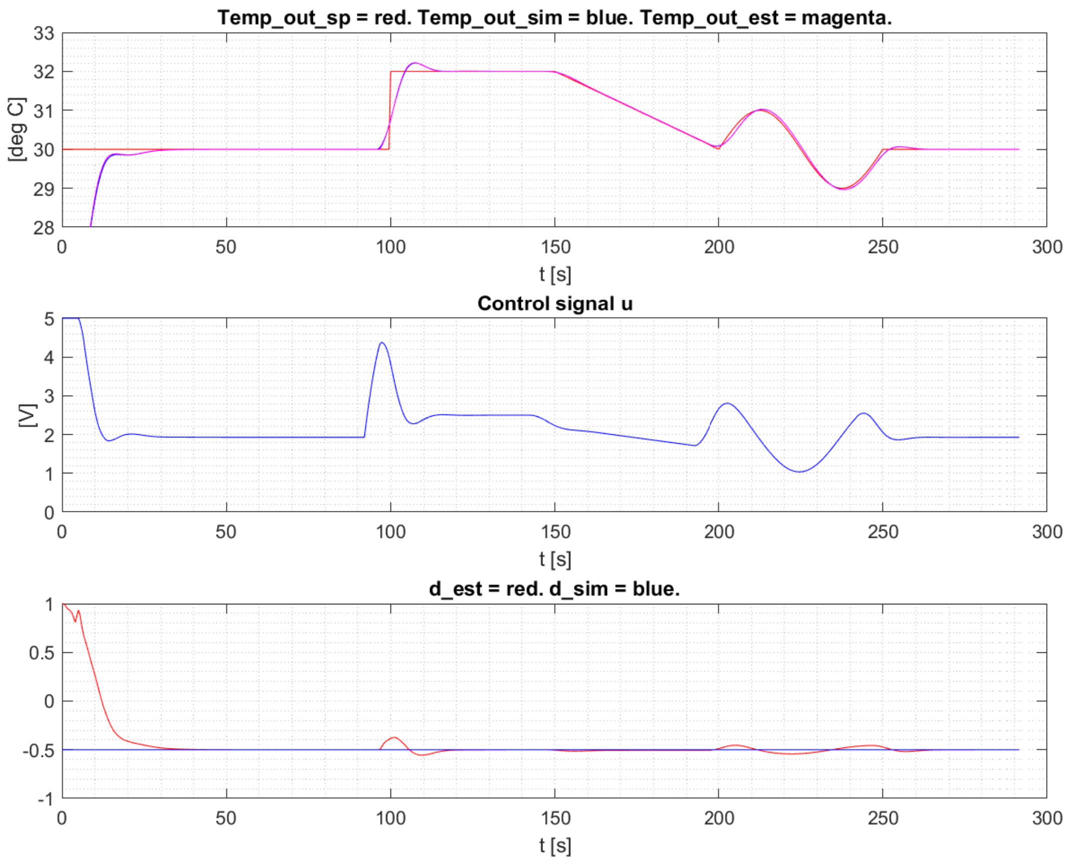


Figure 1.25: Example 1.12: Control of the outlet temperture of the air heater with MPC.

is implemented in Matlab with fmincon as optimizer. Control blocking is not implemented.

Below is a Matlab script, including comments, that implements the air heater simulator with MPC and Kalman Filter, using the fmincon optimizer. fmincon is described in Section 1.2.6.2.

Notes about the Matlab implementation:

- The objective function and the constraints function are defined as local functions within the script, and they exist only within the script. Local functions are supported from Matlab version R2016b.

Comments to the results:

- The MPC seems to work very well as the control is accurate, without excessive changes in the control signal.
- The MPC starts increasing the control signal before the point of time of the setpoint change. This illustrates well the predictive nature of the MPC.
- The tracking of the setpoint ramp and sinusoid is accurate. Although not shown here, with a well tuned PI controller, the control error remains non-zero during the setpoint ramp, and the error is substantial during the sinusoidal setpoint.

