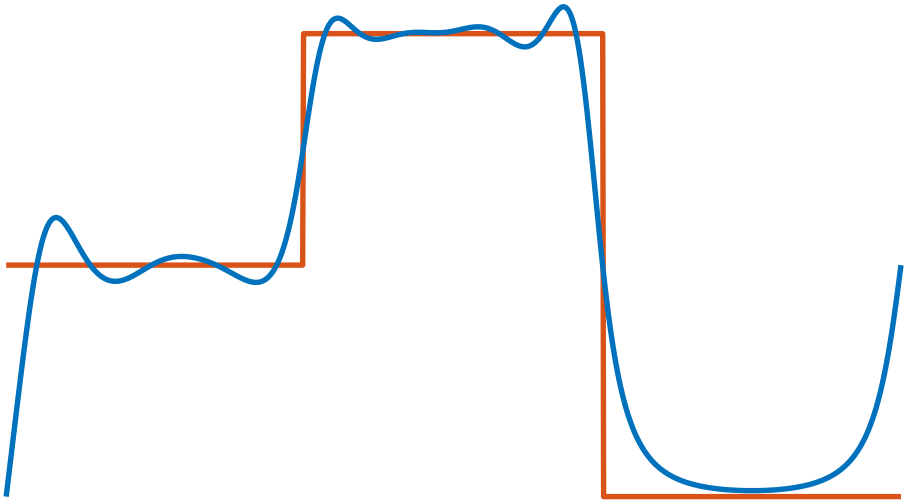
Seminar

Numerical methods for optimization and optimal control



written by
Hannah Weinmann
Melissa Finster
Max Steinlein
Jonas Kleineisel

held by
Prof. Dr. Alfio Borzi

University of Würzburg
August 2019

Contents

Introduction

The following presents a collection of algorithms concerned with optimization and optimal control that were presented and implemented during the seminar "Optimization" held by Prof. Dr. Alfio Borzi at the University of Würzburg during the summer term 2019.

It is not the aim of this text to explain all details and theory that is necessary to fully understand the design of each algorithm. For this we refer to the relevant literature[1]. Instead, we aim to provide to the educated reader a set of descriptions by pseudo-code and implementations that can be be used as starting points for completing practical tasks. We chose Matlab as language for its simplicity and suitability for numerical calculations. All codes should run just as well in Octave, an open-source implementation of Matlab, and can of course easily be transferred to any modern programming language.

Accompanying this PDF-document is an archive containing the source code files that were included below.

[1] see for example:
Jorge Nocedal, Stephen Wright: Numerical Optimization, Springer Science & Business Media, 2000

1 Optimization in finite dimensions

1.1 Linear steepest descent

Steepest Descent is a gradient descent algorithm to solve a system of linear equations $Ax = b$ given by a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$. This relies on the fact, that $x \in \mathbb{R}^n$ solves $Ax = b$ if and only if x is the minimal point of the second-order polynomial in n variables

$$f(x) = \frac{1}{2}x^T Ax - x^T b.$$

Since it is easy to compute the gradient as descent direction for this function and because we have an explicit formula for the optimal step size, this algorithm is rather simple. It is especially preferred over solving $Ax = b$ directly for a large number of variables n, when classical methods like Cholesky decomposition or Gauß elimination are too expensive. However, for ill-conditioned problems, this method can take a long time to converge, which is why one almost always prefers the conjugate gradient method presented below for this task.

Algorithm: Linear steepest descent algorithm

Input:

$A \in \mathbb{R}^{n\times n}$	symmetric positive definite matrix
$b \in \mathbb{R}^n$	arbitrary vector
$x_0 \in \mathbb{R}^n$	starting value
$\varepsilon > 0$	stopping condition

Output:

$x \in \mathbb{R}^n$	approximate solution of Ax = b

1: $v_0 \leftarrow b - Ax_0$
2: **for** $k = 0, \ldots, K_{\max}$ **do**
3: $\quad c_k \leftarrow Av_k$ ▷ update descent direction correction
4: $\quad t_k \leftarrow \dfrac{\langle v_k, v_k\rangle}{\langle v_k, c_k\rangle}$ ▷ calculate step size
5: $\quad x_{k+1} = x_k + t_k \cdot v_k$ ▷ perform optimization step
6: $\quad v_{k+1} = v_k - t_k \cdot c_k$ ▷ update descent direction
7: $\quad$ **if** $\|v_k\| < \varepsilon$ **then return** x_k
8: **return** $x_{K_{max}}$

Because of this algorithm's simplicity, it can be more or less directly converted from pseudo-code to a runnable Matlab-program.

"codes/LinearSteepestDescent/linearSteepestDescent.m"

```
%{
Steepest Descent Algorithm
Find the minimal point of the function Q(x) = (1/2)*x'*A*x - x'*b
where A is symmetric and positive definite
```

```
This is equivalent to solving Ax = b

Returns the approximation at the minimum and optionally the required
number of steps
%}

function [x K] = linearSteepestDescent(A, b, x0)

Kmax = 1000;          % Maximal number of steps
eps = 1e-8;           % Tolerance for accepting x as the minimum

x = x0;
v = b - A*x;

for K = 0:Kmax
    c = A*v;          % Update descent direction correction
    t = norm(v)^2 / (v.' * c); % Calculate step size
    x = x + t*v;      % Peform optimization step
    v = v - t*c;      % Update descent direction
    if norm(v) < eps% Terminate if the possible descent is small
        break;
    end
end

end
```

We test our code with the following test script using symmetric positive definite tridiagonal matrices. The distance from the exact solution is calculated by solving the system of linear equations $Ax = b$ exactly.

"codes/LinearSteepestDescent/testLSD.m"

```
% Test Parameters
n = 5;
A = gallery('tridiag', n, -1,2,-1); % SPD Matrix
b = ones(n,1);
x0 = zeros(n,1);      % Starting point

[xmin K] = linearSteepestDescent(A, b, x0);

fprintf("Solution:");
xmin'
fprintf("Required %i steps\nDistance from true minimum: %d\n", K, ...
    norm(xmin - A\b));
```

This yields the following output:

Matlab-Shell

```
>> testSD
Solution:
ans =

    2.5000    4.0000    4.5000    4.0000    2.5000

Required 122 steps
Distance from true minimum: 2.709315e-08
```

1.2 Line Search with Armijo condition

Though not technically an optimization algorithm, we discuss Line Search seperately since it will be used by several of the following methods.

Line Search is a backtracking method to find, for a given continuous function f, point x and direction p, a step size with which the function admits a sufficient decrease in the specified direction. In this case the condition for accepting a step size t is given by the so-called Armijo condition

$$f(x + t\,d) \leq f(x) - \alpha\, t\, \|p\|^2$$

for some parameter $\alpha > 0$. If a step size is not accepted, we perform the backtracking step by halving the proposed step size, therefore guaranteeing fast convergence.

Algorithm: Line Search

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	continuous function
$x \in \mathbb{R}^n$	current position
$p \in \mathbb{R}^n$	descent direction

Output:

t	suitable step size

1: $t \leftarrow 1$
2: **while** $f(x + tp) > f(x) - \alpha t \|p\|^2$ **do** ▷ Armijo condition for not accepting t
3: $\quad t \leftarrow \frac{t}{2}$ ▷ Backtracking step
4: $\quad$ **if** $t < 10^{-30}$ **then return** t ▷ No suitable step size found
5: **return** t

"codes/LineSearch/linesearch.m"

```
%{
Line search algorithm with Armijo condition

For a given function f and direction p at a point x,
finds a step size t such that f admits a sufficient decrease in
    direction p at x

Terminates with step size approximately 1e-30 if no sufficient
    decrease is found.
%}
function [t] = linesearch(f, p, x)
    alpha = 0.0001; % Specifies desired decrease
    t = 1;
    while f(x +t*p) > f(x) - alpha*t*power(norm(p), 2) % Armijo
        condition for not accepting t
        t = 0.5*t; % Backtracking step
        if t < 1e-30 % No decrease found
            return;
        end
    end
end
```

1.3 Nonlinear steepest descent

Often one is not in the convenient case of minimizing a linear-quadratic function like we saw in the case of the linear steepest descent algorithm. Instead, one has to deal with some arbitrary differentiable function f. In this case we still want to use a gradient descent approach, meaning that we start at some point an descent in the direction in which the function admits the greatest local descent, which is precicely the direction of the gradient. However, unlike in the linear case, we do not have a convenient formula which tells us exactly how far we have to go in order to achieve maximal descent. Therefore we have to use Line Search to determine a suitable step size.

Algorithm: Steepest descent algorithm

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	the gradient of f
$x_0 \in \mathbb{R}^n$	starting value

Output:

$x \in \mathbb{R}^n$	approximation of the minimum

1: **for** $k = 0, \ldots, K_{\max}$ **do**
2: $p_{k+1} \leftarrow -\nabla f(x_k)$ ▷ compute descent direction
3: $t_{k+1} \leftarrow \text{linesearch}(f, \nabla f, x_k)$ ▷ find step size
4: $x_{k+1} = x_k + t_{k+1} \cdot p_{k+1}$ ▷ perform optimization step
5: **if** $\|\nabla f(x_{k+1})\| < \varepsilon$ **then return** x_{k+1} ▷ terminate if the gradient is small
6: **return** $x_{K_{max}}$

In the implementation we additionally save the norm of the gradient in every step for visualization later. The Matlab code looks as follows:

”codes/SteepestDescent/steepestDescent.m”

```
%{
Steepest Descent Algorithm
Approximates the minimal point of a arbitrary function f with
given gradient grad by performing gradient descent with step size
    chosen by line search

Returns the approximation at the minimum, the required
number of steps and a vector with all the norms of the gradient at
    each step
%}
function [x K grads] = steepestDescent(f, grad, x0, Kmax, eps)

grads = zeros(1,Kmax);
x = x0;

for K = 0:Kmax
    p = -grad(x);   % Update descent direction via gradient
    t = linesearch(f, p, x); % Find step size
    x = x + t*p;    % Perform optimization step
    grads(K+1) = norm(p); % Save the gradient norm
    if norm(p) < eps% Terminate if the gradient is small
        break;
    end
end

end
```

To test the Steepest Descent algorithm, we use the so called Rosenbrock - function, a well-known benchmark function with known extrema. The true minimum lies at the constant 1-vector. We will always use this function as a nonlinear test function from now on.

”codes/SteepestDescent/testSD.m”

```
% Test Parameters
n = 5;
x0 = zeros(n,1);     % Starting point

% Use n-dimensional rosenbrock function for testing
[xmin K grads] = steepestDescent(@rosenbrock,@rosenbrockGrad, x0,
    2000, 1e-8);

fprintf("Solution:");
disp(xmin')
fprintf(strcat("Required %i steps\nDistance from true minimum:",...
  " %d\nNorm of gradient: %d\n"), K, norm(xmin - ones(1,n)), ...
  norm(rosenbrockGrad(xmin)));

plot(grads)
xlabel("step")
ylabel("norm of grad f")
xlim([-10, 2010])
```

"codes/SteepestDescent/rosenbrock.m"

```
% N-dimensional Rosenbrock function
function [fx] = rosenbrock(x)
    N = max(size(x));
    fx = 0;
    for i = 1:(N-1)
        fx = fx + 100*(x(i+1) - x(i)^2 )^2 + (1-x(i))^2;
    end
 end
```

"codes/SteepestDescent/rosenbrockGrad.m"

```
% Gradient of the N-dimensional Rosenbrock function
function [g] = rosenbrockGrad(x)
    N = max(size(x));
    g = zeros(N,1);
    for j=2:(N-1)
        g(j) = 200*(x(j+1) - x(j)^2)*(-2*x(j)) - 2*(1-x(j)) + ...
        200*(x(j) - x(j-1)^2);
    end
    g(1) = 200*(x(2) - x(1)^2)*(-2*x(1)) - 2*(1-x(1));
    g(N) = 200*(x(N) - x(N-1)^2);
end
```

Even after 2000 optimization steps we are still rather far away from the correct solution.

Matlab-Shell

```
>> testSD
Solution:    0.9914    0.9829    0.9659    0.9329    0.8699
Required 2000 steps
Distance from true minimum: 3.386944e-01
Norm of gradient: 1.287401e-01
```

If we look at the plot of the norm of the gradient of f at every step, we see that the norm of the gradient is oscillating very fast. This indicates that the method is oscillation between points. This is what prevents it from converging faster. In the following conjugate gradient method we will see how to circumvent this problem.

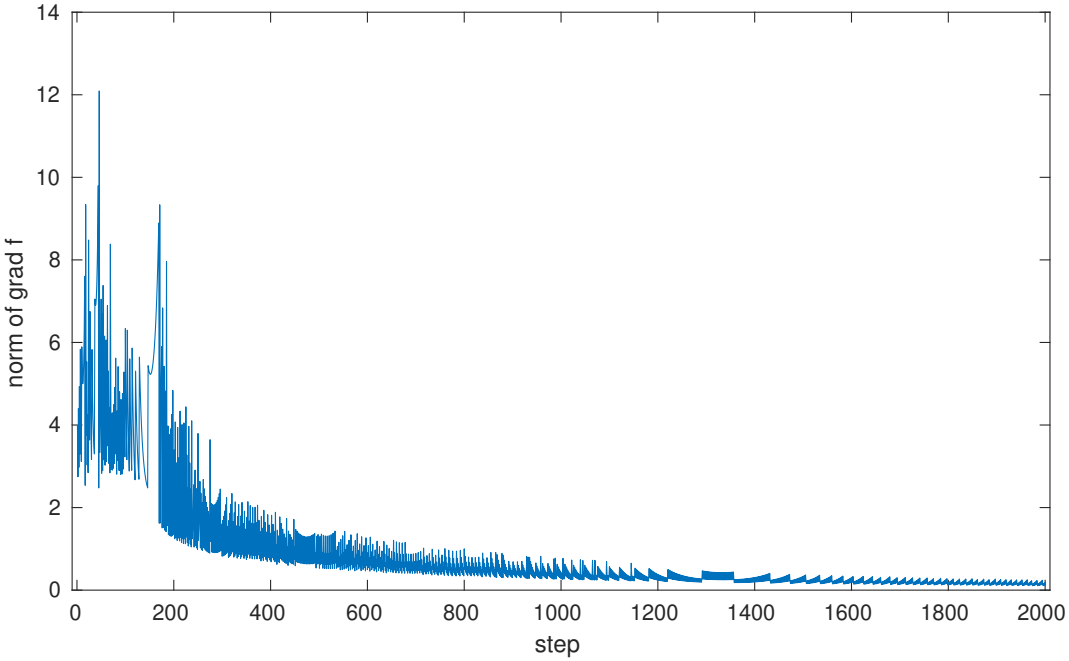


Figure 1.1: $\|\nabla f(x_k)\|$ at each step $k = 0, ..., K_{max}$ for steepest descent

1.4 Linear conjugate gradient

The linear conjugate gradient method (**CG**) is used for solving large systems of linear equations of the form $Ax = b$ with a symmetric positive definite Matrix $A \in \mathbb{R}^{n\times n}$. Since the method does not revert to expensive matrix operations it is especially used for large problems. The error of the solution is monotonically decreasing. One needs at most n steps to attain the exact solution.

