Assignment 1

02610 Optimization and Data Fitting – Anders Hørsted (s082382)

Question 1 - Branin's function

Branin's function $\mathbf{r} : \mathbb{R}^2 \to \mathbb{R}^2$ is defined as

$$\begin{aligned} r_1(\mathbf{x}) &= 1 - 2x_2 + \frac{1}{20}\sin(4\pi x_2) - x_1 \\ r_2(\mathbf{x}) &= x_2 - \frac{1}{2}\sin(2\pi x_1) \end{aligned} \tag{1}$$

and the function f is then defined as

$$f(\mathbf{x}) = \frac{1}{2}(r_1(\mathbf{x})^2 + r_2(\mathbf{x})^2) \tag{2}$$

Question 1.1

Since the function f is defined as the sum of two squared values we have that $f(\mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathbb{R}^2$ and only for $r_1(\mathbf{x}) = r_2(\mathbf{x}) = 0$ is $f(\mathbf{x}) = 0$. From this it is seen that any solution of $\mathbf{r}(\mathbf{x}) = \mathbf{0}$ is also a minimizer of f.

Question 1.2

A contour plot of $\log_{10}(f(\mathbf{x}))$ for $\mathbf{x} \in [-10; 10] \times [-10; 10]$ is plotted and shown in figure 1. The source code is found in appendix A.1

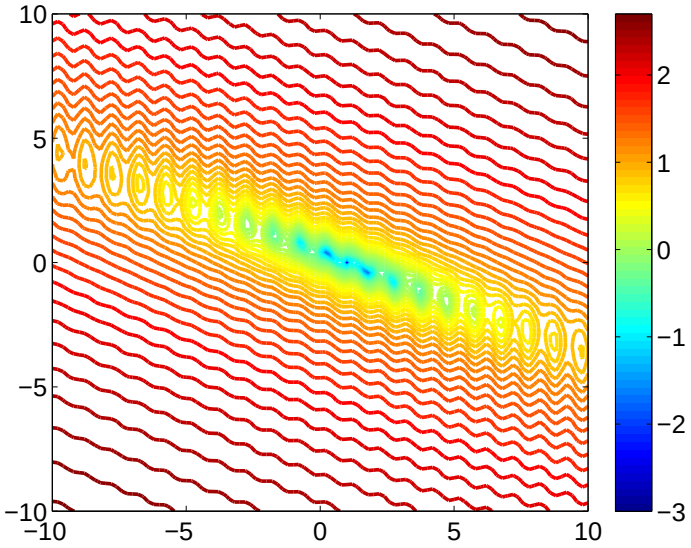

Figure 1: Contour plot of $\log_{10} f(\mathbf{x})$ as defined in question 1.

Question 2 - Newton's Method

Question 2.1

First an MATLAB implementation of Newton's method is created.

```
function [xmin,X,F,DF] = newton(fun,x0,tol,maxiters)

    xmin = x0;
    k = 0;
    x = x0;

    [f,df,ddf] = fun(x);

    X = [x0]; F = [f]; DF = [df];

    converged = (norm(df,'inf') < tol);

    while ~converged && k < maxiters
        s = -(ddf\df);
        x = x + s;

        [f,df,ddf] = fun(x);

        X = [X,x]; F = [F,f]; DF = [DF,df];

        converged = (norm(df,'inf') < tol);

        if converged
            xmin = x;
        else
            xmin = [];
        end
```

```
        k = k + 1;
    end

end
```

Code Listing 1: Implementation of Newton's method

Question 2.2

Source code for this question is in appendix A.2.
The implementation is tested on the quadratic function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x}$ where

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} -1.5 \\ 0 \end{pmatrix}$$

The extremum $\mathbf{x}^*$ can be found analytically by using $\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$, which gives

$$\mathbf{A}\mathbf{x}^* + \mathbf{b} = 0 \quad \Leftrightarrow \quad \mathbf{x}^* = -\mathbf{A}^{-1}\mathbf{b} = \begin{pmatrix} 1 \\ -0.5 \end{pmatrix}$$

Using the implementation of Newton's method we find the same minimizer as the analytical solution, and Newton's method uses only one iteration. This is as expected since the first iteration $\mathbf{x}^{(1)}$ is given by (using that $\nabla^2 f(\mathbf{x}) = \mathbf{A}$)

$$\begin{aligned} \mathbf{x}^{(1)} &= \mathbf{x}^{(0)} - \mathbf{A}^{-1}(\mathbf{A}\mathbf{x}^{(0)} + \mathbf{b}) \quad \Leftrightarrow \\ \mathbf{A}\mathbf{x}^{(1)} + \mathbf{A}\mathbf{x}^{(0)} + \mathbf{b} &= \mathbf{A}\mathbf{x}^{(0)} \quad \Leftrightarrow \\ \mathbf{x}^{(1)} &= -\mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

so $\mathbf{x}^* = \mathbf{x}^{(1)}$ for the quadratic function f.

Question 2.3

Source code for this question is in appendix A.3.
Now the implementation of Newton's method is tested* on f – as defined in (2) – for four different starting guesses: $\mathbf{x}_1^{(0)} = (0,0)^T, \mathbf{x}_2^{(0)} = (1,0)^T, \mathbf{x}_3^{(0)} = (3.9,-1)^T, \mathbf{x}_4^{(0)} = (4.1,-1)^T$. The results are shown in figure 2. For the first three starting guesses the algorithm converges within the first 10 iterations giving the minimizers

$$\mathbf{x}_1^{(8)} = \begin{pmatrix} 0.1487 \\ 0.4021 \end{pmatrix}, \quad \mathbf{x}_2^{(0)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{x}_3^{(6)} = \begin{pmatrix} 3.7305 \\ -1.2306 \end{pmatrix}$$

but for the starting guess $\mathbf{x}_4^{(0)}$ the algorithm do not converge within the allowed 1000 iterations. The reason for the lack of convergence can be found by inspecting the last

*Using a tolerance of 10^{-12} and a maximum of 1000 iterations.