Script name: script_mpc_airheater_fmincon.m.

```
%Finn Aakre Haugen, USN

%18 April 2018

%MPC control of simulated air heater

%http://home.usn.no/finnh/air_heater/

%---------------------------------

clear all

close all

format short

%---------------------------------

%Process params:

gain = 3.5; %[deg C]/[V]

theta_const = 23; %[s]

theta_delay = 3; %[s]

model_params.gain = gain;

model_params.theta_const = theta_const;

model_params.theta_delay = theta_delay;

Temp_env_k = 25; %[deg C]
```

```
%---------------------------------
%Time settings:
Ts = 0.5; %Time-step [s]
t_pred_horizon = 8;
N_pred = t_pred_horizon/Ts;
t_start = 0;
t_stop = 300;
N_sim = (t_stop-t_start)/Ts;
t = [t_start:Ts:t_stop-Ts];
%-----------------------
%MPC costs:
C_e = 1;
C_du = 20;
mpc_costs.C_e = C_e;
mpc_costs.C_du = C_du;
%---------------------------------
%Defining sequence for temp_out setpoint:
Temp_sp_const = 30; %[C]
Ampl_step = 2; %[C]
Slope = -0.04; %[C/s]
Ampl_sine = 1; %[C]
T_period = 50; %[s]
t_const_start = t_start;t_const_stop = 100;
t_step_start = t_const_stop;t_step_stop = 150;
t_ramp_start = t_step_stop;t_ramp_stop = 200;
t_sine_start = t_ramp_stop;t_sine_stop = 250;
```

```
t_const2_start = t_sine_stop;t_const2_stop = t_stop;

for k = 1:N_sim

if (t(k) >= t_const_start & t(k) < t_const_stop),

Temp_sp_array(k) = Temp_sp_const;

end

if (t(k) >= t_step_start & t(k) < t_step_stop),

Temp_sp_array(k)=Temp_sp_const + Ampl_step;

end

if (t(k) >= t_ramp_start & t(k) < t_ramp_stop),

Temp_sp_array(k) = Temp_sp_const+Ampl_step+Slope*(t(k)-t_ramp_start);

end

if (t(k) >= t_sine_start & t(k) < t_sine_stop),

Temp_sp_array(k) = ...

Temp_sp_const+Ampl_sine*sin(2*pi*(1/T_period)*(t(k)-t_sine_start));

end

if (t(k) >= t_const2_start),

Temp_sp_array(k) = Temp_sp_const;

end

end

%----------------------------------

%Initialization:

u_init = 0;

N_delay = floor(theta_delay/Ts) + 1;

delay_array = zeros(1,N_delay) + u_init;

%----------------------------------

%Initial guessed optimal control sequence:
```

```
Temp_heat_sim_k = 0; %[C]

Temp_out_sim_k = 28; %[C]

d_sim_k = -0.5;

%---------------------------------

%Initial values of Kalman Filter:

Temp_heat_est_k = 0; %[C]

Temp_out_est_k = 25; %[C]

d_est_k = 1; %[V]

x_est_pred_k = [Temp_heat_est_k; d_est_k]; %Initial pred state estim

n_states = length(x_est_pred_k);

%Covar of pred estim error:

P_est_error_Temp_heat = (0.01^2);

P_est_error_d = (0.01^2);

P_est_pred_k = diag([0.1*P_est_error_Temp_heat, 0.1*P_est_error_d]);

%---------------------------------

%Tuning of Kalman Filter:

cov_w_Temp_heat = 0.01^2;

cov_w_d = 0.01^2;

Q = diag([1*cov_w_Temp_heat, 1*cov_w_d]);

cov_v_Temp_out = 0.01^2;

R = diag([1*cov_v_Temp_out]);

%---------------------------------

%Initial guessed optimal control sequence:

u_guess = zeros(N_pred,1) + u_init;

%---------------------------------

%Initial value of previous optimal value:
```

```
u_opt_km1 = u_init;

%---------------------------------

%Defining arrays for plotting:

t_plot_array = zeros(1,N_sim);

Temp_out_sp_plot_array = zeros(1,N_sim);

Temp_out_sim_plot_array = zeros(1,N_sim);

u_plot_array = zeros(1,N_sim);

d_est_plot_array = zeros(1,N_sim);

d_sim_plot_array = zeros(1,N_sim);

%---------------------------------

%Matrices defining linear constraints for use in fmincon:

A = [];

B = [];

Aeq = [];

Beq = [];