The method is an iterative procedure which combines the gradient with the conjugated direction. One uses the residuum vectors $r_0, \ldots, r_{n-1} \in \mathbb{R}^n \backslash \{0\}$ and the coefficients $s_0, \ldots, s_{n-2} \in \mathbb{R}$ to determine the conjugate direction $p_0, \ldots, p_{n-1} \in \mathbb{R}^n$.

$$\begin{aligned} p_0 &= r_0 \\ &= b - Ax_0 \\ p_{k+1} &= r_{k+1} + s_k p_k \qquad\qquad k = 0, \ldots, n-2 \end{aligned}$$

Note that

$$\langle p_i, Ap_j \rangle = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

since A is symmetric positive definite.

Theorem

Let A be symmetric positive definite and $p_0, \ldots, p_{n-1} \in \mathbb{R}^n$. The following sequence

$$\begin{aligned} x_{k+1} &= x_k + t_k p_k \\ \text{with } t_k &= \langle b - Ax_k, p_k \rangle \qquad\qquad 0 \leq k \leq n-1 \end{aligned}$$

with any chosen $x_0 \in \mathbb{R}^n$ solves $Ax_n = b$.

Proof

$$
\begin{aligned}
Ax_{k+1} &= Ax_k + t_k Ap_k && k = 0, \ldots, n-1 \\
Ax_n &= Ax_{n-1} + t_{n-1} Ap_{n-1} \\
&= \ldots \\
&= Ax_0 + t_0 Ap_0 + \ldots + t_{n-1} Ap_{n-1} \\
t_k &= \langle b - Ax_k, p_k \rangle \\
&= \langle b - Ax_0, p_k \rangle + \langle Ax_0 - Ax_1, p_k \rangle + \ldots + \langle Ax_{k-1} - Ax_k, p_k \rangle \\
&= \langle b - Ax_0, p_k \rangle - t_0 \langle Ap_0, p_k \rangle - \ldots - t_{k-1} \langle Ap_{k-1}, p_k \rangle \\
&= \langle b - Ax_0, p_k \rangle \\
Ax_n - b &= Ax_0 - b + t_0 Ap_0 + \ldots + t_{n-1} Ap_{n-1} \\
\langle Ax_n - b, p_k \rangle &= \langle Ax_0 - b, p_k \rangle + t_k \\
&= \langle Ax_0 - b, p_k \rangle + \langle b - Ax_0, p_k \rangle \\
&= 0
\end{aligned}
$$

□

For the CG-Algorithm we need to determine s_k and t_k.
With

$$\langle r_{k+1}, p_{k+1} \rangle = \langle r_{k+1}, r_{k+1} \rangle$$

we can compute t_k.

$$
\begin{aligned}
t_k &= \frac{\langle r_k, p_k \rangle}{\langle Ap_k, p_k \rangle} \\
&= \frac{\langle r_k, r_k \rangle}{\langle Ap_k, p_k \rangle}
\end{aligned}
$$

And with

$$
\begin{aligned}
\langle r_{k+1}, r_k \rangle &= \langle r_k, r_k \rangle - t_k \langle Ap_k, r_k \rangle \\
&= \langle r_k, r_k \rangle - t_k \langle Ap_k, p_k - s_{k-1} p_{k-1} \rangle \\
&= 0 \\
\iff & \\
0 &= \langle Ar_{k+1}, p_k \rangle + s_k \langle Ap_k, p_k \rangle
\end{aligned}
$$

we get

$$
\begin{aligned}
s_k &= -\frac{\langle Ar_{k+1}, p_k \rangle}{\langle Ap_k, p_k \rangle} \\
&= -\frac{\langle r_{k+1}, t_k Ap_k \rangle}{\langle p_k, t_k Ap_k \rangle} \\
&= -\frac{\langle r_{k+1}, r_k - r_{k+1} \rangle}{\langle p_k, r_k - r_{k+1} \rangle} \\
&= \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle p_k, r_k \rangle}
\end{aligned}
$$

Now the CG-Algorithm is given by

Algorithm: Linear conjugate gradient

Input:

$A \in \mathbb{R}^{n \times n}$	S.p.d. Matrix
$b \in \mathbb{R}^n$	Vector b
$x_0 \in \mathbb{R}^n$	Starting vector

Output:

$x \in \mathbb{R}^n$	Approximated solution of $Ax = b$

1: $r_0 = b - Ax_0$
2: $p_0 = r_0$
3: **for** $k = 0, \ldots, K_{\max}$ **do**
4: $\quad t_k \leftarrow \frac{\langle r_k, r_k \rangle}{\langle Ap_k, p_k \rangle}$ ▷ Compute step size
5: $\quad x_{k+1} \leftarrow x_k + t_k p_k$ ▷ Perform optimization step
6: $\quad r_{k+1} = r_k - t_k A p_k$ ▷ Compute new residuum
7: $\quad$ **if** $\|r_{k+1}\| < \varepsilon$ **then return** x_{k+1} ▷ Terminate if the residuum is small
8: $\quad s_k = \frac{\langle r_{k+1}, r_{k+1} \rangle}{\langle r_k, r_k \rangle}$ ▷ Find coefficient to compute p_{k+1}
9: $\quad p_{k+1} = r_{k+1} + s_k p_k$ ▷ Compute conjungate direction vector
10: **return** $x_{K_{max}}$

The algorithm starts by adjusting x_{k+1} with a computed step size t_k and the conjugate direction p_k. Afterwards we update the residuum. If the residuum r_{k+1} is very small, t_{k+1} and p_{k+1} will be also very small. Therefore x_{k+2} will change rarely. We terminate and return the current x_{k+1}.

"codes/LinearConjugateGradient/linearCG.m"

```
%{
Linear Conjugate Gradient Method
Find the minimal point of the function Q(x) = (1/2)*x'*A*x - x'*b
where A is symmetric and positive definite
This is equivalent to solving Ax = b
%}
function [x, K] = linearCG(A, b, x0)

x = x0;              % Starting point
Kmax = 1000;         % Maximal number of steps
eps = 1e-8;          % Tolerance for accepting x as the minimum

p = b-A*x;
rOld = p;
rNew = p;

for K = 0:Kmax
    t = norm(rOld)^2 / (p'*A*p); % Calculate optimal step size
    x = x + t*p;     % Perform optimization step
    rNew = rOld - t*A*p; % Calculate residual
    if norm(rNew) < eps % Terminate for small residuals
      break;
    end
    s = norm(rNew)^2 / norm(rOld)^2; % Step size for direction
       correction
```

```
    p = rNew + s*p;   % Update descent direction while ensuring A-
        orthogonality of the p's
    rOld = rNew;
end

end
```

”codes/LinearConjugateGradient/testLCG.m”

```
% Test parameters
N = 10;
A = gallery('tridiag', N, -1,2,-1); % SPD Matrix
b = ones(n,1);

[xmin, K] = linearCG(A, b, zeros(N,1)); % Call linearCG

fprintf("Solution:");
disp(xmin')
fprintf("Required %i steps\nDistance from minimum: %d\n", K, norm(
    xmin - A\b));
```

We use the program *testLCG* to call the function *linearCG*. It generates the following output.

Matlab-Shell

```
>> testLinCG
Solution:    5.0000    9.0000   12.0000   14.0000   15.0000   15.0000   14.0000
12.0000    9.0000    5.0000

Required 4 steps
Distance from minimum: 3.202373e-15
```

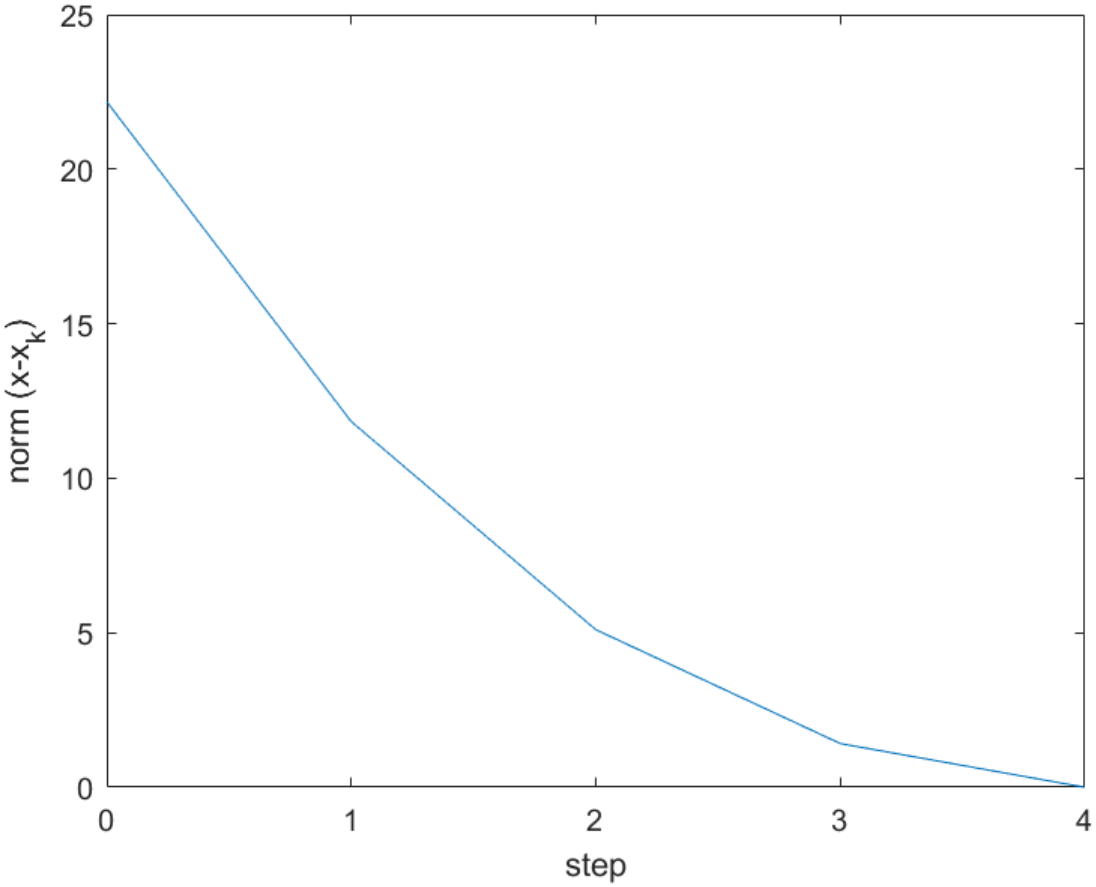


Figure 1.2: Error of step $K = 1, \ldots, 4$ of linear CG

In the chart (Figure 1.2) we see the norm of the difference between the exact solution and the solution of step k. The error is as expected monotonically decreasing. The algorithms stops at $K = 4 < 10 = N$ because of the termination condition.

1.5 Nonlinear conjugate gradient

A generalization of the linear conjugate gradient method is the nonlinear conjugate gradient method (NCG). In order for this method to converge the function must be twice differentiable at the minimum and the second derivative has to be invertible there.
The idea is to start with a given start value x_0, compute the descent direction and use line search to determine a suitable step size receiving $x_1 = x_0 - t_0 \cdot \nabla f(x0)$. After this first iteration an additional conjugate direction follows. There are different ways for the computation of the step size s_k, some examples are given below. After updating the conjugate direction ($p_{k+1} = -\nabla f(x_{k+1}) + s_k \cdot p_k$ with $p_0 = -\nabla f(x_0)$) a line search is accomplished ($t_k = \min_{t_k} f(x_k + t_k \cdot p_k)$). Then the position is updated ($x_{k+1} = x_k + t_k \cdot p_k$). This procedure repeats until a well enough approximation is found.
There are the following different ways for the computation for the step size s_k:

Fletcher - Reeves:

$$s_k = \frac{\|r_{k+1}\|^2}{\|r_k\|}$$

Polak-Ribiere:

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$
$$s_k = -\frac{r_{k+1}^T y_k}{\|r_k\|^2}$$

Dai-Yuan:

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$
$$s_k = -\frac{\|r_{k+1}\|^2}{p_k^T y_k}$$

Heyer-Zhang:

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$
$$s_k = \left(y_k - 2p_k \frac{\|y_k\|^2}{p_k^T y_k}\right)^T \left(-\frac{r_{k+1}}{p_k^T y_k}\right)$$

Algorithm: Nonlinear conjugate gradient

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	the gradient of f
$x0 \in \mathbb{R}^n$	starting value

Output:

$x \in \mathbb{R}^n$	approximation of the minimum

$p_0 \leftarrow -\nabla f(x_k)$ ▷ compute descent direction
for $k = 0, \ldots, K_{\max}$ **do**
  $t_{k+1} \leftarrow \text{linesearch}(f, p_k, x_k)$ ▷ find step size
  $x_{k+1} = x_k + t_{k+1} \cdot p_k$ ▷ perform optimization step
  if $\|\nabla f(x_{k+1})\| < \varepsilon$ **then return** x_{k+1} ▷ terminate if the gradient is small
  $s \leftarrow \frac{\nabla f(x_{k+1})}{\nabla f(x_k)}$ ▷ find step size of the direction correction
  $p_{k+1} \leftarrow -\nabla f(x_{k+1}) + s \cdot p_k$ ▷ update descent direction
return $x_{K_{max}}$

"codes/NonlinearCG/nonlinearCG.m"

```
%{
Nonlinear Conjugate Gradient Method
For any smooth function f with gradient grad,
finds minima of f by searching for zeros of the gradient
Returns the minimum x, the required number of steps and a vector with
     all the norms of the gradient at each step
```

```
%}
function [minx, K, grads] = nonlinearCG(f, grad, x0, Kmax, eps)

    x = x0;          % Starting point
    p = -grad(x);        % Initalize all descent directions with the
        gradient
    rOld = p;
    rNew = p;
    grads = zeros(1,Kmax);

    for K = 0:Kmax
        t = linesearch(f, p, x); % Determine step size by line search
        x = x + t*p;     % Perform optimization step
        rNew = -grad(x);
        grads(K+1) = norm(rNew);
        if(norm(rNew) < eps) % Terminate for small gradient
            break;
        end
        s = (norm(rNew))^2 / (norm(rOld))^2; % Fletcher-Reeves
            formula for step size of the direction correction
        %y = grad(x) - grad(x - t*p); % Dai-Yuan, Polak-Ribiere,
            Heyer-Zhang
        %s = (norm(grad(x)))^2 / (p' * y ); % Dai-Yuan
                %s = (rNew' * y) / (norm(rOld))^2; % Polak-Ribiere
                %s = (y-2p * (norm(y))^2/(p' * y)' * (- rNew / (p' *
                    y)); % Heyer-Zhang
        p = rNew + s*p;   % Update descent direction
        rOld = rNew;
    end
    minx = x;
end
```

”codes/NonlinearCG/testNCG.m”

```
% Test Parameters
n = 5;
x0 = zeros(n,1);      % Starting point

% Use n-dimensional rosenbrock function for testing
[xmin K grads] = nonlinearCG(@rosenbrock,@rosenbrockGrad, x0, 2e3, 1e
    -8);

fprintf("Solution:");
disp(xmin')
fprintf(strcat("Required %i steps\nDistance from true minimum:",...
  " %d\nNorm of gradient: %d\n"), K, norm(xmin - ones(1,n)), ...
  norm(rosenbrockGrad(xmin)));

plot(grads)
xlabel("step")
ylabel("norm of grad f")
xlim([-10, 2010])
```

For testing we use the N-dimensional Rosenbrock function as above and take the parameter in the Line Search as $\alpha = 0.01$.