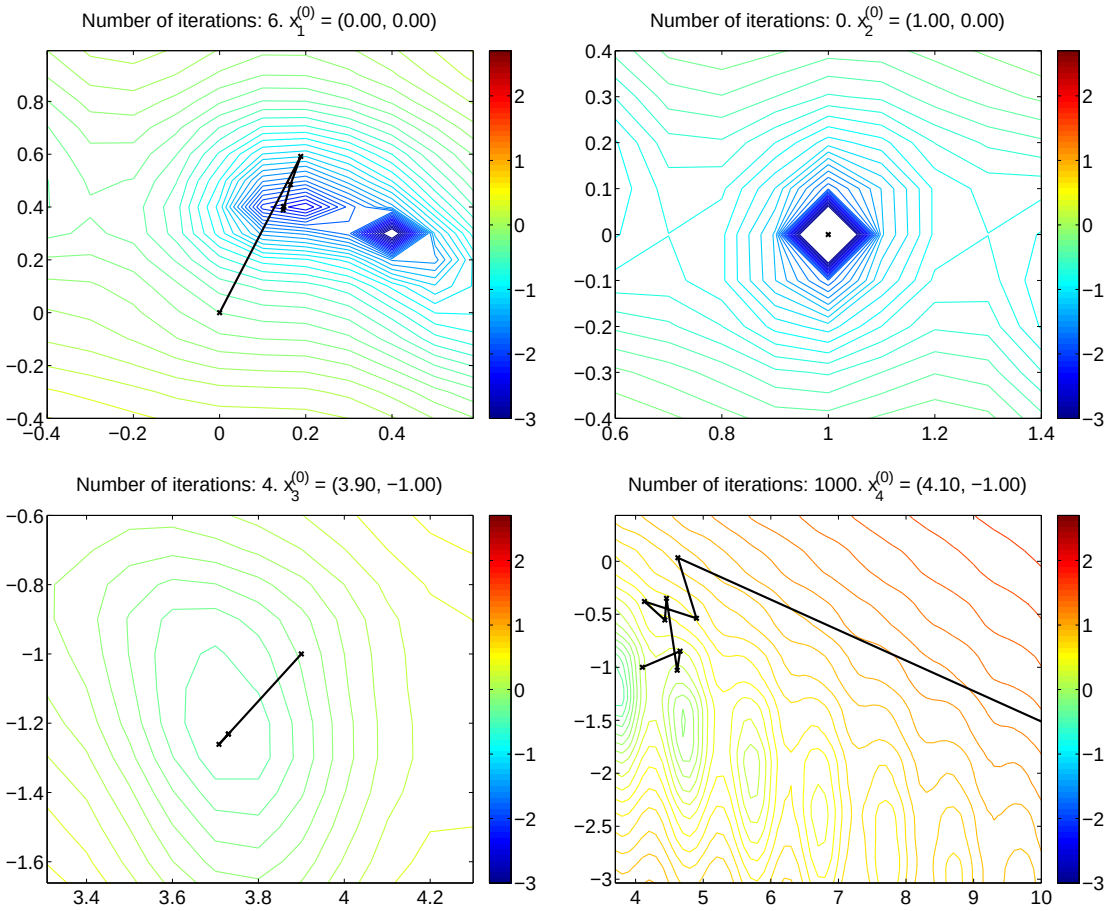

Figure 2: Testing implementation of Newton's method shown in code listing 1

few iterations. For the last 4 iterations the $\mathbf{x}$ values are

$$\mathbf{x}_4^{(997)} = \begin{pmatrix} 13.5082 \\ -2.3639 \end{pmatrix}, \quad \mathbf{x}_4^{(998)} = \begin{pmatrix} 13.5629 \\ -2.5779 \end{pmatrix}, \quad \mathbf{x}_4^{(999)} = \begin{pmatrix} 13.5082 \\ -2.3639 \end{pmatrix}, \quad \mathbf{x}_4^{(1000)} = \begin{pmatrix} 13.5629 \\ -2.5779 \end{pmatrix}$$

and it do look like the algorithm is cycling between two different $\mathbf{x}$ values. Closer inspection of the $\mathbf{x}$ values shows that this is indeed the case.

It is now tested whether the found minimas are solutions for $\mathbf{r}(\mathbf{x}) = 0$ or only local minimizers for $f(\mathbf{x})$. For $\mathbf{x}_2^{(0)} = (1, 0)^T$ it is seen from the expression (1) that it is a solution. For the other three candidates the norms of $\mathbf{r}$ are calculated as

$$||\,\mathbf{r}(\mathbf{x}_1^{(8)})\,||_2 = 2.78 \cdot 10^{-17}, \quad ||\,\mathbf{r}(\mathbf{x}_3^{(6)})\,||_2 = 7.86 \cdot 10^{-1}, \quad ||\,\mathbf{r}(\mathbf{x}_4^{(1000)})\,||_2 = 7.82$$

Which shows that $\mathbf{x}_1^{(8)}$ is a solution, but $\mathbf{x}_3^{(6)}$ and $\mathbf{x}_4^{(1000)}$ are ony local minimizers of f.

Question 3 - Least Squares Methods

In this question different function will be fitted to a dataset containing measurements of light intensity in a optical fibre as a function of time after source cut off.

Question 3.1

Source code for this question is in appendix A.4.
First polynomials of degrees from 1 to 6 are fitted to the data using the MATLAB function `polyfit`. The results are shown in figure 3 and from the figure it is seen that the fit can be improved for values inside the data interval, by raising the degree of the polynomial. This is as expected since we could make the polynomial fit the data exactly by choosing the degree equal to the number of datapoints minus one (as long as there aren't two measurements for the same t). Outside the data interval though the behaviour of the fitted polynomials do not behave as well since all polynomials makes a sharp upward or downward turn when $t > 32$. This behaviour do not match the physical reality well, so instead of using polynomials, exponential functions are fitted instead.

Question 3.2

Source code for this question is in appendix A.5.
Now the data $\{(t_i, y_i)\}_{i=1}^{m}$ is fitted by the model

$$M(x, t) = x_2 e^{-x_1 t} + x_3 \tag{3}$$

First a function returning the residual vector and the jacobian matrix for the data given the parameters $\mathbf{x}$ as input is implemented. To make the function more generic we let it receive the data `t` and `y` as second and third argument. The actual data can then be passed in to the `marquardt` function.

```
function m1 = M1(x, t)
    m1 = x(2)*exp(-x(1)*t) + x(3);
end
```

Code Listing 2: Function returning value of model 1

```
function [r, J] = residual_jacobian_M1(x, t, y)
    r = M1(x,t) - y;
    J = [-x(2)*t.*exp(-x(1)*t) exp(-x(1)*t) ones(size(t))];
end
```

Code Listing 3: Function returning residual vector and jacobian for a given data set

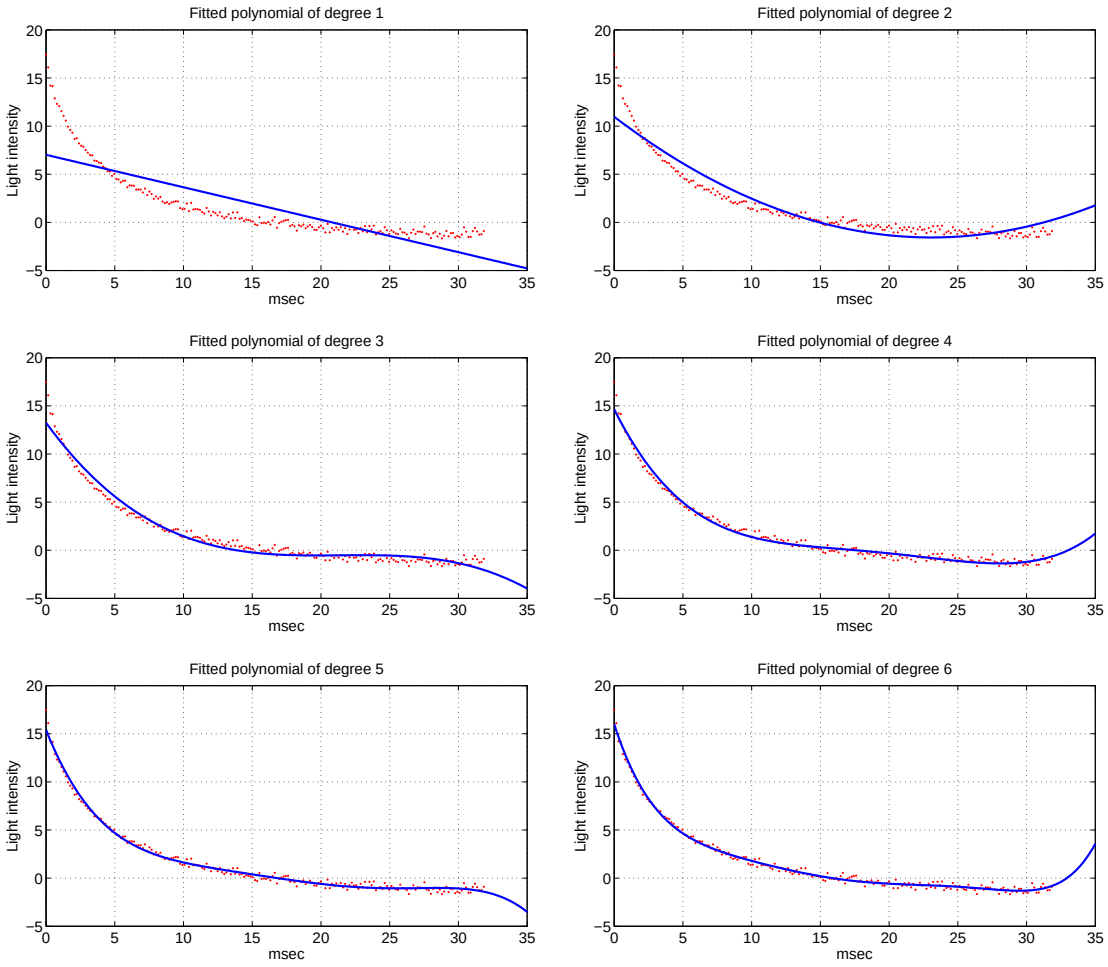

Figure 3: Fitted polynomials of degrees 1 to 6.