%---------------------------------

%Lower and upper limits of optim variable for use in fmincon:

u_max = 5;

u_min = 0;

u_ub = zeros(1,N_pred) + u_max;

u_lb = zeros(1,N_pred) + u_min;

u_delayed_k = 2;

%----------------------------------

%Figure settings:

fig_posleft=8;fig_posbottom=2;fig_width=24;fig_height=18;

fig_pos_size_1=[fig_posleft,fig_posbottom,fig_width,fig_height];
```

```
h = figure(1);

set(gcf,'Units','centimeters','Position',fig_pos_size_1);

figtext='MPC control of air heater';

set(gcf,'Name',figtext,'NumberTitle','on')

%---------------------------------

%For-loop for MPC of simulated process incl Kalman Filter:

tic

for k = 1:(N_sim-N_pred)

t_k = t(k);

t_plot_array(k)= t_k;

%----------------------

%Kalman Filter for estimating states Temp_heat and d using meas of
Temp_out.

%Also, Temp_out is estimated.  All these estimates are used by the
MPC.

%Note:  The time-delayed u is used as control signal here.

%Matrices in linearized model:

A_cont = [-1/theta_const, gain/theta_const; 0,0];

C_cont = [1 0];

A_disc = eye(n_states) + Ts*A_cont;

C_disc = C_cont;

%Kalman gain:

K_k = P_est_pred_k*C_disc'*inv(C_disc*P_est_pred_k*C_disc' + R);

%Innovation process:

e_est_k = Temp_out_sim_k - Temp_out_est_k;

%Measurement-based correction of estimate = the applied estimate:

x_est_corr_k = x_est_pred_k + K_k*e_est_k;
```

```
Temp_heat_est_k = x_est_corr_k(1,1);

d_est_k = x_est_corr_k(2,1);

%Applied estimated process meas:

Temp_out_est_k = Temp_heat_est_k + Temp_env_k;

%Prediction of state estimate for next time-step:

dTemp_heat_est_corr_dt_k = ...

(1/theta_const)*(-Temp_heat_est_k + gain*(u_delayed_k + d_est_k))...

+ K_k(1,1)*e_est_k;

dd_est_corr_dt_k = 0 + K_k(2,1)*e_est_k;

dx_est_corr_dt_k = [dTemp_heat_est_corr_dt_k; dd_est_corr_dt_k];

x_est_pred_kp1 = x_est_corr_k + Ts*dx_est_corr_dt_k;

%Auto-covariance of error of corrected estimate:

P_est_corr_k = (eye(n_states)-K_k*C_disc)*P_est_pred_k;

%Auto-covariance of error of predicted estimate of next time step:

P_pred_kp1 = A_disc*P_est_corr_k*A_disc' + Q;

%Time shift:

P_est_pred_k = P_pred_kp1;

x_est_pred_k = x_est_pred_kp1;

%-----------------------

%Storage for plotting:

Temp_out_est_plot_array(k) = Temp_out_est_k;

d_est_plot_array(k) = d_est_k;

%-----------------------

%Setpoint array to optimizer:

Temp_sp_to_mpc_array = Temp_sp_array(k:k+N_pred);

Temp_out_sp_plot_array(k) = Temp_sp_array(k);
```

```
%-----------------------

%Estimated state to optimizer:

state_est.Temp_heat_est_k = Temp_heat_est_k;

state_est.d_est_k = d_est_k;

%-----------------------

%Calculating optimal control sequence:

fun_handle = @(u) fun_objectfunction_mpc_airheater...

(u,state_est,Temp_env_k,Temp_sp_to_mpc_array,...

model_params,mpc_costs,N_pred,Ts);

fmincon_options = optimoptions(@fmincon,'display','none');

% fmincon_options =
optimoptions(@fmincon,'algorithm','active-set','display','none');

% fmincon_options =
optimoptions(@fmincon,'algorithm','sqp','display','none');

[u_opt,fval,exitflag,output,lambda,grad,hessian] =...

fmincon(fun_handle,u_guess,A,B,Aeq,Beq,u_lb,u_ub,@fun_constraints_mpc_airheater,fmincon_optio

u_guess = u_opt; %Optimal solution to be used as guessed solution in
next iteration.

u_k = u_opt(1); %Optimal control signal (sample) to be applied

u_plot_array(k) = u_k; %Storage for plotting