Matlab-Shell

```
>> testNCG
```

```
Solution:   1.00000   1.00000   1.00000   1.00000   1.00000
Required 1089 steps
Distance from true minimum: 5.96334e-09
Norm of gradient: 8.04072e-09
```

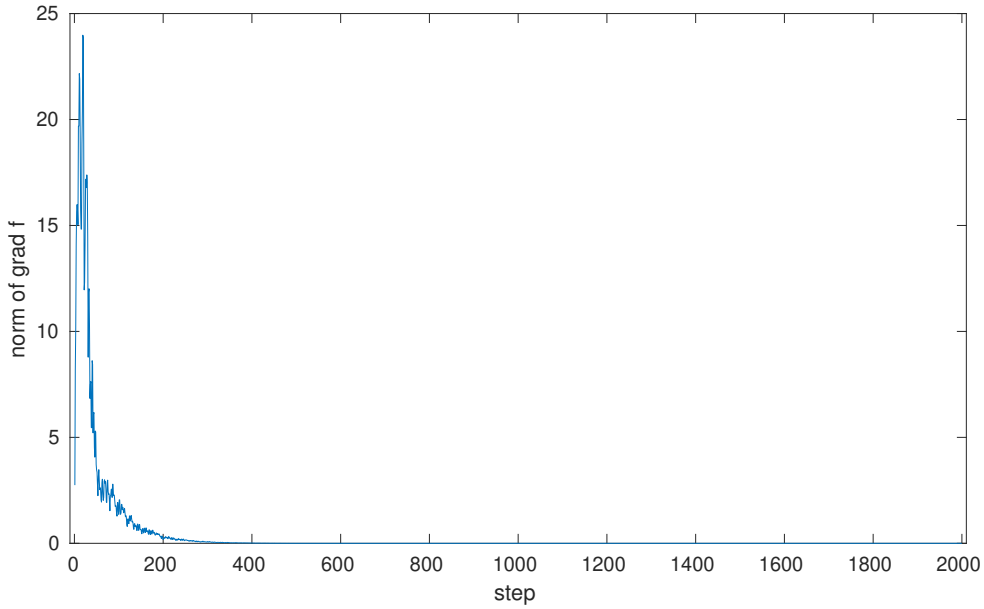


Figure 1.3: Norm of gradient for nonlinear conjugate gradient algorithm

1.6 Projected conjugate gradient

The projected conjugate gradient method is a variation of the NCG method with the difference that the optimum x must be within an admissible set

$$\Omega = \{x \in \mathbb{R}^n : a_i \leq x_i \leq b_i, \quad i = 1, ..., n\}$$

with some $a, b \in \mathbb{R}^n$. In each step of the PCG method, the solution x_k is determined as in the NCG method, where a modified linesearch only searches for step sizes that give a sufficient decrease within the admissible set. Afterwards a correction is made if x after the step is not in the admissible set:

$$\begin{aligned} &\text{if} \quad x_i < a_i, && \text{then} \quad x_i = a_i \\ &\text{if} \quad x_i > b_i, && \text{then} \quad x_i = b_i \\ &\text{if} \quad a_i \leq x_i \leq b_i, && \text{then} \quad x_i = x_i \end{aligned}$$

Algorithm: Projected conjugate gradient

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	the gradient of f
$x0 \in \mathbb{R}^n$	starting value
$a \in \mathbb{R}^n$	minimum admissible set
$b \in \mathbb{R}^n$	maximum admissible set

Output:

$x \in \mathbb{R}^n$	approximation of the minimum

1: $p_0 \leftarrow -\nabla f(x_k)$ ▷ compute descent direction
2: **for** $k = 0, \ldots, K_{\max}$ **do**
3: $t_{k+1} \leftarrow \text{linesearchProjected}(f, p_k, x_k, a, b)$ ▷ find step size in admissible set
4: $x_{k+1} = x_k + t_{k+1} \cdot p_k$ ▷ perform optimization step
5: $x_{k+1} \leftarrow \text{Proj}(x_{k+1}, a, b)$ ▷ correct x_k for the admissible set
6: **if** $\|\nabla f(x_{k+1})\| < \varepsilon$ **then return** x_{k+1} ▷ terminate if the gradient is small
7: $s \leftarrow \frac{\nabla f(x_{k+1})}{\nabla f(x_k)}$ ▷ find step size of the direction correction
8: $p_{k+1} \leftarrow -\nabla f(x_{k+1}) + s \cdot p_k$ ▷ update descent direction
9: **return** $x_{K_{max}}$

"codes/ProjectedCG/projectedCG.m"

```
%{
Projected Nonlinear Conjugate Gradient Method
For any smooth function f with gradient grad,
finds minima of f within an admissible set by searching for zeros of
    the gradient
Returns the minimum x, the required number of steps and a vector with
     all the norms of the gradient at each step
%}
function [minx, K, grads] = projectedCG(f, grad, x0, Kmax, eps,a,b)

    x = x0;          % Starting point
    p = -grad(x);    % Initalize all descent directions with the
        gradient
    rOld = p;
    rNew = p;
    grads = zeros(1,Kmax);

    for K = 0:Kmax
        t = linesearchProjected(f, p, x,a,b); % Determine step size
            by projected line search
        x = proj(x + t*p, a,b);  % Perform optimization step within
            the admissible set
        rNew = -grad(x);
        grads(K+1) = norm(rNew);
        if(norm(rNew) < eps) % Terminate for small gradient
            break;
        end
        s = power(norm(rNew), 2) / power(norm(rOld),2); % Fletcher-
            Reeves formula for step size of the direction correction
        %y = grad(x) - grad(x - t*p); % Dei-Yuan
```

```
        %s = power(norm(grad(x)),2) / (p' * y ); % Dei-Yuan
        p = rNew + s*p;   % Update descent direction
        rOld = rNew;
    end
    minx = x;
end
```

"codes/ProjectedCG/proj.m"

```
function [z] = proj(x,a,b)
    n = length(x);
    z = x;
    for i = 1:n
        if x(i) < a(i)
            z(i) = a(i);
        elseif x(i)> b(i)
            z(i) = b(i);
        end
    end
end
```

"codes/ProjectedCG/linesearchProjected.m"

```
%{
Projected Line search algorithm with Armijo condition

For a given function f and direction p at a point x,
finds a step size t such that f admits a sufficient decrease in the
projected direction p at x

Terminates with step size approximately 1e-30 if no sufficient
    decrease is found.
%}
function [t] = linesearchProjected(f, p, x,a,b)
    alpha = 0.01; % Specifies desired decrease
    t = 1;
    while proj(f(x +t*p),a,b) > f(x) - alpha*t*power(norm(p), 2) %
        Armijo condition for not accepting t
        t = 0.5*t; % Backtracking step
        if t < 1e-30 % No decrease found
            return;
        end
    end
end
```

"codes/ProjectedCG/testPCG.m"

```
% Test Parameters
n = 5;
x0 = zeros(n,1);      % Starting point
a = -1 * ones(n,1);  % Admissible set, lower boundary
b = 0.8*ones(n,1);       % Admissible set, upper boundary
% Use n-dimensional rosenbrock function for testing
[xmin K grads] = projectedCG(@rosenbrock,@rosenbrockGrad, x0, 3000, 1
    e-8,a,b);
```

```
fprintf("Solution:");
disp(xmin')
fprintf(strcat("Required %i steps\nNorm of gradient: %d\n"),...
  K, norm(rosenbrockGrad(xmin)));

plot(grads)
xlabel("step")
ylabel("norm of grad f")
xlim([-10, 2010])
```

For testing we use the N-dimensional Rosenbrock function as above and take the parameters a = (-1, ..., -1), b = (0.8, ..., 0.8). With these parameters, the global minimum $(1, ..., 1)$ does not lie in the admissible set.

Matlab-Shell

```
>> testPCG
Solution:   0.800000   0.665974   0.448899   0.221497   0.053064
Required 3000 steps
Norm of gradient: 10.1751
```

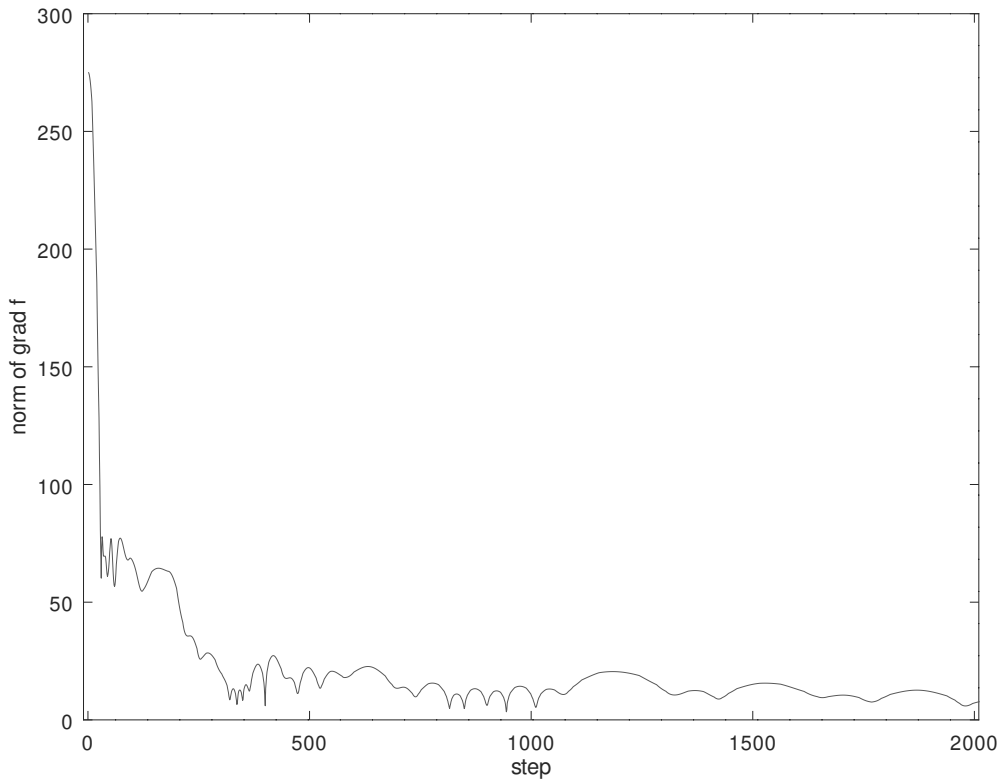


Figure 1.4: Norm of gradient for projected conjugate gradient

1.7 Newton method

Newton's method is a standard algorithm to approximate the roots of a nonlinear function. In the first place a one dimensional function f is considered. The idea of Newton's method is to begin with a start value x_0, linearize the function in this point i.e determine the tangent through $f(x_0)$ and compute the root of the tangent. With the obtained approximation a new linearization can be received to get a new approximation for the root. This leads to the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

This method can be generalized for systems of equations by using the Jacobian instead of the derivative of f.
To solve an optimization problem of a function f, Newton's method can be applied to the derivative f', since the derivative is zero at a minimum or maximum.

Algorithm: Newton method

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	the gradient of f
$\nabla^2 f : \mathbb{R}^n \to \mathbb{R}^{n \times n}$	the Hessian of f
$x_0 \in \mathbb{R}^n$	starting value

Output:

$x \in \mathbb{R}^n$	approximation of the minimum

1: **for** $k = 0, \ldots, K_{\max}$ **do**
2: $\quad x_{k+1} \leftarrow x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$ ▷ perform optimization step
3: $\quad$ **if** $\|f(x_{k+1})\| < \varepsilon$ **then return** x_{k+1} ▷ terminate if the function is small
4: **return** $x_{K_{max}}$

"codes/Newton/newton.m"

```
%{
Newton method
approximates an optimal point of a function f by approaching a root
    of the gradient of f with given gradient and Hessian of f.

Returns the approximation of the optimum, the required number of
    steps and a vector with all the norms of the gradient at each step
%}
function [x k grads] = newton (f, grad, hessian, x0, Kmax, eps)
        grads = zeros(1,Kmax);
        grads(1) = norm(f(x0));
        x = x0

        for k = 1 : Kmax
                x = x - inv(hessian(x)) * grad(x);
                grads(k+1) = norm(grad(x));
                if grads(k+1) < eps
                        break;
                end
```

```
        end
end
```

To test the Newton method, we use again the Rosenbrock - function.

"codes/Newton/testNewton.m"

```
% test parameters
n = 5;
x0 = zeros(n,1); % starting point

%Use n-dimensional rosenbrock function for testing
[x k grads] = newton(@rosenbrock, @rosenbrockGrad, @rosenBrockHessian
    , x0, 2000, 1e-8);

fprintf("Solution:");
disp(x')
fprintf(strcat("Required %i steps \n Distance from true optimum:",...
" %d\n Norm of gradient: %d\n "), k, norm(xmin - ones(1,n)),...
norm(rosenbrockGrad(x)));

plot(grads);
xlabel("step")
ylabel("norm of grad f")
xlim ([-10, 2010])
```

"codes/Newton/rosenbrockHessian.m"

```
% Hessian of the N-dimensional Rosenbrock function
function [h] = rosenbrockHessian(x)
        N = max(size(x));
        h = zeors(N,N);
        for j = 2:(N-1)
                h(j-1,j) = -400 x(j-1);
                h(j,j) = 800 x(j)^2 - 400*(x(j+1) - x(j)^2) + 202;
                h(j+1,j) = -400 x(j);
        end
        h(1,1) = 800 x(1)^2 - 400*(x(2)-x(1)^2) + 2;
        h(2,1) = -400 x(1);
        h(N,N) = 200;
        h(N-1,N) = -400 x(N-1);
end
```

The Newton method converges very fast, but it is important to mention that computing the inverse of the Hessian is quite expensive especially for great matrix dimensions. Therefore one always tries to bypass the direct computation of an inverse and instead uses an approximation which leads to quasi-newton methods. An example is described in the next section.