The actual model is defined in a separate function `M1` since it is going to be reused when plotting the model. To ensure that the function is correctly implemented it is checked by running the `checkgrad` function.

```
[maxJ, err, ind] = checkgrad(@residual_jacobian_M1, x0, 1e-5, t, y);
```

The function returns the maximum value J_m of the calculated jacobian, the maximum error of the forward- (δ_F), backward- (δ_B), and extrapolated difference approximations and the indicies of where the maximum difference approximation errors was obtained. If the jacobian was correctly implemented it should be expected that $\delta^B \simeq \frac{1}{2}\delta^F$ and that δ^E should be of the order of magnitude $\simeq (\delta^F)^2$ (see [p.3]nielsen00). The actual results was

$$\delta^F = 4.05 \cdot 10^{-5}, \quad \delta^B = -2.03 \cdot 10^{-5}, \quad \delta^E = -3.29 \cdot 10^{-10}$$

which strongly indicates that the `residual_jacobian_M1` function was correctly implemented.

Question 3.3

A reasonably starting point for the least squares fitting should now be obtained, from figure 2 in the assignment description. From (3) it is seen that for any x, $M(x,t) \to x_3$ for $t \to \infty$. From the figure a starting guess for x_3 could be $x_3 = -1$. From the figure is looks as if

$$M(x,0) = 17 \quad \Leftrightarrow \quad x_2 - 1 = 17 \quad \Leftrightarrow \quad x_2 = 18$$

Finally it is seen from the figure that approximately

$$M(x,15) = 0 \quad \Leftrightarrow \quad 18e^{-15x_1} - 1 = 0 \quad \Leftrightarrow \quad x_1 = \frac{\log(18)}{15} \approx 0.2$$

A decent starting guess will therefore be $\mathbf{x}^{(0)} = (0.2, 18, -1)^T$

Question 3.4

Source code for this question is in appendix A.5.
It is now time to fit the model (3) using the `marquardt` function from `immoptibox`. Using the start guess found in previous question the Levenberg-Marquardt method fits the model

$$M_1^*(t) = 15.66e^{-0.19t} - 0.95$$

to evaluate the model the fit is plotted along with the data. Also a plot of the residuals is created. Both plots can be seen in figure 4. From the plot of the fitted model the fit seems to be decent, but the model is seen to consistently give too high values for $t \in [2;6]$ and too low values for $t \in [6;15]$. This is confirmed by the residuals plot where it is also seen that the model fits really bad for t near 0. Also there seems to be some structure left in the residuals which shouldn't be there if the fit is perfect.

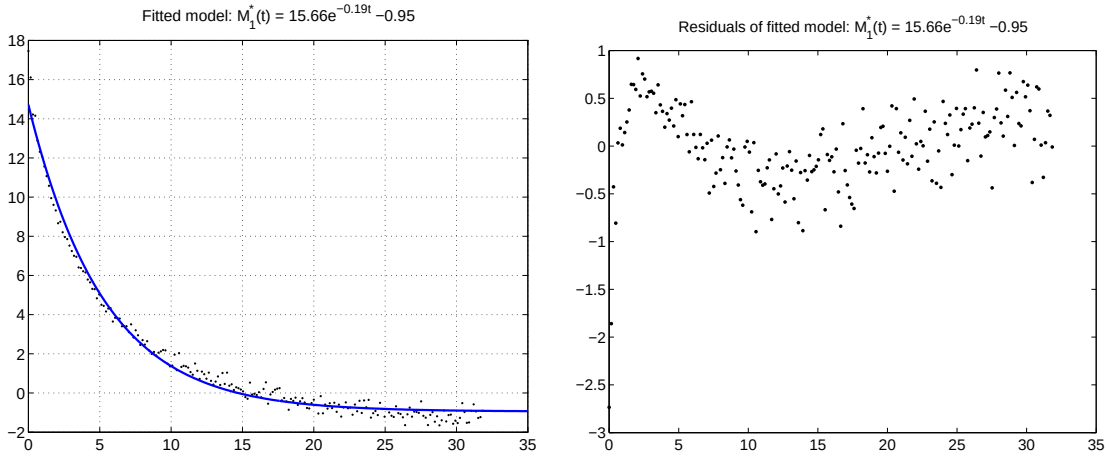

Figure 4: Plot of fitted model and residuals for model M_1 given by (3)

Iteration	$\|\| x_k - x_{16} \|\|$	Iteration	$\|\| \nabla f \|\|$	Iteration	f
11	8.24e-04	11	9.02e-04	11	1.95e+01
12	3.22e-04	12	1.38e-04	12	1.95e+01
13	1.25e-04	13	2.10e-05	13	1.95e+01
14	4.72e-05	14	3.20e-06	14	1.95e+01
15	1.35e-05	15	4.88e-07	15	1.95e+01
16	0.00e+00	16	4.47e-08	16	1.95e+01

Table 1: x errors, gradient norm and function values for last 6 iterations of the Levenberg-Marquardt algorithm for M_1

Comparing the model with the polynomial models found earlier, it is seen that the polynomial models of degree 5 and 6 fitted the data better inside the data interval ($t \in [0; 32]$), but the exponential model behaves more plausible outside the data interval.

The Levenberg-Marquardt algorithm converged in 16 iterations and from table 1 it is seen that the convergence was superlinear for the last iterations. This corresponds with the theory as mentioned in [1, p. 262].

Question 3.5

Source code for this question is in appendix A.5.
Since the residuals of the previous model still showed some structure an extended model is now fitted. The model is given as

$$M_2(x, t) = x_3 e^{-x_1 t} + x_4 e^{-x_2 t} + x_5 \tag{4}$$

To fit the model a starting guess needs to be found. Using the fitted M_1 model a good starting point could be

$$\mathbf{x}^{(0)} = (0.2, 0.2, 8, 8, -1)^T$$

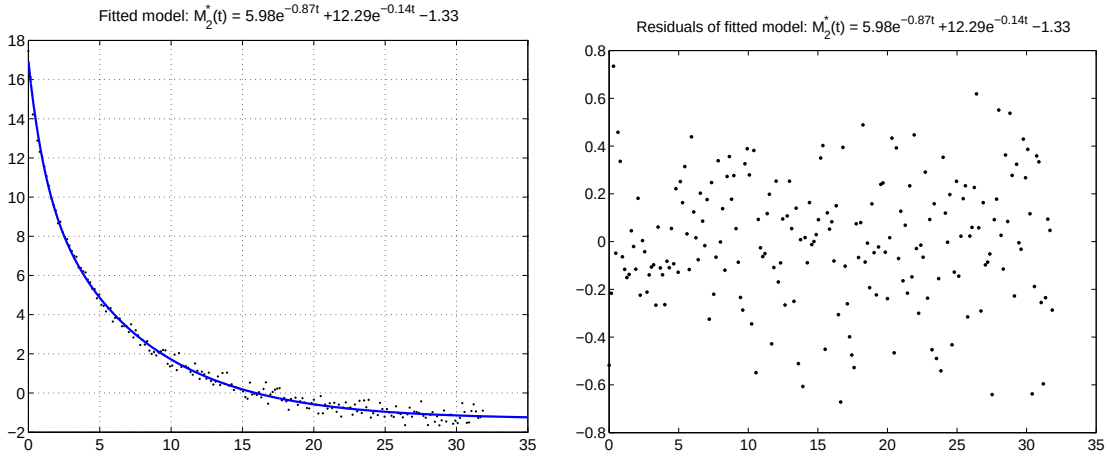

Figure 5: Plot of fitted model and residuals for model given by (4)

Using this starting guess a least squares fit for the new model is found using the `marquardt` function and the optimal model is found as

$$M_2^*(t) = 5.98e^{-0.87t} + 12.29e^{-0.14t} - 1.33$$

The new model is plotted along with the residuals of the model in figure 5

Question 3.6

From figure 5 it is seen that the fit has improved overall compared with model M_1. The residuals looks much more like white noise which is expected for a well-fitted model. It is often assumed that the variance of the residuals is independent of the input t, but it is seen that the variance is smaller for $t \in [1; 8]$ than for other t values. This could maybe be fixed with a transformation of the data, but shouldn't be a big problem. Most importantly it is seen that the residuals center around a mean of 0 for all values of t. All in all the fit seems to be satisfactory for the data.