u_opt_km1 = u_opt(1);

%------------------------------

%Applying optimal control signal to simulated process:

d_sim_k = -0.5;

d_sim_plot_array(k) = d_sim_k;

u_delayed_k = delay_array(N_delay);

u_nondelayed_k = u_k;
```

```
delay_array = [u_nondelayed_k,delay_array(1:end-1)];

dTemp_heat_sim_dt_k = ...

(1/theta_const)*(-Temp_heat_sim_k + gain*(u_delayed_k + d_sim_k));

Temp_heat_sim_kp1 = Temp_heat_sim_k + Ts*dTemp_heat_sim_dt_k;

Temp_out_sim_k = Temp_heat_sim_k + Temp_env_k;

Temp_out_sim_plot_array(k) = Temp_out_sim_k;%Storage for plotting

%------------------------------

%Time shift for simulator:

Temp_heat_sim_k = Temp_heat_sim_kp1;

%------------------------------

%Continuous plotting:

x_lim_array = [t_start,t_stop];

if (k>1 & k<N_sim)

pause(0.0);

subplot(3,1,1)

plot([t_plot_array(k-1),t_plot_array(k)],...

[Temp_out_sp_plot_array(k-1),Temp_out_sp_plot_array(k)],'r-',...

[t_plot_array(k-1),t_plot_array(k)],...

[Temp_out_sim_plot_array(k-1),Temp_out_sim_plot_array(k)],'b-',...

[t_plot_array(k-1),t_plot_array(k)],...

[Temp_out_est_plot_array(k-1),Temp_out_est_plot_array(k)],'m-');

if k==2

hold on

grid minor

xlim(x_lim_array);

ylim([28 33]);
```

```
title('Temp\_out\_sp = red.  Temp\_out\_sim = blue.  Temp\_out\_est =
magenta.')

ylabel('[deg C]')

xlabel('t [s]')

end

subplot(3,1,2)

plot([t_plot_array(k-1),t_plot_array(k)],...

[u_plot_array(k-1),u_plot_array(k)],'b-');

if k==2

hold on

grid minor

xlim(x_lim_array);

ylim([0 5]);

title('Control signal u')

ylabel('[V]')

xlabel('t [s]')

end

subplot(3,1,3)

plot([t_plot_array(k-1),t_plot_array(k)],...

[d_est_plot_array(k-1),d_est_plot_array(k)],'r-',...

[t_plot_array(k-1),t_plot_array(k)],...

[d_sim_plot_array(k-1),d_sim_plot_array(k)],'b-');

if k==2

hold on

grid minor

xlim(x_lim_array);
```

```
ylim([-1 1]);

title('d\_est = red.  d\_sim = blue.')

%ylabel('[m]')

xlabel('t [s]')

end

end %if (k>1 & k<N)

end

%Elapsed total time and avg loop time:

toc_total=toc

toc_loop=toc_total/k

%---------------------------------------------------

%Printing figure as pdf file:

% saveas(h,'example_mpc_air_heater','pdf')

%---------------------------------------------------

%Functions defined as local functions:

function f =
fun_objectfunction_mpc_airheater(u,state_est,Temp_env_k,...

Temp_sp_to_mpc_array,model_params,mpc_costs,N_pred,Ts)

gain = model_params.gain;

theta_const = model_params.theta_const;

theta_delay = model_params.theta_delay;

C_e = mpc_costs.C_e;

C_du = mpc_costs.C_du;

Temp_heat_k = state_est.Temp_heat_est_k;

d_k = state_est.d_est_k;

N_delay = floor(theta_delay/Ts) + 1;
```

```
delay_array = zeros(1,N_delay) + u(1);

u_km1 = u(1);

J_km1 = 0;

%Applying optimal control signal to simulated process using explicit
Euler:

for k = 1:N_pred

u_k = u(k);

Temp_sp_k = Temp_sp_to_mpc_array(k);

%Time delay:

u_delayed_k = delay_array(N_delay);

u_nondelayed_k = u_k;

delay_array = [u_nondelayed_k,delay_array(1:end-1)];

%Solving diff eq:

dTemp_heat_dt_k = ...

(1/theta_const)*(-Temp_heat_k + gain*(u_delayed_k + d_k));

Temp_heat_kp1 = Temp_heat_k + Ts*dTemp_heat_dt_k;

Temp_out_k = Temp_heat_k + Temp_env_k;

%Updating objective function:

e_k = Temp_sp_k - Temp_out_k;

du_k = (u_k - u_km1)/Ts;

J_k = J_km1 + Ts*(C_e*e_k^2 + C_du*du_k^2);

%Time shift:

Temp_heat_k = Temp_heat_kp1;

u_km1 = u_k;

J_km1 = J_k;

end
```