Matlab-Shell

```
>> testNewton
Solution:   1   1   1   1   1
Required 11 steps
Distance from true minimum: 0
Norm of gradient: 0
```

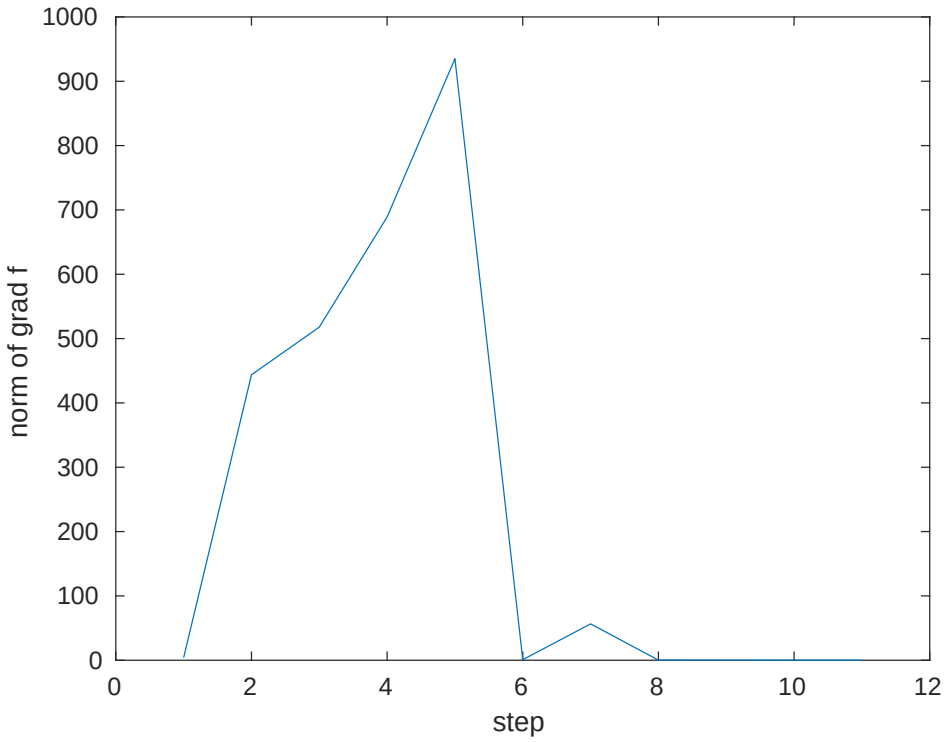


Figure 1.5: $\|\nabla f(x_k)\|$ at each step $k = 1, ..., k_{reqSteps}$ for the Newton method

1.8 BFGS-method

The Broyden-Fletcher–Goldfarb-Shanno algorithm **(BFGS)** is a numerical method for solving optimization problems without constraints. For this algorithm we do not compute the exact Hessian, but instead we use an iteratively approximated matrix H. Therefore it belongs to the Quasi-Newton-methods.

Algorithm: BFGS

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	Differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	Gradient of f
$x_0 \in \mathbb{R}^n$	Approximation of the minimum

$H_0^{-1} = I$
for $k = 0, \ldots, K_{\max}$ **do**
  $d_k \leftarrow -H_k^{-1} \nabla f(x_k)$ ▷ Compute search direction
  $t_k \leftarrow \text{linesearch}(f, d_k, x_k)$ ▷ Find step size
  $s_k \leftarrow t_k d_k$
  $y_k \leftarrow \nabla f(x_k + s_k) - \nabla f(x_k)$ ▷ Gradient delta
  $x_{k+1} \leftarrow x_k + s_k$ ▷ Perform optimization step
  $H_{k+1}^{-1} \leftarrow (I - \frac{s y_k^T}{s_k^T y_k}) H_k^{-1} (I - \frac{y_k s_k^T}{s_k^T y_k}) + \frac{s_k s_k^T}{s_k^T y_k}$ ▷ Update approx. Hessian
  if $\|s_k\| < \varepsilon$ **then return** x_{k+1} ▷ Termination condition
return $x_{K_{max}}$

"codes/BFGS/BFGS.m"

```
function [x, K] = BFGS(func, gradf, x0)
Kmax = 1000; % Maximal number of steps
eps = 1e-8;  % Disables stopping condition
N = max(size(x0));
x = x0; % Starting value
grad = gradf(x); % Compute gradient of f(x)
I = eye(N); % Identity matrix
invH = I; % Start with identity matrix as approx. Hessian

for K = 0:Kmax
    d = -invH*grad; % Compute search direction
    t = linesearch(func,d,x); % Find step size
    s = t*d;
    newGrad = gradf(x+s); % Compute new gradient
    y = newGrad-grad; % Compute gradient delta
    x = x+s; % Perform optimization step
    grad = newGrad; % Overwrite the old gradient
    invH = (I-(s*y')/(s'*y))*invH*(I-(y*s')/(s'*y))+(s*s')/(s'*y); %
        Update approx. Hessian
    if norm(s)<eps % Termination condition
        break
    end
end
end
```

"codes/BFGS/testBFGS.m"

```
N = 2;
x0 = zeros(N,1); % Iinitial value
[xmin, K] = BFGS(@rosenbrock,@rosenbrockGrad,x0); % Call BFGS

fprintf("Solution:");
disp(xmin')
```

```
fprintf("Required %i steps\nDistance from true minimum: %d\n",...
 K, norm(xmin - [1;1]));
```

With testBFGS we get the following output.

Matlab-Shell

```
>> testBFGS
Solution:    1.0000    1.0000

Required 26 steps
Distance from true minimum: 7.851116e-12
```

1.9 Penalty method : SUMT

After having discussed algorithms for unconstrained minimization problems in finite dimensions, we now want to impose additional constraints. That is, we consider for $f : \mathbb{R}^n \to \mathbb{R},\ g : \mathbb{R}^n \to \mathbb{R}^m,\ g = (g_1, ..., g_m)^T$ the problem

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} \quad f(x) \\ &\text{s.t.} \quad g(x) \leq 0 \end{aligned}$$

SUMT, standing for sequential unconstrained minimization technique, is a so-called penalty method. The idea is to reformulate the constrained problem in a way, that allows us to use one of the algorithms for unconstrained minimization repeatedly to get a sequence of solutions of unconstrained problems (hence the name) which converge to the solution for the constrained problem. This is achieved by adding a *penalty term* to the minimization function which incorporates the constraints. We consider

$$\min_{x\in\mathbb{R}^n} \quad \Theta(x, c) := f(x) + c\, P(x)$$

with P beeing a penalty function P such that

- P is continuous
- $P(x) \geq 0$
- $P(x) = 0$ if and only if $x \in S := \{x \in \mathbb{R}^n \mid g(x) \leq 0\}$.

There are different options for the penalty function and depending on this choice the algorithm will converge faster for some problems and slower for others. We take

$$P(x) = \sum_{i=1}^{m} (\max(0, g_i(x)))^2$$

The general scheme is now to minimize $\Theta(x, c)$ for fixed c, then increase c and minmize again with starting value from the previous step. For minimizing $\Theta(x, c)$ we compute the gradient

$$\nabla_x \Theta(x, c) = \nabla f(x) + 2c \sum_{i=1}^{m} (\max(0, g_i(x))\, \nabla g_i(x))$$

Then, the SUMT algorithm is given by

Algorithm: SUMT

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	the gradient of f
$g : \mathbb{R}^n \to \mathbb{R}^m$	differentiable function of constraints
$\nabla g : \mathbb{R}^n \to \mathbb{R}^{m \times n}$	the Jacobian of g
$c_0 \in \mathbb{R}$	starting value for penalty parameter
$\eta \in \mathbb{R}$	increasing factor for penalty parameter
$x_0 \in \mathbb{R}^n$	starting value

Output:

$x \in \mathbb{R}^n$	approximation of the constrained minimum

1: $\Theta(x, c) := f(x) + c\, g(x)$
2: $\nabla_x \Theta(x, c) := \nabla f(x) + c \nabla g(x)$
3: **for** $k = 0, \ldots, K_{\max}$ **do**
4: $\quad x_{k+1} \leftarrow \min_{x \in \mathbb{R}^n} \Theta(x, c_k)$ ▷ solve unconstrained problem, starting value x_k
5: $\quad c_{k+1} \leftarrow \eta\, c_k$ ▷ increase penalty parameter
6: $\quad$ **if** $\|x_{k+1} - x_k\| < \varepsilon$ **then return** x_{k+1} ▷ stopping condition
7: **return** $x_{K_{max}}$

A major problem is the stopping condition: As we will see in the test, it may very well occur that for several steps, x_k does not change at all but for some larger c, i.e. some later step, it does. Therefore the presented stopping condition was included more for completeness than for practical use. Alternatively, one can just take a fixed number of steps or come up with a more clever stopping condition like for example considering not just the last one but the last 10 steps and only stop if there was no change in all of those.

In the implementation we use use nonlinear conjugate gradient to solve the unconstrained minimization problem. Of course, any other minimization algorithm will work as well.

"codes/SUMT/sumt.m"

```
function [x, k] = sumt(f, gradf, g, gradg, x0)

eta = 1.5;% Increasing factor for penalty parameter
c = 1.5;    % Starting value for penalty parameter

Kmax = 50;% Maximal number of steps
eps = -1; % Disables stopping condition

x = x0;
xprev = x;

for k = 0:Kmax
    % Define penalized function
    thetaC = @(x) f(x) + c*sum(max(0,g(x)));
```

```
    % Compute gradient of g only where g(x) >= 0
    h = gradg(x).*(g(x) >= 0);

    % Compute gradient of penalized function
    gradThetaC = @(x) (gradf(x) + [2*c*sum(g(x).*h(:,1));...
      2*c*sum(g(x).*h(:,2))]);

    % Solve unconstrained minimization problem
    x = nonlinearCG(thetaC,gradThetaC, x, 100, 1e-8);
    % Alternatively use steepestDescent(thetaC,gradThetaC, xold, 500,
        1e-8)
    % or use any unconstrained minimization algorithm

    if norm(xprev - x) < eps % Stopping condition
      break;
    end

    c = eta*c; % Increase penalty parameter
    xprev = x;
end
end
```

We test our code with a quadratic polynomial in $\mathbb{R}^2$ and linear constraints.

"codes/SUMT/f.m"

```
function [fx] = f(x)
    fx = (x(1) - 6)^2 + (x(2) - 7)^2;
end
```

"codes/SUMT/gradf.m"

```
function [gfx] = gradf(x)
    gfx = [2*(x(1) - 6); 2*(x(2) - 7)];
end
```

"codes/SUMT/g.m"

```
function [gx] = g(x)
    gx = zeros(4, 1);
    gx(1) = -3*x(1) - 2*x(2) + 5;
    gx(2) = -x(1) + x(2) - 3;
    gx(3) = x(1) + x(2) - 7;
    gx(4) = (2/3)*x(1) - x(2) - 4/3;
 end
```

"codes/SUMT/gradg.m"

```
function gradgx = gradg(x)
  gradgx = [-3, -2; -1, 1; 1,1;2/3, -1];
end
```

"codes/SUMT/testSUMT.m"

```
x0 = [6;7];

[xmin, k] = sumt(@f, @gradf, @g, @gradg, x0);

fprintf("Solution:");
disp(xmin')
fprintf("Required %i steps\nDistance from true minimum: %d\n",...
  k, norm(xmin - [3;4]));
```

This produces the following output:

Matlab-Shell

```
>> testSUMT
Solution:    3.0000    4.0000
Required 50 steps
Distance from true minimum: 9.930137e-16
```

By plotting the norm of the steps, we see the phenomenon described above: for several steps there is no change in x_k and only for a larger c does x_k jump to a better solution. If we would have used the stopping condition, the algorithm would have terminated after only 6 steps and produced a significantly worse solution.

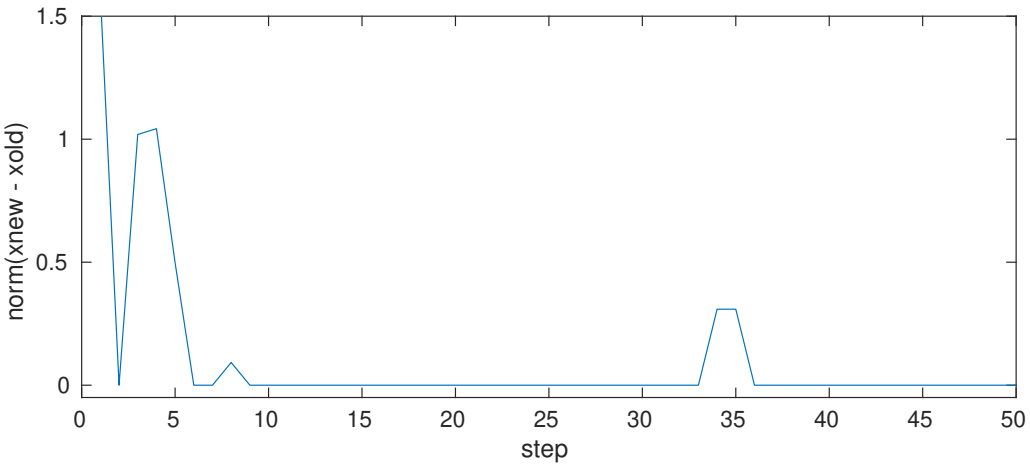


Figure 1.6: Step sizes for SUMT algorithm

1.10 Barrier method

As for the penalty method dicussed before, we want to solve the constrained optimization problem

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} \ f(x) \\ &\text{s.t. } \ g(x) \leq 0 \end{aligned}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, $g = (g_1, ..., g_m)^T$ are given differentiable mappings. Now, the idea for the barrier method is to instead optimize another function B, which converges to infinity against the boundary of the feasible set. Furthermore, B depends on a scaling parameter r, and is assumed to converge to the function

f as r goes to zero. Then the algorithm consists of iteratively optimizing B and decreasing r until a stationary point is reached.