Question 4

Question 4.1

Source code for this question is in appendix A.6.
In this question a line search algorithm based on steepest decent, newton's method and BFGS quasi-Newton is developed. Since the main logic in the three different methods are very much alike, all three methods are implemented in a single function called `linesearch`. The function spans more than one page when all comments are included so the code is found in appendix A.6

Question 4.2

Since all three algorithms implemented in the previous question are line search algorithms they share much of the same general structure. For the k'th iteration a search direction p_k in the x-space is determined. When the direction has been chosen a step-length selection algorithm determines how large a step that should be taken in the chosen direction p_k. The differences between the algorithms are how they choose the search direction and how much information about the function f that should be minimized they need to choose the direction.

Steepest decent

In the steepest decent method the direction p_k is chosen as the direction where the decrease in f is largest pr. step length. This means that the direction is chosen as

$$p_k = -\nabla f_k$$

The primary advantage of the steepest decent method is that convergence to the real minimizer can be guarenteed as long as the step lengths satisfies the Wolfe conditions (see [1, p.38-40]). The main disadvantage is that only linear convergence is expected for all iterations [2, p.29].

Newton's method

In Newton's method the objective function f is approximated by its 2nd order Taylor polynomial around the current iterate x_k. The next iterate is then chosen as the minimizer of the Taylor polynomial. As shown in [2, p.44] this gives the direction

$$p_k = -(\nabla^2 f_k)^{-1} \nabla f_k$$

This choice of direction is based on the assumption that the hessian isn't singular. Also the hessian must be positive definite to ensure that the algorithm converges towards a minimum. The advantage of the Newton method is that if the hessian is positive definite the algorithm converges quadratically for x "close" to the minimizer (theorem 3.4 in [2]). The disadvantages is that for many problems the hessian won't be positive definite. It may even be singular and then the direction isn't even defined. Also the information about the hessian may be expensive to calculate.

BFGS Quasi-Newton

The Quasi-Newton method is an attempt to correct for the disadvantages of the Newton method. This is done by using an approximation H_k of the inverse hessian $(\nabla^2 f_k)^{-1}$ giving the direction

$$p_k = -H_k \nabla f_k$$

For each iteration the approximate hessian is updated. The update formula is derived from the equation

$$H_{k+1}(\nabla f_{k+1} - \nabla f_k) = x_{k+1} - x_k$$

along with some extra conditions on H_{k+1} to make the solution unique. The final update formula is given by e.g. equation 6.17 in [1].

Step length selection

Common for all three algorithms described above is that they need to choose a step length when a direction has been determined. For the implementation in the previous question the simple backtracking algorithm on page 37 in [1] was used. This algorithm satisfy the Wolfe conditions, by starting with a large step length and then decreasing the step length by a factor ρ until the sufficient decrease condition (eq. 3.6a in [1]) is satisfied.

Question 4.3

Source code for this question is in appendix A.7.
The algorithms are now tested on the Rosenbrock problem

$$\min_x f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

Since f is the sum of two squared terms, $\min f(x) \geq 0$. From the second term it is then seen that $x_1 = 1$ and the first term then gives $x_2 = 1$. Therefore $x = (1, 1)^T$ is the true minimizer of f.

Now using respectively $x_1 = (-1.2, 1)^T$ and $x_2 = (1.2, 1.2)^T$ as starting guesses, the three algorithms are tested on the Rosenbrock problem, and the convergence of the gradient norm is plotted along with the path taken in the x-plane. The results for starting point x_1 is seen in figure 6 and the results for starting point x_2 in figure 7. For no of the starting points is the steepest descent algorithm able to converge within 1000 iterations. For both starting points the gradient of the steepest descent algorithm zig-zags in size and only very slowly converges toward 0. As already mentioned the gradient size will eventually converge but only very slowly.

For both starting points the Newton method is seen to perform well. For x_2 the Newton method converges in just 8 iterations. This wasn't guaranteed since nothing in the implementation ensures that the hessian is positive definite. Not even regularity is checked for. It did work for x_1 and x_2 in Rosenbrocks problem though and with fast convergence.

The Quasi-Newton method uses a little more than 21 iterations for both x_1 and x_2. The convergence of the gradient is slow in the start, but as the x_k gets closer to the real minimum the convergence becomes super linear, just as expected. For both x_1 and x_2 it is seen that the first step of the Quasi-Newton method is equal to the first step of the steepest descent method. This is because the identity matrix is used as the initial approximation of the hessian which results in the first step being exactly the same as the steepest descent method.

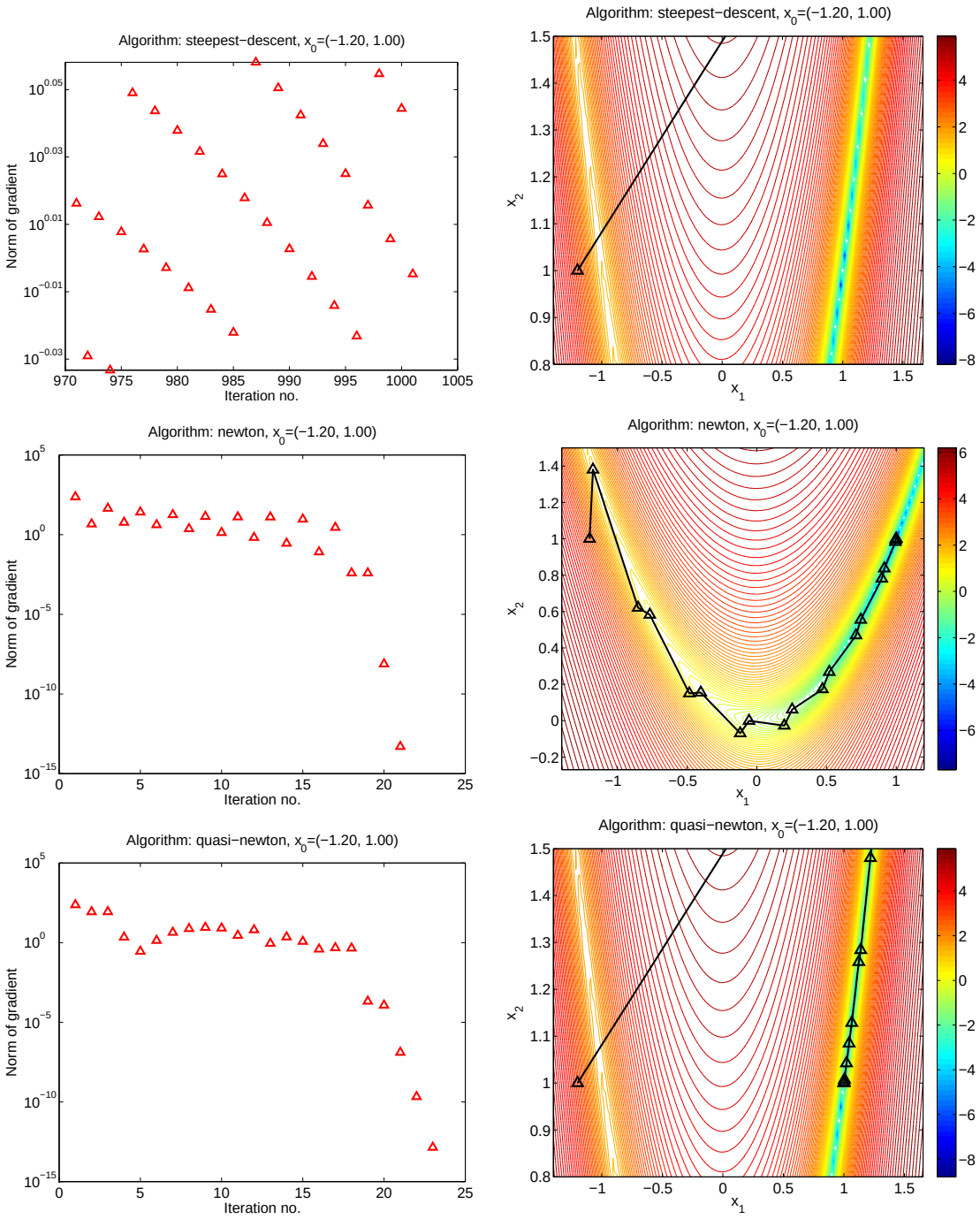

Figure 6: Performance and iteration path for starting point $x_1 = (-1.2, 1)^T$ using the `linesearch` implementation