Table 1.2: Abbreviations

Abbreviation	**Meaning**
DMC	Dynamic matrix control
LS	Least squares (method)
MHE	Moving horizon estimation (or -estimator)
MPC	Model-predictive control (or -controller)
NLS	Nonlinear least squares (method)
SSPE	Sum of squared prediction errors

```
f = J_k;

end

%-------------------------------

function
[cineq,ceq]=fun_constraints_mpc_airheater(u,state_est,Temp_env_k,...

Temp_sp_to_mpc_array,model_params,mpc_costs,N_pred,Ts)

cineq = []; % Compute nonlinear inequalities.

ceq = []; % Compute nonlinear equalities.

end
```

[End of Example 1.12]

1.3.6 Process optimization

To appear

1.4 Nomenclature

1.4.1 Abbreviations

Table 1.2 defines abbreviations used in this document.

1.4.2 Mathematical symbols

To appear.

Bibliography

[Boegli, 2014] Boegli, M. 2014. *Real-Time Moving Horizon Estimation for Advanced Motion Control Application to Friction State and Parameter Estimation.* PhD thesis. Arenberg Doctoral School, KU Leuven, Belgium.

[Cutler & Ramaker, 1980] Cutler, C. R., & Ramaker, B. L. 1980. Dynamic matrix control - a computer control algorithm. *Proc. Joint Automatic Control Conference, USA-CA.*

[Edgar *et al.*, 2001] Edgar, Th. F., Himmelblau, D., & Lasdon, L. 2001. *Optimization of Chemical Processes.* McGraw-Hill.

[Lee, 2011] Lee, J. H. 2011. Model Predictive Control: Review of the Three Decades of Development. *International Journal of Control, Automation, and Systems*, **9**(3), 415–424.

[Maciejowski, 2002] Maciejowski, J.M. 2002. *Predictive Control with Constraints.* Prentice-Hall.

[Nocedal & Wright, 2006] Nocedal, J., & Wright, S. J. 2006. *Numerical Optimization, 2nd ed.* Springer.

[Qin & Badgwell, 2003] Qin, S. J., & Badgwell, Th. A. 2003. A survey of industrial model predictive control technology. *Control Engineering Practice*, **11**, 733–764.

[Robertson *et al.*, 1996] Robertson, D., Lee, J., & Rawlings, J. 1996. A Moving Horizon-Based Approach for Least-Squares Estimation. *AIChE Journal*, **8**, 2209–2224.