In general, the barrier B has the shape

$$B(x, r) \;=\; f(x) \;+\; r\, h(x),$$

where $h : \mathbb{R}^n \to \mathbb{R}$ is a barrier function depending on g. For h we considered

$$h(x) = -\sum_{i=1}^{m} \frac{1}{g_i(x)} \qquad \text{inverse barrier,}$$

$$h(x) = -\sum_{i=1}^{m} \log(-g_i(x)) \qquad \text{logarithmic barrier,}$$

although several other options are thinkable. Note, that these terms are well-defined, if $g(x) < 0$. However, on the boundary of the admissible set they imply the desired divergence of the barrier B. Their respective gradients are given by

$$\nabla h(x) = \sum_{i=1}^{m} \frac{\nabla g_i(x)}{g_i(x)^2} \qquad \text{inverse barrier gradient,}$$

$$\nabla h(x) = -\sum_{i=1}^{m} \frac{\nabla g_i(x)}{g_i(x)} \qquad \text{logarithmic barrier gradient.}$$

Algorithm: Barrier method

Input:

$f : \mathbb{R}^n \to \mathbb{R}$	differentiable function
$\nabla f : \mathbb{R}^n \to \mathbb{R}^n$	gradient of f
$g : \mathbb{R}^n \to \mathbb{R}^m$	differentiable function of constraints
$\nabla g_i : \mathbb{R}^n \to \mathbb{R}^n$	for $i = 1, \ldots, m$, gradients of g
$h : \mathbb{R}^n \to \mathbb{R}$	barrier function
$r \in \mathbb{R}$	starting value for barrier scaling
$\eta \in \mathbb{R}$	decreasing factor for barrier scaling
$x_0 \in \mathbb{R}^n$	starting value

Output:

$x \in \mathbb{R}^n$	approximation of the constrained minimum

1: $B(x, r) := f(x) + r\, h(x)$
2: $\nabla_x B(x, r) := \nabla f(x) + r\, \nabla h(x)$
3: **for** $k = 0, \ldots, K_{\max}$ **do**
4: $\quad x_{k+1} \leftarrow \min_{x \in \mathbb{R}^n} B(x, r_k)$ ▷ solve unconstrained problem, starting value x_k
5: $\quad r_{k+1} \leftarrow \eta\, r_k$ ▷ decrease scaling parameter
6: $\quad$ **if** $\|x_{k+1} - x_k\| < \varepsilon$ **then return** x_{k+1} ▷ stopping condition
7: **return** $x_{K_{max}}$

For the implementation we used a nonlinear conjugate gradient method to optimize the function, but any gradient based optimization scheme would suffice. Note that

the parameter α in our linesearch has been adapted to increase stability. Moreover, the barrier B is set to infinity outside of the admissible set to prevent the linesearch in our gradient based algorithms from jumping over the boundary of the feasible set. If the minimum of f lies on the boundary of the admissible set, the problem becomes numerically unstable, since the minimum of B converges to the boundary as r approaches zero. On the other hand B diverges to infinity against the boundary. Therefore, the problem is prone to oscillation with increasing number of iterations. To compensate we included an additional stopping criterion. As an example we tested the code for the functions

$$f : \mathbb{R}^2 \to \mathbb{R}, \quad x \mapsto 2\,x_1^2 \;+\; 9\,x_2$$
$$g : \mathbb{R}^2 \to \mathbb{R}, \quad x \mapsto 4 \;-\; x_1 \;-\; x_2$$

with the analytic solution $x^* = (2.25,\, 1.75)$.

"codes/BarrierMethod/linesearch.m"

```
%{
Line search algorithm with Armijo condition

For a given function f and direction p at a point x,
finds a step size t such that f admits a sufficient decrease in
    direction p at x

Terminates with step size approximately 1e-5 if no sufficient
    decrease is found.
%}
function [t] = linesearch(f, p, x)
    alpha = 0.00005; % Specifies desired decrease
    t = 1;
    while f(x +t*p) > f(x) - alpha*t*power(norm(p), 2) % Armijo
        condition for not accepting t
        t = 0.5*t; % Backtracking step
        if t < 1e-5 % No decrease found
            return;
        end
    end
end
```

"codes/BarrierMethod/barrierMethod.m"

```
%the barrier method minimizes the function f(x) s.t. g(x)<=0
%inputs: functions f and g, their gradients gradf and gradg, and the
    initial value x0 for x
%outputs: solution x, required number of steps k, changes of the
    solutions dx with each iteration
function [x, k, dx] = barrierMethod(f, gradf, g, gradg, x0)

%%%initialization

r = 10;              %scaling factor of the barrier
eta = .5;            %decrease of the barrier scaling
Kmax = 100;          %maximal number of iterations
eps = 1e-4;          %threshold for the change of x to break
    iterations
x = x0;              %initial value for x
xold = x;            %save old value for x
```

```
dx = zeros(1,Kmax); %save changes in the solution

for k = 1:Kmax

    %specify either inverse or log barrier in barrier.m and
        gradBarrier.m
    B = @(x) barrier(f, g, r, x);
    gradB = @(x) gradBarrier(gradf, g, gradg, r, x);

    %optimization of the barrier function
    %use e.g. nonlinear CG or steepest descent
    x = nonlinearCG(B,gradB, xold, 1000, 1e-4);
    %x = steepestDescent(B,gradB, xold, 1000, 1e-4);

    dx(k) = norm(x-xold); %save difference of old and new solution

    if dx(k) < eps %break, if the difference in the solutions is
        small enough
      break

    %break, if the solution starts oscillating,
    %or if the gradient descent doesn't yield valid results
    elseif k > 1 && dx(k)>dx(k-1) || sum(isnan(x))>0
      fprintf('Stability warning: solution starts oscillating!\n')
      x=xold;
      k=k-1;
      break
    end

    r = eta*r; %decrease barrier scaling parameter
    xold = x;
end
```

”codes/BarrierMethod/barrier.m”

```
function Bx = barrier(f, g, r, x)
%%use either inverse or log barrier
if g(x) <= 0 %then x in admissible set
  %Bx = f(x) + sum(r*-1./g(x)); %inverse barrier
  Bx = f(x) - r*sum(log(-g(x))); %log barrier
else
  Bx = inf;  %sets the barrier outside the feasible set to infinity
      for numerical stability
end
end
```

”codes/BarrierMethod/gradBarrier.m”

```
function dB = gradBarrier(gradf, g, gradg, r, x)
%%use either inverse or log barrier
%dB = gradf(x) + r*sum(g(x)'.^(-2)*gradg(x),2); %gradient inverse
    barrier
dB = gradf(x) - r*sum(1./g(x)'*gradg(x),2); %gradient log barrier
end
```

”codes/BarrierMethod/barrierMethodTest.m”

```
%define the functions f and g, and their gradients
f = @(x) 2*x(1)^2+9*x(2);
gradf = @(x) [4*x(1);9];

g = @(x) -x(1)-x(2)+4;
gradg = @(x) [-1;-1];

x0 = [4;4]; %starting value for x

[x,k,dx] = barrierMethod(f, gradf, g, gradg, x0);

error = norm(x-[2.25;1.75]);

fprintf("Solution: ")
disp(x')
fprintf("Required %d steps \nDistance from true minimum: %f\n",k,
    error)

figure
plot(1:k, dx(1:k))
xlabel('step')
ylabel('dx')
title('change of x')
set(gca, 'YScale', 'log')
```

We get the following output for the inverse barrier:

Matlab-Shell

```
>> barrierMethodTest
Solution:     2.2500    1.7503
Required 25 steps
Distance from true minimum: 0.000268
```

And the following for the logarithmic barrier:

Matlab-Shell

```
>> barrierMethodTest
Solution:     2.2500    1.7501
Required 15 steps
Distance from true minimum: 0.000069
```

The algorithm utilizing the logarithmic barrier converged faster and with more accuracy in our example. Between the iterations, the numerical minima of B converged steadily to the minimum of f as could be seen by the output of the step-sizes dx.

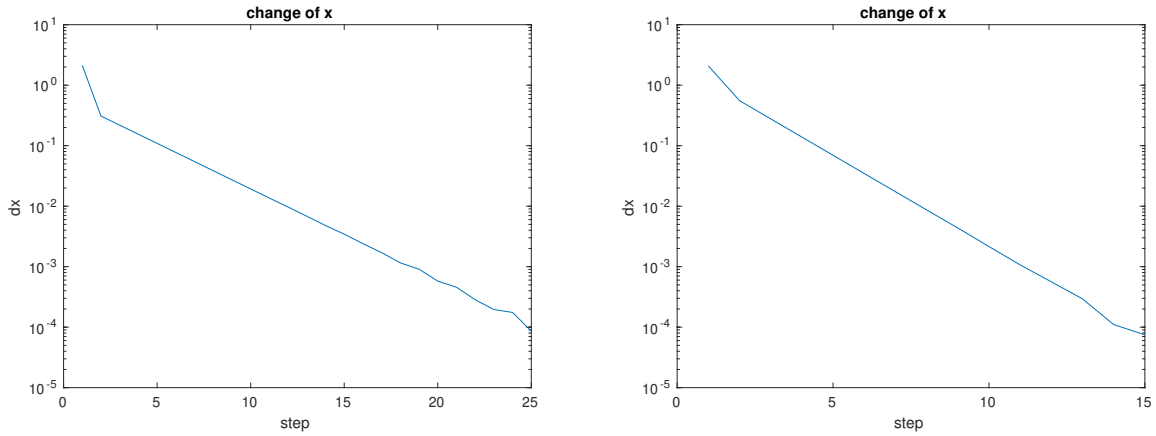


Figure 1.7: Step-sizes for barrier method with inverse barrier (left) and logarithmic barrier (right)

2 Calculus of variations

In this chapter we talk about solving variations problems numerically. We will minimize the value of a functional with the direct and indirect method. We consider a variation problem in the form

$$\begin{aligned} \min_{y\in V}\ J(y) &:= \int_a^b l\left(x, y(x), y'(x)\right) dx \\ \text{with}\quad V &:= \{v \in C^1[a,b] \mid v(a) = y_a,\ v(b) = y_b\} \end{aligned}$$

with $l \in C^2$.

2.1 Direct method

For the direct method we discretize the variation problem. Consequently we obtain a finite-dimensional variation problem. Now we can use methods for non-linear minimization problem. We already introduced some non-linear minimization methods in chapter 1.

For the direct method we observe the minimization problem

$$\begin{aligned} \min_{y\in V}\ J(y) &:= \int_1^2 \left(\frac{1}{2}\, y^2 \ +\ \frac{1}{2}\,(y')^2\right) dx \\ \text{with}\quad V &:= \{v \in C^1[1,2] \mid v(1) = 1,\ v(2) = 4\} \end{aligned}$$

First we discretize x on the given interval $I = [1, 2]$. We divide our interval I into N subintervals with step size $h = \frac{1}{N}$. With $x_j = jh$ for $j = 0, \ldots, N$ and $y_j = y(x_j)$ for $j = 1, \ldots, N-1$ we can discretize the functional J. Note that y_0 and y_N are fixed by the interval bounds. To illustrate that we talk about the numerical solution we write y^h.

$$J_h(y^h) = h \sum_{j=1}^{N} l\left(\frac{x_{j-1} + x_j}{2}, \frac{y_{j-1} + y_j}{2}, \frac{y_j - y_{j-1}}{h}\right)$$

For our given example we get

$$J_h(y^h) = h \sum_{j=1}^{N} \left(\frac{1}{2}\left(\frac{y_{j-1} + y_j}{2}\right)^2 + \frac{1}{2}\left(\frac{y_j - y_{j-1}}{h}\right)^2\right)$$

as discretized functional.
Now we compute the gradient of the descretized functional

$$\frac{\partial J_h(y^h)}{\partial y_j} \ = \ \frac{h}{2}\left(\left.\frac{\partial l}{\partial y}\right|_j + \left.\frac{\partial l}{\partial y}\right|_{j+1}\right) + \left.\frac{\partial l}{\partial y'}\right|_j - \left.\frac{\partial l}{\partial y'}\right|_{j+1}$$

with $l|_j = l\left(\frac{x_{j-1}+x_j}{2}, \frac{y_{j-1}+y_j}{2}, \frac{y_j - y_{j-1}}{h}\right)$.

For our example we obtain

$$\frac{\partial J_h(y^h)}{\partial y_j} = \frac{h}{2}\left(\frac{y_{j-1}+y_j}{2}+\frac{y_j+y_{j+1}}{2}\right)+\frac{y_j-y_{j-1}}{h}-\frac{y_{j+1}-y_j}{h}$$

for $j = 1, \ldots, N-1$.

Now we can use the nonlinear CG-method which was discussed in chapter 1.5 to solve this minimization problem.

"codes/CVDirectLinear/DirectCV.m"

```
a = 1; % Intervall I=[a,b]
b = 2;

N = 20;
h = (b-a)/N; % Step size

x= (a:h:b)'; % Discretized x-vector with step size h
ya = 1; % Fixed boudaries
yb = 4;
jhReduced = @(y) Jh(x,[ya; y ; yb]); % Store function Jh(x,y) in a
    function handle with argument y
gradJhReduced = @(y) gradJdy(x, [ya; y ; yb]); % Store function
    gradJdy(x,y) in a function handle with argument y

y0 = zeros(N-1,1); % Start value

ymin = nonlinearCG(jhReduced, gradJhReduced, y0); % Call function
    nonlinearCG with fucntion and gradient of the corresponding
    example and start value y0
```

"codes/CVDirectLinear/l.m"

```
function ret = l(x, yx, ypx)
    ret=0.5*(ypx^2+yx^2);
end
```

"codes/CVDirectLinear/Jh.m"

```
function sum = Jh(x,y)
    N = length(x);
    h = x(2) - x(1);
    sum = 0;
    for j = 2:N
        sum = sum + l(0.5*(x(j-1) + x(j)), 0.5*(y(j-1) + y(j)), (y(j)
            - y(j-1))/h);
    end
    sum = sum*h;
end
```

"codes/CVDirectLinear/gradJdy.m"

```
function grad = gradJdy(x, y)
    N = length(x);
```

```
    h = x(2) - x(1);
    grad = zeros(N-2,1);
    for j = 2:N-1
        grad(j-1) = h/2 * dldy(0.5*(x(j-1) + x(j)), 0.5*(y(j-1) + y(j
            )) , (y(j) - y(j-1))/h ) + ...
                    h/2 * dldy( 0.5*(x(j+1) + x(j)) , 0.5*(y(j+1) + y
                        (j)) , (y(j+1) - y(j))/h ) + ...
                    dldyp(0.5*(x(j-1) + x(j)) , 0.5*(y(j-1) + y(j)) ,
                         (y(j) - y(j-1))/h ) - ...
                    dldyp( 0.5*(x(j+1) + x(j)) , 0.5*(y(j+1) + y(j))
                        , (y(j+1) - y(j))/h );
    end
end
```

"codes/CVDirectLinear/dldy.m"

```
function ret = dldy(x, yx, ypx)
    ret = yx;
end
```

"codes/CVDirectLinear/dldyp.m"

```
function ret = dldyp(x, yx, ypx)
    ret = ypx;
end
```

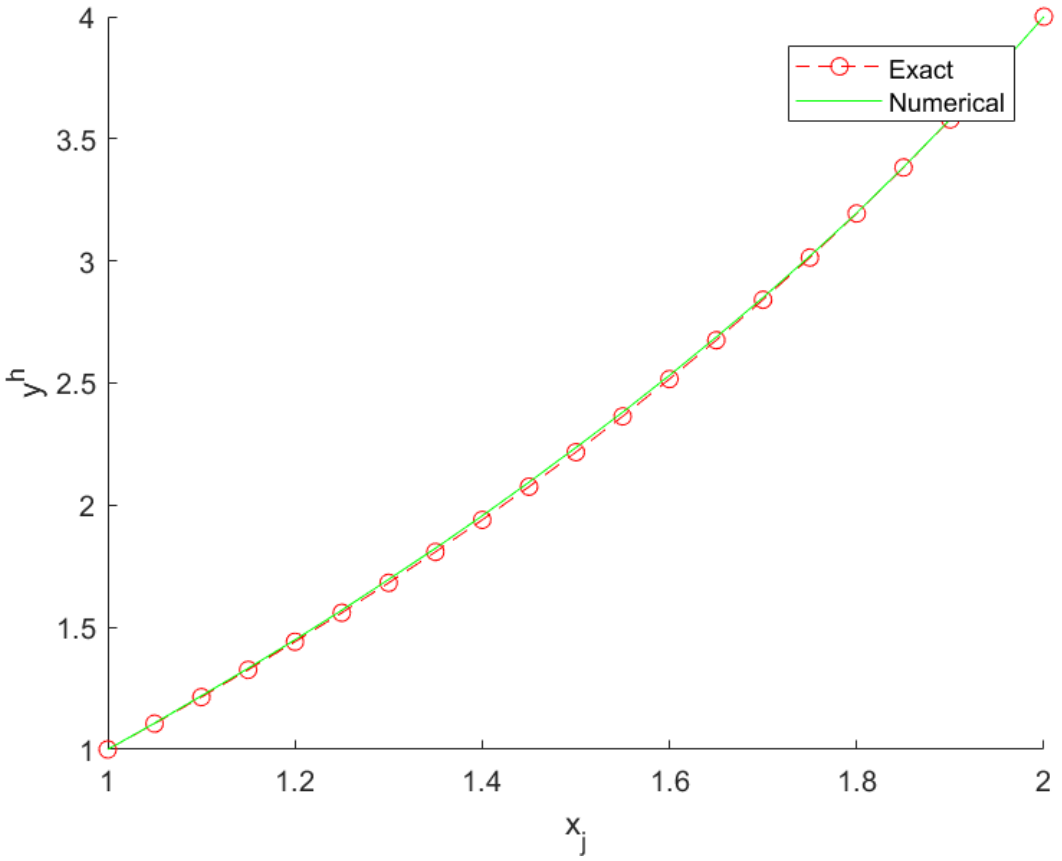


Figure 2.1: Numerical solution with DirectCV and exact solution

The line diagram (figure 2.1) shows the numerical approximated solution for all x_j and the exact solution for the interval $I = [1, 2]$. The numerical and exact solution differs especially in the centre of the interval. This is due to the start solution y_0. We have only information about the boundaries y_a and y_b therefore we get the best approximation by the interval bounds.