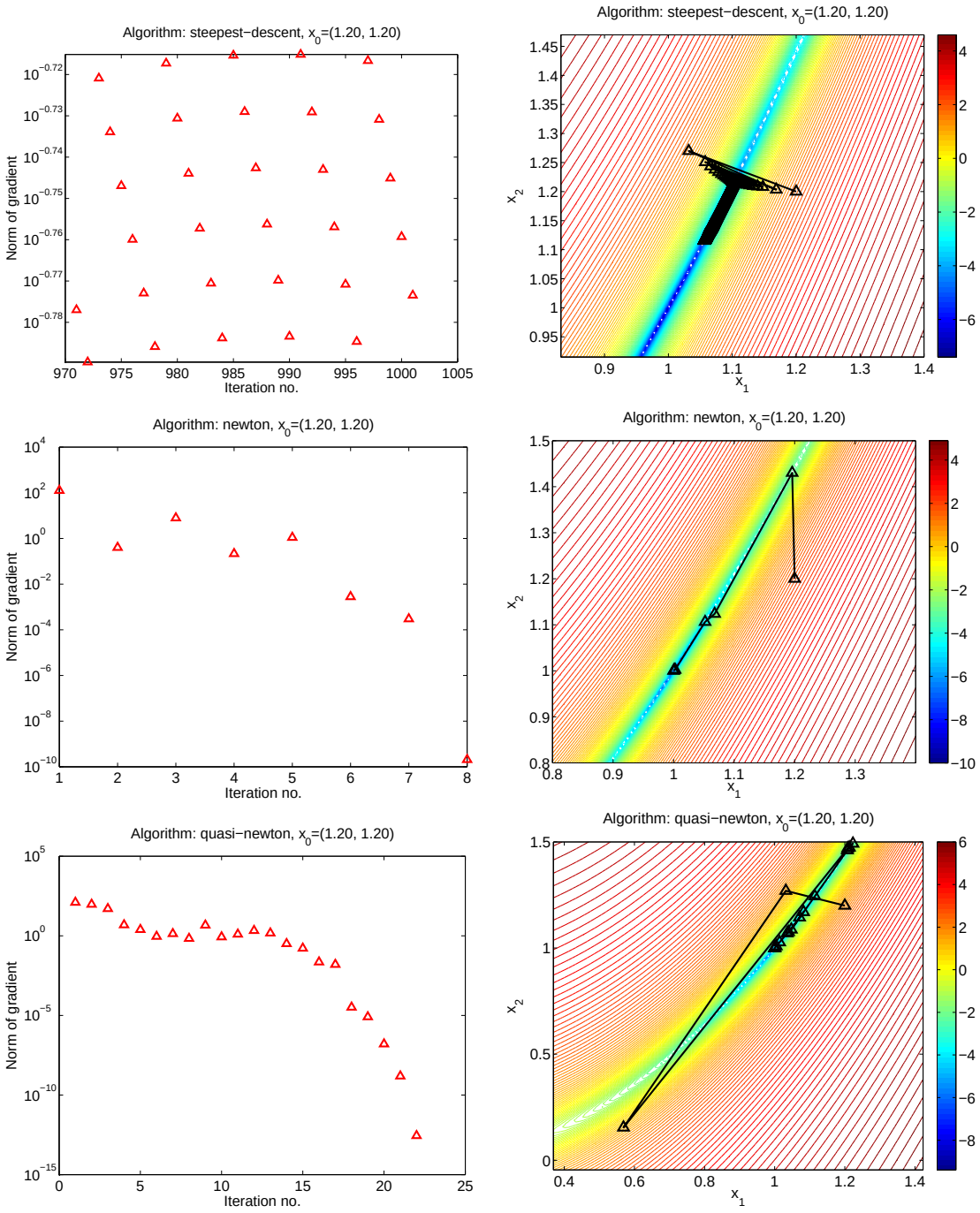

Figure 7: Performance and iteration path for starting point $x_2 = (1.2, 1.2)^T$ using the `linesearch` implementation

Question 4.4

Source code for this question is in appendix A.8.
The Rosenbrock problem is now solved using the built-in MATLAB function `fminunc`. By tweaking the options for `fminunc` both the steepest descent method and the Quasi-Newton method can be calculated. The results are shown in figure 8 and figure 9.

As can be seen from the plots the steepest descent method start out by taking a much smaller step, compared with the implementation in the `linesearch` function from question 4.1. This is probably due to the different step-length algorithm used. In the `linesearch` implementation the simple backtracking algorithm ([1, algorithm 3.1]) was used and in the `fminunc` function a cubic interpolation algorithm was used. Due to the small first step size this version of the steepest descent is able to converge in around 850 iterations which is better than the `linesearch` implementation that wasn't able to converge in 1000 iterations.

For starting point x_1 the `fminunc` implementation of the Quasi-Newton method performs much like the `linesearch` implementation. Due to the large difference in the first step size of the two implementations they approach the minimum from opposite directions in the alley. In this case it doesn't affect the performance though. For starting point 2 the `fminunc` Quasi-Newton method performs better. The difference between the `fminunc` and `linesearch` implementation, again seems to be the step length algorithm. In both step 1 and 2 the `linesearch` implementation takes a too large step and then it uses a couple of iterations to "get back in the alley".