2.2 Indirect method with linear ∇J

Unlike in the direct method for calculus of variations where we minimized directly on the discretized function space, another possibility is to indirectly find the minimizing function by solving the Euler-Lagrange equations.
The example problem is the same as before:

$$\min_{y \in V} J(y) := \int_0^1 \left(\frac{1}{2} y^2 + \frac{1}{2} (y')^2 \right) dx$$
$$\text{with} \quad V := \{ v \in C^1[0,1] \mid v(0) = 1,\ v(1) = 4 \}$$

and thus $l(x, y, y') := \frac{1}{2} y(x)^2 + \frac{1}{2} (y'(x))^2$. We discretize the interval [0,1] by N equidistant points $x_k = \frac{k-1}{N-1}$ with step size $h := \frac{1}{N-1}$. Now, there are in principle

two possible approaches:

a) optimize-before-discretize:

We could first write the Euler-Lagrange equation

$$\frac{\partial l}{\partial y} - \frac{\mathrm{d}}{\mathrm{d}x}\frac{\partial l}{\partial y'} = 0 \quad \Leftrightarrow \quad y - y'' = 0$$

and then discretize y to be a vector $y = (y_k)_{k=1,...,N}$ where $y_k = y(x_k)$. Using the forward Euler-method the second derivative of y at the grid point x_k is then given by

$$y''(x_k) \;=\; (y'')_k \;=\; \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2}$$

for $k = 2, ..., N-1$. The discretized Euler-Lagrange equation is thus given by

$$y_k - \frac{1}{h^2}\left(y_{k+1} - 2y_k + y_{k-1}\right) = 0 \qquad \text{for} \quad k = 2, ..., N-1$$

with boundary conditions

$$y_1 = 1 \qquad y_N = 4.$$

Since this is just a linear system of equations we can write it in matrix form

$$\left(\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{pmatrix} - \frac{1}{h^2}\begin{pmatrix} 0 & & \dots & & 0 \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ 0 & & \dots & & 0 \end{pmatrix}\right) \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 4 \end{pmatrix}$$

and easily solve (numerically) for $(y_k)_{k=1,...,N}$ since the left-hand side is invertible.

b) discretize-before-optimize:

The second possiblity is to first discretize the functional J and then derive the corresponding Euler-Lagrange equations by numerically computing the gradient of J. Using the discretization $y = (y_k)_{k=1,...,N}$ with $y_k = y(x_k)$ and the midpoint-method for numerical integration, the functional J is discretized by

$$J_h((y_k)_k) \;=\; h\sum_{j=2}^{N} l\left(\frac{1}{2}(x_{j-1} + x_j), \frac{1}{2}(y_{j-1} + y_j), \frac{1}{h}(y_j - y_{j-1})\right) \tag{2.1}$$

By abbreviating

$$\left.\frac{\partial l}{\partial y}\right|_j := \frac{\partial l}{\partial y}\left(\frac{1}{2}(x_{j-1} + x_j), \frac{1}{2}(y_{j-1} + y_j), \frac{1}{h}(y_j - y_{j-1})\right)$$

and analogous for $\left.\frac{\partial l}{\partial y'}\right|_j$ we derive 2.1 for y_j to get

$$\begin{aligned} \frac{\partial J_h(y)}{\partial y_j} \;&=\; \frac{h}{2}\left(\left.\frac{\partial l}{\partial y}\right|_j + \left.\frac{\partial l}{\partial y}\right|_{j+1}\right) + \left.\frac{\partial l}{\partial y'}\right|_j - \left.\frac{\partial l}{\partial y'}\right|_{j+1} \\ &=\; h\left(\frac{1}{2}\left(\left.\frac{\partial l}{\partial y}\right|_j + \left.\frac{\partial l}{\partial y}\right|_{j+1}\right) - \frac{1}{h}\left(\left.\frac{\partial l}{\partial y'}\right|_{j+1} - \left.\frac{\partial l}{\partial y'}\right|_j\right)\right) \end{aligned}$$

since only two terms in the sum remain. Because we know that in general the gradient of J is given by

$$(\nabla J(y), h) = \int_a^b h \cdot \left(\frac{\partial l}{\partial y} - \frac{\mathrm{d}}{\mathrm{dx}} \frac{\partial l}{\partial y'} \right) dx$$

this yields the discretized Euler-Lagrange equation

$$\frac{1}{2} \left(\left. \frac{\partial l}{\partial y} \right|_j + \left. \frac{\partial l}{\partial y} \right|_{j+1} \right) - \frac{1}{h} \left(\left. \frac{\partial l}{\partial y'} \right|_{j+1} - \left. \frac{\partial l}{\partial y'} \right|_j \right) \quad = \quad 0 \quad \text{for} \quad j = 2, ..., N-1.$$

Computing $\frac{\partial l}{\partial y} = y$ and $\frac{\partial l}{\partial y'} = y'$ we obtain with our abbreviations $\left. \frac{\partial l}{\partial y} \right|_j = \frac{1}{2}(y_j + y_{j-1})$ and $\left. \frac{\partial l}{\partial y'} \right|_j = \frac{1}{h}(y_j - y_{j+1})$ the explicit system of linear Euler-Lagrange equations

$$\frac{1}{2} \left(\frac{y_{j+1} + 2y_j + y_{j+1}}{2} \right) - \frac{1}{h} \left(\frac{y_{j+1} - 2y_j + y_{j+1}}{h} \right) \quad \text{for} \quad j = 2, ..., N-1$$

with boundary conditions

$$y_1 = 1 \qquad y_N = 4.$$

Writing this sytem of linear equations again in matrix form

$$\left(\frac{1}{4} \begin{pmatrix} 4 & & \dots & & 0 \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ 0 & & \dots & 0 & 4 \end{pmatrix} - \frac{1}{h^2} \begin{pmatrix} 0 & & \dots & & 0 \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ 0 & & \dots & & 0 \end{pmatrix} \right) \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 4 \end{pmatrix}$$

we can easily solve numerically for (y_k).

One can easily verify that the exact solution to the Euler-Lagrange equation is given by

$$y(x) = c_1 \exp(x) + c_2 \exp(-x)$$

where the boundary conditions determine $(c_1, c_2)^T$ uniquely by

$$\begin{pmatrix} e^1 & e^{-1} \\ e^2 & e^{-2} \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \end{pmatrix}.$$

The implementation is rather simple since it only amounts to solving a system of linear equations. For comparison, we plot the exact solution as well as the deviation from the exact solution.

"codes/CVIndirectLinear/CVIndirectLinear.m"

```
a = 1;
b = 2;
ya = 1;
yb = 4;

N = 100; % Mesh size

h = (b-a)/(N-1);
x = linspace(a,b,N)';
```

```
B = gallery('tridiag', N, 1,-2,1);
B(1,:) = zeros(1,N);
B(N,:) = zeros(1,N);

% Generate matrix for discretize before optimize
ADBO = gallery('tridiag', N, 1,2,1);
a = zeros(1,N);
a(1) = 4;
ADBO(1,:) = a;
a = zeros(1,N);
a(N) = 4;
ADBO(N,:) = a;
CDBO = (1/4)*ADBO  - 1/power(h, 2)*B;

% Generate matrix for optimize before discretize:
AOBD = 4*eye(N);
COBD = (1/4)*AOBD  - 1/power(h, 2)*B;

% Generate right-hand side of equation
r = zeros(N,1);
r(1) = ya;
r(N) = yb;

solDBO = CDBO \ r; % Solve DBO system of equations
solOBD = COBD \ r; % Solve OBD system of equations

% Compute exact solution for comparison
c = [exp(1) , exp(-1) ; exp(2) , exp(-2)] \ [1;4];
solExact = c(1)*exp(x) + c(2)*exp(-x);


figure('Position', [10 10 700 450])
clf
%plot(x, exactY, 'r--o', "MarkerIndices", 1:5:length(exactY));
% Plot all three solutions
subplot(2,1,1)
hold on
plot(x, solExact, 'r--');
plot(x, solDBO,'g-.' )
plot(x, solOBD,'b' )
legend("Exact","Numerical DBO", "Numerical OBD")
title("Solutions")
hold off% Error plot for DBO and OBD
subplot(2,1,2)
hold on
plot(x, solExact -solDBO);
plot(x, solExact -solOBD);
legend("DBO","OBD")
title("Errors")
ylim([-3e-6, 6e-6]);
```

This gives:

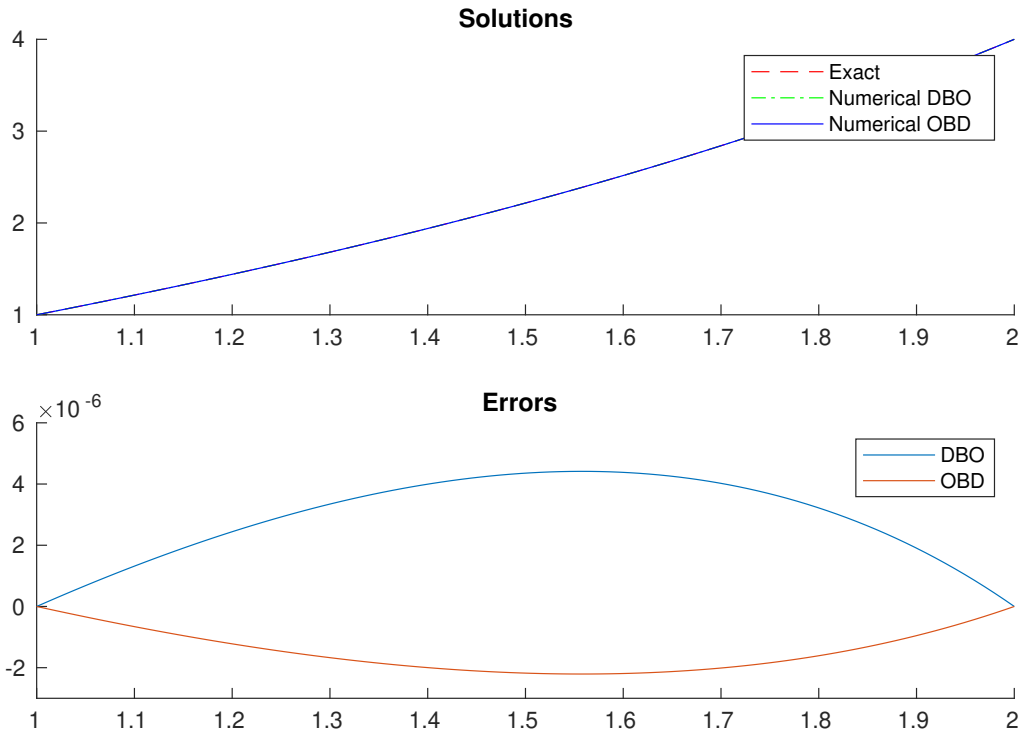


Figure 2.2: Solutions and errors for indirect method for CV with dbo and obd

We see that both methods have errors of the same magnitude with OBD having a slightly smaller error in this case.

Since the gradient ∇J was linear, solving the Euler-Lagrange equations meant in this case just solving a system of linear equations, for both DBO and OBD. For nonlinear ∇J we can apply the same techniques as above but will then get a system of nonlinear equations. In this case one could use a Newton method to approximate a solution.

3 Optimal control

3.1 Optimal control of linear ODEs with initial conditions

As a first example of an optimal control problem we consider the control of the following tracking functional

$$\begin{aligned} \min_{u\in L^2(\Omega)} \; J(y,u) \; &:= \frac{1}{2}\|y-y_d\|^2_{L^2(\Omega)} \; + \; \frac{\nu}{2}\|u\|^2_{L^2(\Omega)} \\ y' \; &= \; y \; + \; u \quad \text{s.t.} \quad y(0) \; = \; 1 \end{aligned}$$

with $\Omega = (0,1)$, $\nu > 0$ and target trajectory $y_d = 2 \cdot \chi_{(\frac{1}{2},1)}$. The corresponding optimality system reads as follows:

$$\begin{aligned} y' \; &= \; y \; + \; u && \text{s.t.} \quad y(0) \; = \; 1 && \text{state equation,} \\ -p' \; &= \; p \; - \; (y-y_d) && \text{s.t.} \quad p(1) \; = \; 0 && \text{adjoint equation,} \\ 0 \; &= \; \nu\, u \; - \; p && && \text{optimality condition.} \end{aligned}$$

By solving the state equation we obtain the implicit dependence of y on u and thereby reduce the optimization problem to the minimization of the reduced functional

$$\min_{u\in L^2(\Omega)} \; \hat{J}(u) \; := \; J\left(y(u),u\right).$$

The gradient of $\hat{J}$ is given by the optimality condition, yielding

$$\nabla \hat{J}(u) \; = \; \nu\, u \; - \; p(u).$$

Note how p implicitly depends on y by the adjoint equation, which in turn implicitly depends on u as mentioned already. Thus one deduces the implicit dependence of p on u.