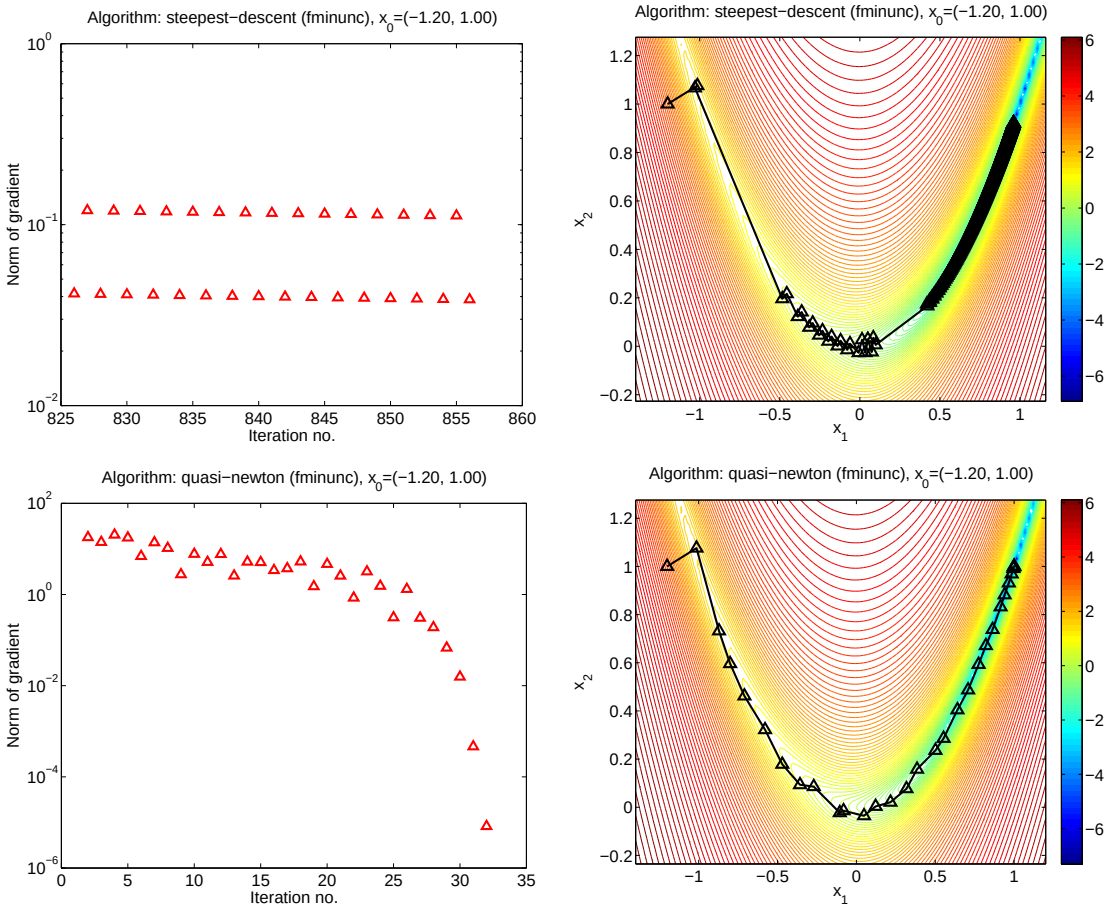

Figure 8: Performance and iteration path for starting point $x_1 = (-1.2, 1)^T$ using the `fminunc` function

Question 4.5

Source code for this question is in appendix A.9.
The Rosenbrock problem is now formulated as a least squares problem. First the Rosenbrock function is written as

$$f(x) = (10(x_2 - x_1^2))^2 + (1 - x_1)^2$$

By defining

$$r_1(x) = 10(x_2 - x_1^2), \quad r_2(x) = 1 - x_1$$

the Rosenbrock function can be written

$$f(x) = \sum_{i=1}^{2} r_i(x)^2$$

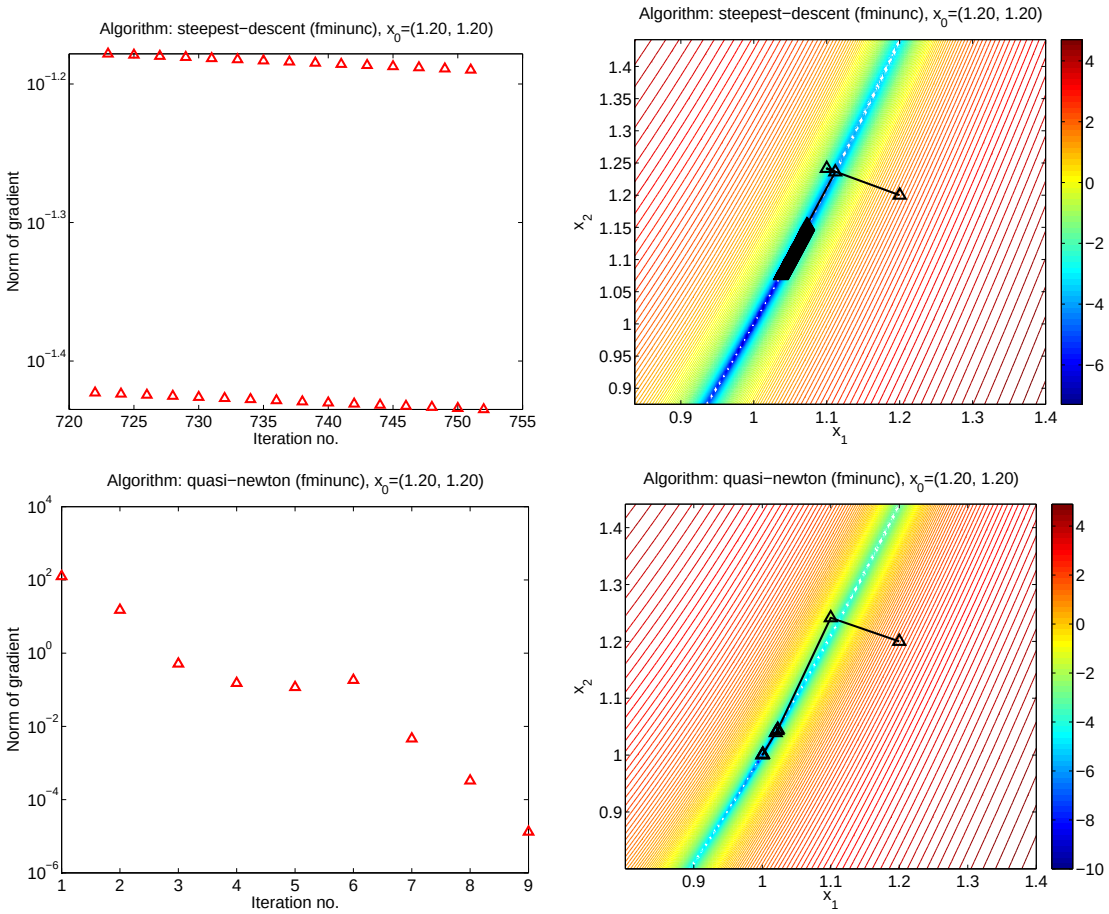

Figure 9: Performance and iteration path for starting point $x_2 = (1.2, 1.2)^T$ using the `fminunc` function

which is the form of the objective function of the least squares problem. The jacobian is the given by

$$J(x) = \begin{bmatrix} -20\,x_1 & 10 \\ -1 & 0 \end{bmatrix}$$

A line search algorithm based on the Gauss-Newton approximation is now developed. This is just the Newton method where the hessian is approximated by using information for the jacobian J. Using the `linesearch` implementation from question 4.1 the Gauss-Newton method can be implemented as

```
function [xmin, X, F, DF, A] = gaussnewton(fun, x0, opts)

    function [f, df, ddf] = calc_f(x)
        [r, J] = fun(x);
        f = (r'*r)/2;
        df = J'*r;
```

```
        ddf = J'*J;
    end

    opts.algorithm = 'newton';
    [xmin, X, F, DF, A] = linesearch(@calc_f, x0, opts);

end
```

Code Listing 4: Implementation of Gauss-Newton method

Using this implementation, the method is now tested on the Rosenbrock problem and the results are shown in figure 10. The performance of the Gauss-Newton is seen to be really good for this problem. As mentioned in [1, page. 257] the convergence can be expected to be superlinear if the hessian approximation $(J^T J)$ is an good approximation. This seems to be the case for this problem and for starting point x_2 the Gauss-Newton algorithm converges in just 3 steps.

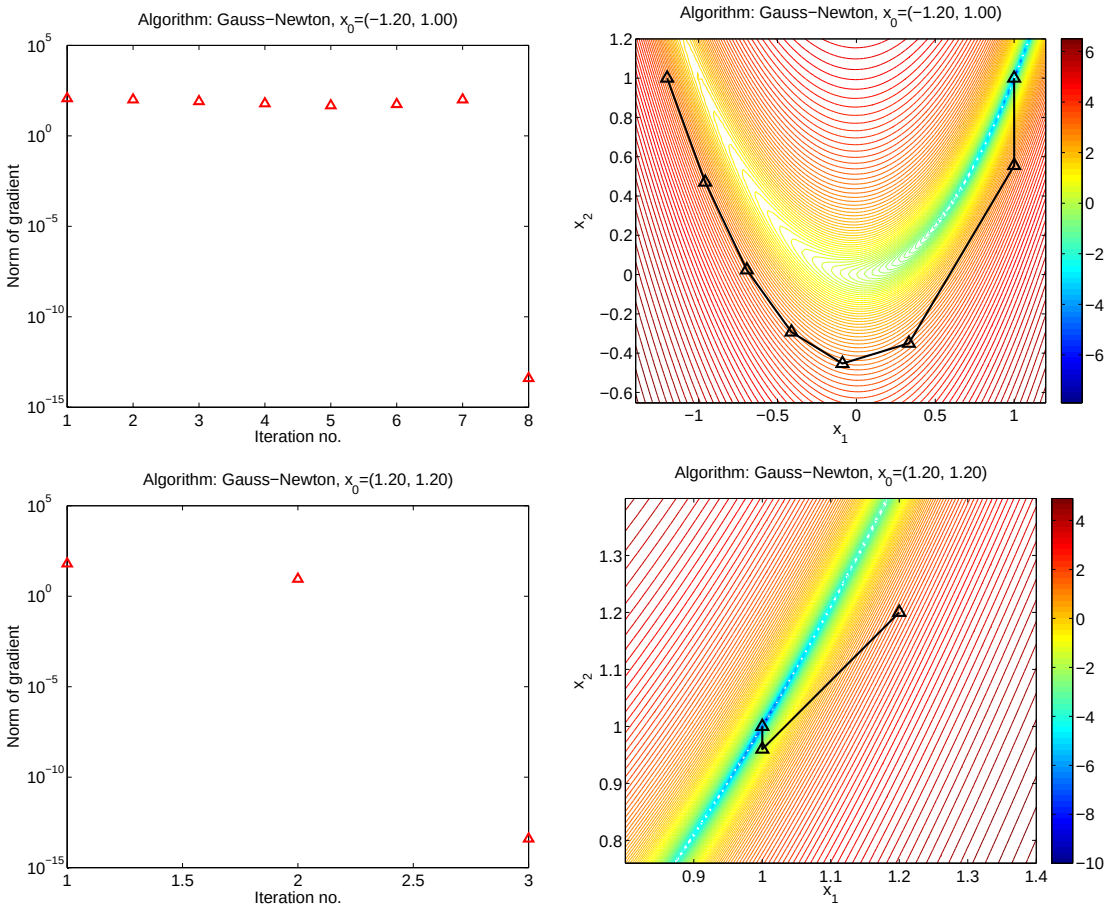

Figure 10: Performance and iteration path for the Gauss-Newton implementation

Conclusion

In this assignment a couple of line search algorithms have been implemented. The performance of the algorithms was compared and it was seen that the steepest descent method performed very bad on the Rosenbrock problem. The Newton method performed very well but the method can break down if the hessian isn't positive definite. For the Rosenbrock problem with the two starting point used there was no problems with the Newton method though. The Quasi-Newton BFGS method was also implemented, and it performed well on the Rosenbrock problem.

In question 1 it was seen how a system of non-linear equations can be expressed as a least squares problem. Later it was also found that the Rosenbrock problem could be expressed as a least squares problem and this gave the opportunity to use the Gauss-Newton to solve the problem. In question 3 least squares was used for data fitting and the performance of the Levenberg-Marquardt algorithm was touched upon.

A Appendices

All MATLAB source code is included in the appendices. All the source code including the LaTex code used for the report can also be found at `https://github.com/alphabits/dtu-fall-2011/tree/master/02610/assignment-1`.

A.1 Exercise 1.2

```
x = -10:0.2:10;
y = -10:0.2:10;
v = -3:0.1:3;
[X, Y] = meshgrid(x, y);

logfs = max(log10(arrayfun(@mybranin, X, Y)), -5);

contour(X, Y, logfs, v, 'linewidth', 1);
colorbar;
set(gca, 'fontsize', 14);
```

Code Listing 5: exercise12.m

A.2 Exercise 2.2

```
function [xmin, X, F, DF] = exercise22()
% Is really a script but since matlab won't allow functions to be defined
% in a script file a function is used instead of a script.

    A = [2 1; 1 2];
    b = [-1.5; 0];

    % Define input function for newton function
    function [f, df, ddf] = fun(x)
        f = (x'*A*x)/2 + b;
        df = A*x + b;
        ddf = A;
    end

    % Call newton implementation
    [xmin, X, F, DF] = newton(@fun, [0;0])
end
```

Code Listing 6: exercise22.m

A.3 Exercise 2.3

```
% Define the four starting points
```

```
x0s = [0 1 3.9 4.1;
       0 0 -1 -1];

% Calculate function values for the contour plot
x = -10:0.1:10;
y = -10:0.1:10;
v = -3:0.1:3;
[Xg, Yg] = meshgrid(x, y);
logfs = max(log10(arrayfun(@mybranin, Xg, Yg)), -5);

tol = 1e-12;
maxiters = 1000;

% Initialize cell arrays to hold the results from the four
% different starting points.
xmins = {};
Xs = {};
Fs = {};
DFs = {};

for i=1:4
    % Calculate results and save
    [xmin, X, F, DF] = newton(@branin, x0s(:,i), tol, maxiters);
    xmins{i} = xmin;
    Xs{i} = X;
    Fs{i} = F;
    DFs{i} = DF;

    % Plot contour plot
    contour(Xg, Yg, logfs, v, 'linewidth', 1);
    colorbar;
    set(gca, 'fontsize', 14);
    hold on;
    xpath = X(1,:);
    ypath = X(2,:);

    % Determine the plot limits based on the path points
    xlimits = [max(min(xpath)-0.4, -10) min(max(xpath)+0.4, 10)];
    ylimits = [max(min(ypath)-0.4, -10) min(max(ypath)+0.4, 10)];
    plot(xpath, ypath, 'kx-', 'linewidth', 2);
    xlim(xlimits);
    ylim(ylimits);
    plottitle = sprintf('Number of iterations: %d. x^{(0)}_%d = (%.02f, %.02f)', ...
                        length(xpath)-1, i, x0s(1,i), x0s(2,i));
    title(plottitle, 'fontsize', 18);
    hold off;
    set(gca, 'fontsize', 18);

    % Save as eps
    print('-depsc', '-loose', sprintf('newton-on-branin-%d', i));
end
```

Code Listing 7: exercise23.m

A.4 Exercise 3.1

```
addpath('/home/anders/dtu/E11/02610/src/immoptibox');

data = load('optic.dat');

% Polynomial degrees to fit
degrees = 1:6;
% Fontsize in plots
fs = 14;

for i=degrees
    coeffs = polyfit(data(:,1), data(:,2), i)
    xs = linspace(0, 35, 200);
    ys = polyval(coeffs, xs);
    % Plot data
    plot(data(:, 1), data(:, 2), 'r.', 'MarkerSize', 6);
    hold on;
    % Plot fitted polynomial
    plot(xs, ys, 'linewidth', 2)
    xlabel('msec', 'fontsize', fs); ylabel('Light intensity', 'fontsize', fs);
    xlim([0 35]);
    title(sprintf('Fitted polynomial of degree %d', i), 'fontsize', fs);
    set(gca, 'fontsize', fs);
    a = get(gca, 'DataAspectRatio');
    % Change aspect of plot. Can probably be done much easier
    set(gca, 'DataAspectRatio', a.*[1 2 1]);
    grid on;
    saveeps(sprintf('fitted-polynomial-%d.eps', i));
    hold off;
end
```