For the numerical computation of the state and adjoint equations we use explicit Euler methods, utilizing forward and backward differences respectively. Since the adjoint equation is evolving backwards in time we will consider the algorithm more closely in the following. First we discretize Ω as a mesh $(x_k)_{k=1,\dots,N}$ with step size $h = \frac{1}{N-1}$, i.e., $x_k = \frac{k-1}{N-1}$. Now the terminal condition of p is given as $p(x_N) = 0$ and from the approximation $p'(x_k) \approx \frac{1}{h}(p(x_k) - p(x_{k-1}))$ we iteratively calculate

$$p(x_{k-1}) \; = \; p(x_k) \; - \; h \cdot (-p(x_k) \; + \; (y(x_k) - y_d(x_k))),$$

for all $k = N, \dots, 2$ to obtain a discretized p. The gradient $\nabla \hat{J}$ is then derived with the above formula. For the minimization step we can use any gradient-based optimization scheme, such as steepest descent or nonlinear conjugate gradient methods. Note, however, that we adapted the parameters of our linesearch to increase the rate of convergence.

”codes/OPC/linesearch.m”

```
%{
Line search algorithm with Armijo condition

For a given function f and direction p at a point x,
finds a step size t such that f admits a sufficient decrease in
    direction p at x

Terminates with step size approximately 1e-5 if no sufficient
    decrease is found.
%}
function [t] = linesearch(f, p, x)
    alpha = 0.00005; % Specifies desired decrease
    t = 4;
    while f(x +t*p) > f(x) - alpha*t*power(norm(p), 2) % Armijo
        condition for not accepting t
        t = 0.5*t; % Backtracking step
        if t < 1e-5 % No decrease found
            return;
        end
    end
end
```

”codes/OPC/targetTrajectory.m”

```
%receives an input vector x and calculates the corresponding values
%outputs values of the target trajectory as a vector
function y=targetTrajectory(x)
  N=length(x);
  y=zeros(N,1);
  for i=1:N
    if x(i)<1/2
      y(i)=0;
    else
      y(i)=2;
    end
  end
end
```

”codes/OPC/solveStateEq.m”

```
%numerically solves the state equation with initial condition y(a)=ya
%by explicit Euler
function y=solveStateEq(x,u,ya)
  N=length(x);
  y=zeros(N,1);
  y(1)=ya;
  dx=x(2)-x(1);
  for i=2:N
    y(i)=dx*stateEq(x(i-1),y(i-1),u(i-1))+y(i-1);
  end
end
```

”codes/OPC/stateEq.m”

```
%calculates the state equation given by the functional J
function dy=stateEq(x,y,u)
  dy=y+u;
end
```

"codes/OPC/solveAdjointEq.m"

```
%numerically solves backwards evolving adjoint equation by explicit
    Euler
%with terminal condition p(b)=pb for p given by the functional
function p=solveAdjointEq(x,y,u,pb)
  N=length(x);
  dx=x(2)-x(1);
  p=zeros(N,1);
  p(N)=pb;
  for j=1:N-1
    i=N-j;
    p(i)=-dx*adjointEq(x(i+1),y(i+1),u(i+1),p(i+1))+p(i+1);
  end
end
```

"codes/OPC/adjointEq.m"

```
%calculates the adjoint equation given by the functional J
function dp=adjointEq(x,y,u,p)
  dp=-p+y-targetTrajectory(x);
end
```

"codes/OPC/OPC.m"

```
%solves the optimal control problem for the functional J(y,u)
%J(y,u)=|y-y_d|^2+1/2*nu*|u|^2
%y is the state, u is the control, y_d is the target trajectory
%solves the optimization problem via gradient descent

%%%initialization

a=0;            %lower interval boundary
b=1;            %upper interval boundary
ya=1;           %initial value for y
N=1e2;          %number of partition points of the interval
dx=(b-a)/N;     %determine step size of the mesh
x=[a:dx:b];     %generate mesh
u=zeros(N+1,1); %initialize the control u
Kmax=5e3;       %maximal number of iterations
eps=1e-2;       %threshold for gradient magnitude to break iterations

%cost of the penalty term; decrease nu to increase accuracy of
%approximation to target trajectory
nu=1e-3;

%generate function for the functional J; receives vectors as inputs
J = @(y,u) 1/2*dx*sum((y-targetTrajectory(x)).^2)+nu/2*dx*sum(u.^2);
%generate reduced functional J_hat
J_hat =  @(u) J(solveStateEq(x,u,ya),u);
%generate gradient of J_hat; grad_J_hat = -p+nu*u
```

```
grad_J_hat = @(u) -solveAdjointEq(x,solveStateEq(x,u,ya),u,0)+nu*u;

%calculation via e.g. nonlinearCG, steepestDescent
[u,k,grads]=nonlinearCG(J_hat,grad_J_hat,u,Kmax,eps);
%alternatively use:
%[u,k,grads]=steepestDescent(J_hat,grad_J_hat,u,Kmax,eps);

%calculate y from the optimal control u
y=solveStateEq(x,u,ya);

figure
hold on
subplot(3,1,1)
plot(x,[y,targetTrajectory(x)])
title('y')
lgd=legend('solution', 'target trajectory');
lgd.Location='northwest';
subplot(3,1,2)
plot(x,u)
title('u')
subplot(3,1,3)
plot(1:k,grads(1:k))
title('norm grad J hat')
```

The algorithm terminates after 91 steps. After 20 steps the approximation of y to the minimum is already fairly accurate. If the parameter ν is decreased, y approaches the target trajectory more closely, but the control u diverges at the points 0 and $\frac{1}{2}$. This divergence is typical for the forced discrepancy between the initial conditions for y and y_d, as well as the discontinuity of y_d .

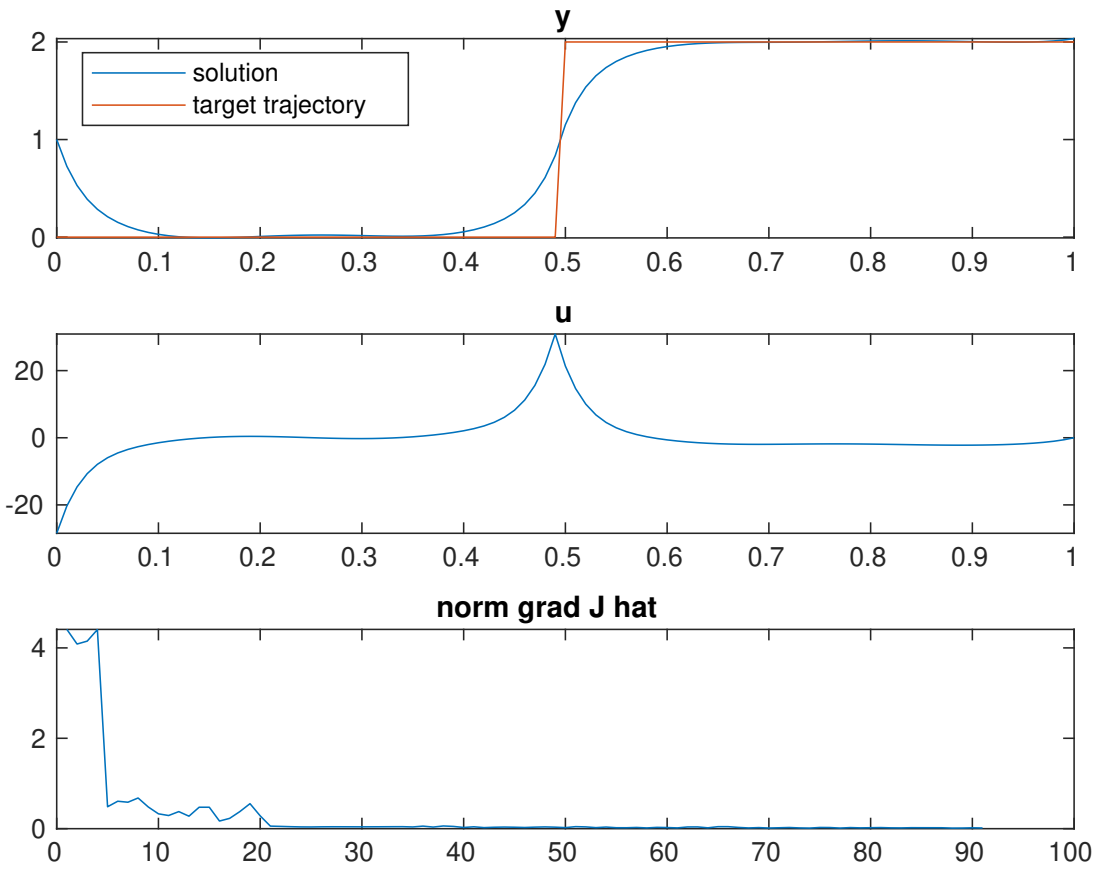


Figure 3.1: Solution, control and $\|\nabla \hat{J}(u_k)\|$ for optimal control problem

3.2 Optimal control of linear ODEs with initial conditions and bounded controls

As before, we consider the optimal control problem given by the tracking functional

$$\begin{aligned} \min_{u \in U} \; J(y,u) \; &:= \frac{1}{2}\|y - y_d\|^2_{L^2(\Omega)} \; + \; \frac{\nu}{2}\|u\|^2_{L^2(\Omega)} \\ y' \; &= \; y \; + \; u \qquad \text{s.t.} \qquad y(0) \; = \; 1 \end{aligned}$$

with $\Omega = (0,1)$, $\nu > 0$ and target trajectory $y_d = 2 \cdot \chi_{(\frac{1}{2},1)}$. But now we only allow constrained controls $u \in U$ with

$$U \; := \; \{u \in L^2(\Omega), \text{ s.t. } u_{low} \leq u \leq u_{high} \text{ a.e.}\},$$

where $u_{low}, u_{high} \in L^2(\Omega)$ are given functions.

In our implementation we assumed $u_{low}, u_{high} \in \mathbb{R}$ to be constant functions, although the algorithm can be easily adapted to the arbitrary case. The problem is again solved numerically by a gradient based optimization scheme. We utilize a

projected conjugate gradient method, where the control u is projected onto the admissasble set U after each update-step. For the discussion of the optimality system, numerical solution of the state and adjoint equation, and derivation of the reduced functional $\hat{J}$ see section 3.1.

"codes/OPCboundedControl/OPCboundedControl.m"

```
%solves the optimal control problem for the functional J(y,u)
%J(y,u)=|y-y_d|^2+1/2*nu*|u|^2
%allows only bounded controls u, bounded form below by u_low,
%and bounded from above by u_high
%y is the state, u is the control, y_d is the target trajectory
%solves the optimization problem via projected gradient descent

%%%initialization

a=0;            %lower interval boundary
b=1;            %upper interval boundary
ya=1;           %initial value for y
N=1e2;          %number of partition points of the interval
dx=(b-a)/N;     %determine step size of the mesh
x=[a:dx:b];     %generate mesh
u=zeros(N+1,1); %initialize the control u
u_low=-10;      %lower boundary for the control values
u_high=10;      %upper boundary for the control values
Kmax=3e2;       %maximal number of iterations
eps=1e-2;       %threshold for gradient magnitude to break iterations

%cost of the penalty term; decrease nu to increase accuracy of
%approximation to target trajectory
nu=1e-3;

%generate function for the functional; receives vectors as inputs
J = @(y,u) 1/2*dx*sum((y-targetTrajectory(x)).^2)+nu/2*dx*sum(u.^2);
%generate reduced functional J_hat
J_hat =  @(u) J(solveStateEq(x,u,ya),u);
%generate gradient of J_hat; grad_J_hat = -p+nu*u
grad_J_hat = @(u) -solveAdjointEq(x,solveStateEq(x,u,ya),u,0)+nu*u;


%calculation via projectedCG
[u,k,grads]=projectedCG(J_hat,grad_J_hat,u,Kmax,eps,u_low*ones(N+1,1)
    ,u_high*ones(N+1,1));

%calculate y from the optimal control u
y=solveStateEq(x,u,ya);

figure
hold on
subplot(3,1,1)
plot(x,[y,targetTrajectory(x)])
title('y')
lgd=legend('solution', 'target trajectory');
lgd.Location='northwest';
subplot(3,1,2)
plot(x,u)
title('u')
subplot(3,1,3)
plot(1:k,grads(1:k))
title('norm grad J hat')
```

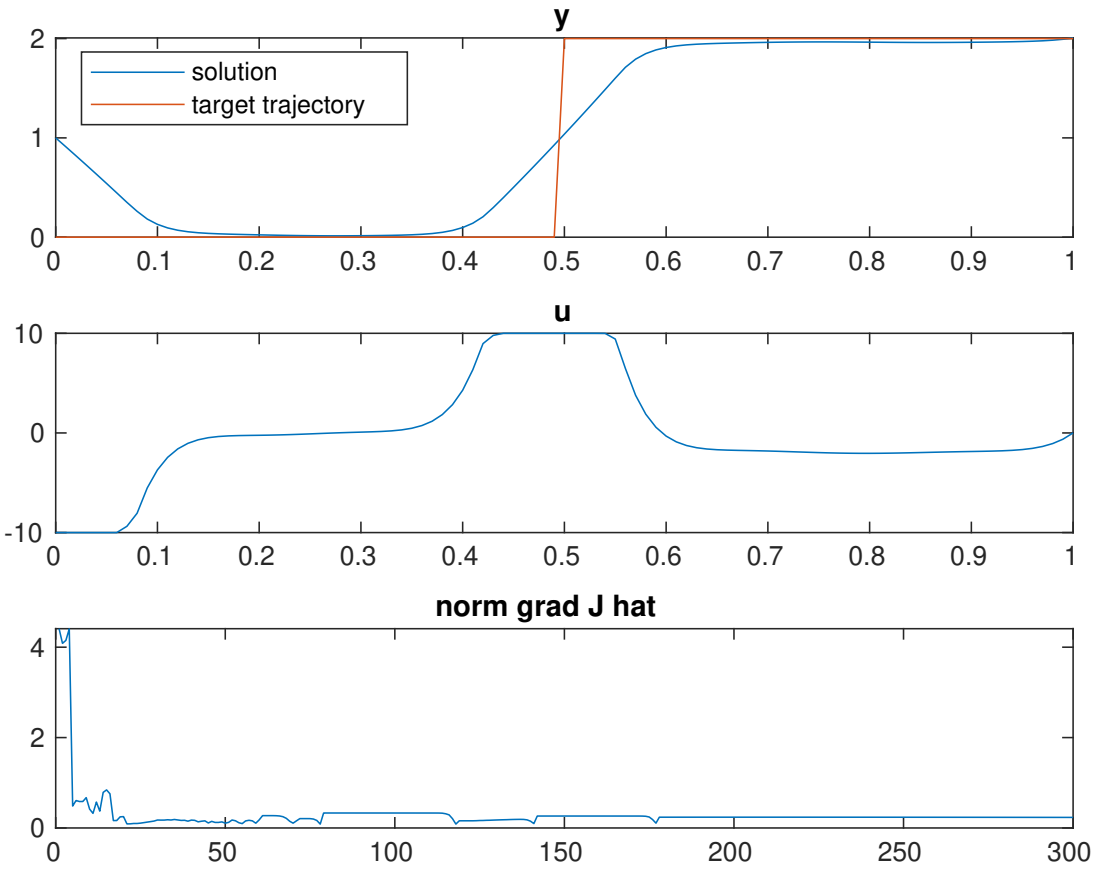


Figure 3.2: Solution, control and $\|\nabla \hat{J}(u_k)\|$ for optimal control problem with bounded controls

From the plot of the derived functions y and u one can deduce the influence of the imposed conditions on the controls. The solution y approximates the target trajectory worse than in the previous section, where arbitrary controls were allowed. The control u piecewisely attains the values u_{low} and u_{high}, and resembles a cut-off version of the unconstrained control derived before.

3.3 Optimal control of elliptic ODEs with Dirichlet boundary condition

For a second example in optimal control we consider the elliptic-type problem with Dirichlet boundary condition

$$\begin{aligned} \min_{u \in L^2(\Omega)} \; J(y,u) \; &:= \frac{1}{2}\|y - y_d\|^2_{L^2(\Omega)} \; + \; \frac{\nu}{2}\|u\|^2_{L^2(\Omega)} \\ -y'' + u\,y \; &= \; f \quad \text{on} \quad \Omega \\ y \; &= \; g \quad \text{on} \quad \partial\Omega \end{aligned}$$

with $\Omega = (0,1)$, $\nu = 10^{-8} > 0$, $f \equiv 1$, boundary conditions $g(0) = 0$, $g(1) = 1$ and target trajectory $y_d = \chi_{(1,\frac{2}{3})} + \chi_{(\frac{1}{3},\frac{2}{3})}$. The control term $u\,y$ in the state equation is called a *bilinear control*. One can easily compute the optimality system as

$$\left.\begin{array}{rcll} -y'' + u\,y &=& f & \text{on } \Omega \\ y &=& g & \text{on } \partial\Omega \end{array}\right\} \text{ state equation}$$

$$\left.\begin{array}{rcll} -p'' + u\,p &=& -(y - y_d) & \text{on } \Omega \\ p &=& 0 & \text{on } \partial\Omega \end{array}\right\} \text{ adjoint equation}$$

$$\nu\,u \;+\; y\,p \;=\; 0 \qquad \text{optimality condition}$$

The numerical calculation is done on a grid $(x_k)_{k=1,\dots,N}$, $x_k = \frac{k-1}{N-1}$ with N points and step size $h := \frac{1}{N-1}$. All functions y, p, u, are discretized as N-vectors by $y_k := y(x_k)$ and so on.

We notice that the state equation and the adjoint equation are of the same structure with only the right-hand sides f and g changed. This is typical for elliptic problems. Therefore we only discuss the numerical solution of the state equation, the adjoint equation is handled analogous. Since the state equation is linear in y, we can use a similar approach like we did for the indirect method in CV in the case optimize-before-discretize above:

By approximating $(y'')(x_j) \approx \frac{1}{h^2}(y_{j+1} - 2y_j + y_{j-1})$ and $(u\,y)(x_j) \approx u_j\,y_j$ the discretized state equation reads

$$-\frac{1}{h^2}(y_{j+1} - 2y_j + y_{j-1}) \;+\; u_j\,y_j \;=\; 1 \quad \text{for} \quad k = 2, ..., N-1$$
$$y_1 = 0, \quad y_N = 1$$

As before, we write this in matrix form

$$\frac{1}{h^2}\begin{pmatrix} 1 & 0 & & \dots & 0 \\ -1 & (2+h^2u_2) & -1 & & \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & -1 & (2+h^2u_{N-1}) & -1 \\ 0 & \dots & & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix} = \begin{pmatrix} \frac{g(0)}{h^2} \\ f_2 \\ \vdots \\ f_{N-1} \\ \frac{g(1)}{h^2} \end{pmatrix}.$$

Thereby we see that in this case evaluating the control-to-state map just amounts to solving a system of linear equations. Using this control-to-state map, we obtain the reduced functional $\hat{J}(u) = J(S(u), u)$ which we want to minimize using a gradient descent algorithm. The gradient of $\hat{J}$ is given by the optimality condition

$$\nabla\hat{J}(u) = \nu\,u \;+\; y\,p$$

which requires us to solve the adjoint equation $p = p(x, y, u)$. As mentioned above, this can be done analogously to the state equation. We want to use some gradient descent scheme, in which the step size has to be determined by Line Search. Note, that in every evaluation of $\hat{J}$ during Line Search, the state equation has to be solved. The gradient in the gradient can be computed as follows:

Algorithm: Compute $\nabla \hat{J}$ for elliptic OPC problem

Input:

ν	cost factor
$y = y(x, u)$	solution to the state equation
$p = p(x, y, u)$	solution to the adjoint equation
u	control function

Output:

$\nabla \hat{J}(u)$	gradient of $\hat{J}$ at u

1: $y \leftarrow y(x, u_k)$ ▷ solve state equation
2: $p \leftarrow p(x, y, u)$ ▷ solve adjoint equation
3: **return** $\nabla \hat{J}(u) \;=\; p\,y \;+\; \nu\,u$ ▷ get gradient via optimality condition

The implementation basically just consists of steepest descent or a conjugate gradient method and requires additionally to solve the state- and adjoint equation. But this is just a system of linear equations like in the SUMT method. We only need to add those two pieces together. It is also worth noting that for this purpose we adjusted the parameter in the Line Search to start out bigger to allow for as faster descent in the beginning and accept smaller decreases to make any descent near the minimum possible.

”codes/OPCelliptic/linesearch.m”

```
%{
Line search algorithm with Armijo condition

For a given function f and direction p at a point x,
finds a step size t such that f admits a sufficient decrease in
    direction p at x

Terminates with step size approximately 1e-5 if no sufficient
    decrease is found.
%}
function [t] = linesearch(f, p, x)
    alpha = 0.00005; % Specifies desired decrease
    t = 4;
    while f(x +t*p) > f(x) - alpha*t*power(norm(p), 2) % Armijo
        condition for not accepting t
        t = 0.5*t; % Backtracking step
        if t < 1e-5 % No decrease found
            return;
        end
    end
end
```

”codes/OPCelliptic/J.m”

```
% Functional J(y,u) = integral_0^1 ( 0.5*(y-yd)^2 + (ny/2)*u^2)
function Jyu = J(x,u, y,yd,  ny)
N = length(x);
dx = (x(N)-x(1))/(N-1);
Jyu = sum( 0.5*(y - yd).^2 + 0.5*ny*u.^2 );
end
```

”codes/OPCelliptic/solveStateEq.m”

```
% Solves the state equation
% -y'' + yu = f on (0,1)
%  y        = g on the boundary of (0,1)
% with f given as a N-vector
% and g given as a 2-vector with the boundary values
function y = solveStateEq(x, u,g, f)
N = length(x);
h = 1/(N-1);
% generate matrix corresponding to the equation
A = (1/(h^2)) * full(gallery("tridiag", N, -1,2,-1));
A = A + diag(u);
A(1,1) = 1;
A(1,2) = 0;
A(N,N) = 1;
A(N, N-1) = 0;

f(1) = g(1);         % boundary condition y(0) = g(0)
f(length(f)) = g(2);% boundary condition y(1) = g(1)

y = (A\(f'))';       % solve linear equation A y = f
end
```

"codes/OPCelliptic/solveAdjointEq.m"

```
% Solves the adjoint equation
% -p'' + pu = -(y-yd) on (0,1)
%  p        = 0       on the boundary of (0,1)
% with yd given as a N-vector
function p = solveAdjointEq(x,y,u, yd)
N = length(x);
h = 1/(N-1);
% generate matrix corresponding to the equation
A = (1/h^2) * full(gallery("tridiag", N, -1,2,-1));
A = A + diag(u);
A(1,1) = 1;
A(1,2) = 0;
A(N,N) = 1;
A(N, N-1) = 0;

f = -(y -yd);      % right-hand side
f(1) = 0;          % boundary condition p(0) = 0
f(length(f)) = 0; % boundary condition p(1) = 0

p = (A\(f'))';     % solve linear equation A p = f
end
```

"codes/OPCelliptic/OPCelliptic.m"

```
a = 0;
b = 1;
N = 200;         % mesh size
x = linspace(a, b, N);
Kmax = 5000;     % maximal number of steps
ny = 1e-8;       % cost factor
g = [0,1];       % boundary values
f = ones(1,N);  % state equation right-hand side
yd = zeros(1,N);% generate target trajectory function
yd(1:1:round(N/3)) = 1;
yd(round(N/3):1:round(2*N/3)) = 2;

Jhat = @(u) J( x, u, solveStateEq(x,u,g, f),yd,   ny);
grad = @(u) gradJhat(x,u, g, f, yd, ny);
% alternatively use
% steepestDescent(Jhat, grad, zeros(1,N), Kmax, 1e-8);
[u k grads] = nonlinearCG(Jhat, grad, zeros(1,N), Kmax, 1e-8);

clf
y = solveStateEq(x,u, g, f);
subplot(3,1,1)
plot(x,y, x, yd) % plot solution and target trajectory
ylim([-0.2,2.3])
legend("solution", "target trajectory")
title("y")
subplot(3,1,2)
plot(x,u) % plot control function
title("u")
subplot(3,1,3)
plot(1:k , grads(1:k)) % plot the gradient of Jhat
title("norm grad J hat");
set(gcf, 'Position',  [100, 100, 800, 500])
```

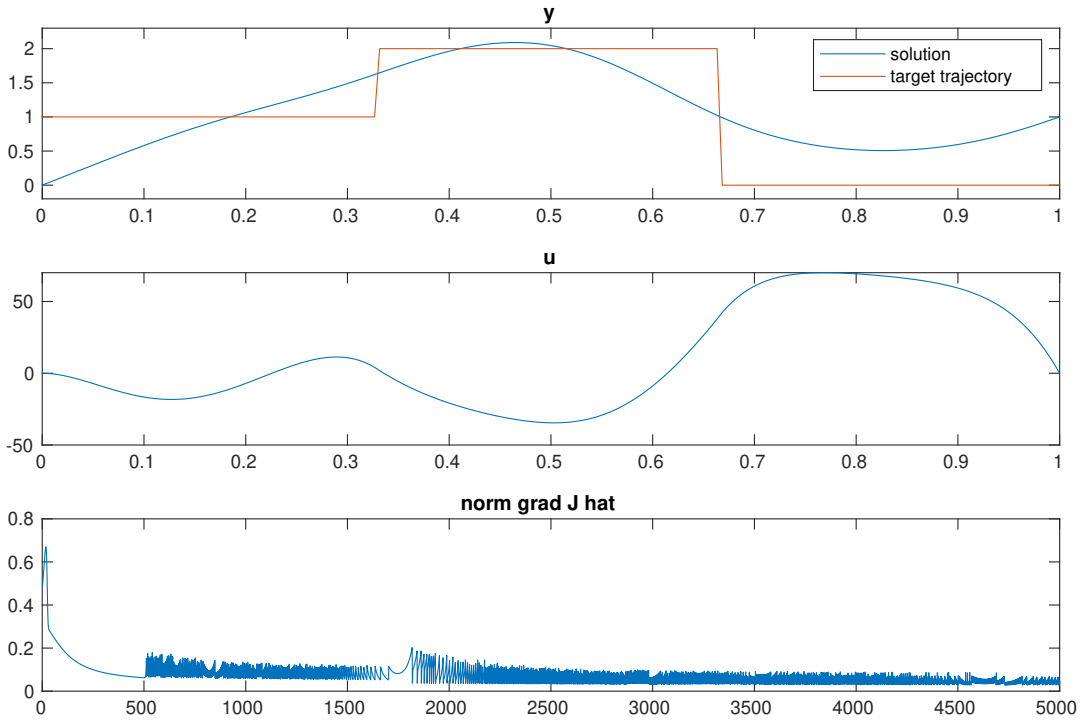


Figure 3.3: Solution, control and $\|\nabla \hat{J}(u_k)\|$ for elliptic optimal control problem with steepest descent

Typical for bilinear controls is very slow descent near the minimum. Comparing the plots generated by steepest descent and conjugate gradient in Figure 3.3 and 3.4, we see that conjugate gradient achieves significantly better results in much fewer steps. It converges much faster away from the minimum and gets a lot closer to it. Steepest descent has the typical problem of oscillating around the minimum we have seen before. Conjugate gradient also oscillates somewhat but much less regular and with smaller amplitude around the minimum.

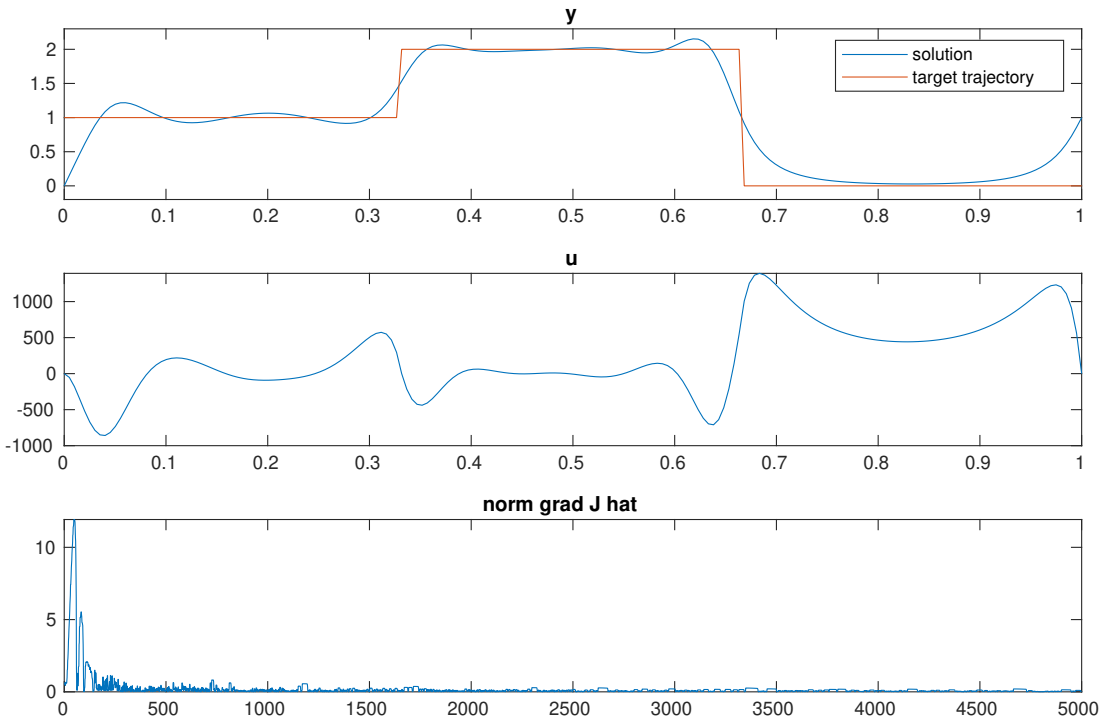


Figure 3.4: Solution, control and $\|\nabla \hat{J}(u_k)\|$ for elliptic optimal control problem with conjugate gradient

List of Figures