Code Listing 8: exercise3.m

A.5 Exercise 3.2-3.6

```
addpath('/home/anders/dtu/E11/02610/src/immoptibox');

data = load('optic.dat');

t = data(:, 1);
y = data(:, 2);

% Choose between model 1 or 2.
model = 1;

if model == 1
    x0 = [0.2, 18, -1];
    res_jac = @residual_jacobian_M1;
    M = @M1;
else
    x0 = [0.2, 0.2, 8, 8, -1];
    res_jac = @residual_jacobian_M2;
    M = @M2;
end

[xs, info, perf] = marquardt(res_jac, x0, [0 1e-7 1e-12 0], t, y);
```

```
xmin = xs(:,end);

% Plot gradient as function of iteration number
set(gca, 'fontsize', 16);
semilogy(perf.ng, 'r^', 'markersize', 8, 'linewidth', 2);
titlestr = sprintf('Gradient size for model M_%d (Levenberg-Marquardt)', model);
title(titlestr, 'fontsize', 16);
xlabel('Iteration no.'); ylabel('Norm of gradient');
saveeps(sprintf('least-squares-convergence-model-%d.eps', model));

% Save results as latex tables
xerr = xs - repmat(xmin, 1, size(xs,2));
xerr_norms = sqrt(xerr(1,:).^2 + xerr(2,:).^2 + xerr(3,:));
tbldata = {perf.ng, perf.f, xerr_norms};
tbldatafilenames = {sprintf('gradient-norm-model-%d.tex', model), ...
                    sprintf('function-values-model-%d.tex', model), ...
                    sprintf('xerror-model-%d.tex', model)};
for i=1:3
    tmpdata = tbldata{i};
    num = length(tmpdata);
    tosave = 1:num;
    file = fopen(tbldatafilenames{i}, 'w');
    fprintf(file, '%d & %.2e \\\\ \n', [tosave; tmpdata(tosave)]);
    fclose(file);
end

% Calculate residuals
r = res_jac(xmin, t, y);

% Define string representation of model
if model == 1
    modelstr = sprintf('M_1^*(t) = %.02fe^{%.02ft} %+.02f', ...
                       xmin(2), -xmin(1), xmin(3));
else
    modelstr = sprintf('M_2^*(t) = %.02fe^{%.02ft} %+.02fe^{%.02ft} %+.02f', ...
                       xmin(3), -xmin(1), xmin(4), -xmin(2), xmin(5));
end

% Save string representation for latex report
fid = fopen(sprintf('model-%d-representation.tex', model), 'w');
fwrite(fid, modelstr);
fclose(fid);

% Plot the fitted model and the data set
newt = linspace(0, 35, 200);
plot(t, y, 'k.', newt, M(xmin, newt), 'linewidth', 2);
title(['Fitted model: ' modelstr], 'fontsize', 14);
set(gca, 'fontsize', 14);
grid on;
saveeps(sprintf('least-squares-model-%d.eps', model));

% Plot the residuals
plot(t, r, 'k.', 'MarkerSize', 10);
title(['Residuals of fitted model: ' modelstr], 'fontsize', 14);
set(gca, 'fontsize', 14);
saveeps(sprintf('least-squares-model-%d-res.eps', model));
```

Code Listing 9: exercise32.m

```
function m1 = M1(x, t)
    m1 = x(2)*exp(-x(1)*t) + x(3);
end
```

Code Listing 10: M1.m

```
function m2 = M2(x, t)
    m2 = x(3)*exp(-x(1)*t)+x(4)*exp(-x(2)*t)+x(5);
end
```

Code Listing 11: M2.m

```
function [r, J] = residual_jacobian_M1(x, t, y)
    r = M1(x,t) - y;
    J = [-x(2)*t.*exp(-x(1)*t) exp(-x(1)*t) ones(size(t))];
end
```

Code Listing 12: residual_jacobian_M1.m

```
function [r, J] = residual_jacobian_M2(x, t, y)
    r = M2(x,t) - y;
    J = [-x(3)*t.*exp(-x(1)*t), -x(4)*t.*exp(-x(2)*t), exp(-x(1)*t), ...
         exp(-x(2)*t), ones(size(t))];
end
```

Code Listing 13: residual_jacobian_M2.m

A.6 Exercise 4.1

```
function [xmin, X, F, DF, A] = linesearch(fns, x0, opts)
    %LINESEARCH : Perform a line search that finds a minimum  of the function
    %             represented by the input fns. An initial guess of the
    %             minimum must be given as the input x0. Various parameters
    %             concerning the execution of the linesearch can be controlled
    %             by passing options as the input opts.
    %
    % Call  [xmin, X, F, DF, A] = linesearch(fns, x0, opts)
    %
    % Input parameters
    % fns  :  Handle to the function to minimize. When the algorithm is
    %         'steepest-descent' or 'newton' the function should have the
    %         interface
    %
    %         [f, df, ddf] = fns(x)
```

```
%
%         where f is the function value at x. df is the gradient at x, and
%         ddf is the hessian at x. When the algorithm is 'quasi-newton' the
%         hessian is not needed and the interface should be
%
%         [f, df] = fns(x)
%
% x0   :  Initial guess for the minimum.
% opts :  Struct containing settings for the execution of the linesearch.
%         The available settings are
%         algorithm  :  Choose between 'steepest-descent', 'newton',
%                       'quasi-newton'. (default: 'steepest-descent')
%         maxiter    :  The maximum number of iterations performed.
%                       (default: 1000)
%         tol        :  The size the norm of the gradient should reach
%                       before the algorithm terminates. (default: 1e-5)
%         rho        :  Damping parameter used by the linesearch algorithm.
%                       (default: 0.9)
%         c          :  Parameter used for the sufficient decrease condition
%                       in the linesearch algorithm. (default: 1e-4)
%         hessian    :  Only used when the algorithm is quasi-newton. Set to
%                       the initial approximation of the hessian.
%                       (default: eye(length(x0)))
%
% Output parameters
% xmin  :  Minimum determined by the algorithm. Is an empty vector if the
%          algorithm terminated before the norm of the gradient reached the
%          tolerence set in opts.tol.
% X     :  Matrix containing all x values visited by the algorithm. Each x
%          value is a column in X so the maximum size of X is
%          "length(x0) x maxiter".
% F     :  Matrix containing function values for all x values visited.
% DF    :  Matrix containing gradient values for all x values visited.
% A     :  Matrix containing step sizes for all x values visited.


% Define default options
defaultopts = struct( ...
    'algorithm', 'steepest-descent', ...
    'maxiter', 1000, ...
    'tol', 1e-5, ...
    'rho', 0.9, ...
    'c', 1e-4, ...
    'hessian', eye(length(x0)) ...
);
% Overwrite default options with options given as input
opts = mergestructs(defaultopts, opts);

% Helpers to make later code more readable
is_steepest_descent = strcmp(opts.algorithm, 'steepest-descent');
is_newton = strcmp(opts.algorithm, 'newton');
is_quasi_newton = strcmp(opts.algorithm, 'quasi-newton');

% Check that algorithm exists
if ~(is_steepest_descent || is_newton || is_quasi_newton)
    error('Algorithm not recognized');
end

k = 0;
x = x0;
xmin = x0;
```

```
    % Initialize function value, gradient and hessian
    if is_quasi_newton
        [f, df] = fns(x);
        Hk = opts.hessian;
        I = eye(length(x));
    else
        [f,df,ddf] = fns(x);
    end

    X = [x0]; F = [f]; DF = [df]; A = [];

    converged = (norm(df,'inf') < opts.tol);

    while ~converged && k < opts.maxiter
        % Determine search direction
        if is_steepest_descent
            pk = -df;
        elseif is_quasi_newton
            pk = -Hk*df;
        else
            pk = -ddf\df;
        end

        % Find step length and update x
        a = backtracking(fns, x, pk, f, df, opts.rho, opts.c);
        sk = a*pk;
        x = x + sk;

        % Update function value, gradient and hessian
        if is_quasi_newton
            [f, df] = fns(x);
            yk = df - DF(:, end);
            rhok = 1/(yk'*sk);
            syk = sk*yk';
            Hk = (I - rhok*syk)*Hk*(I - rhok*syk') + rhok*sk*sk';
        else
            [f,df,ddf] = fns(x);
        end

        converged = (norm(df,'inf') < opts.tol);

        if (converged) xmin = x; else xmin = []; end

        k = k + 1;

        % Update history matrices
        A = [A, a]; X = [X,x]; F = [F,f]; DF = [DF,df];
    end

end
```

Code Listing 14: linesearch.m

```
function a = backtracking(fns, xk, pk, fk, dfk, rho, c)
    %BACKTRACKING : Perform a backtracking step length selection. Base on
    %               algorithm 3.1 on page 37 in "Numerical Optimization"
    %               (2nd ed. 2006) by Nocedal and Wright.
    %
```

```
    % Call  a = backtracking(fns, xk, pk, fk, dfk, rho, c)
    %
    % Input parameters
    % fns  :  Handle to the function to minimize.
    % xk   :  The current x iteration.
    % pk :  The search direction
    % fk : The current function value
    % dfk : The current gradient vector
    % rho : Damping parameter of a.
    % c :  Affects the stopping criteria
    %
    % Output parameters
    % a  :  The step length

    a = 1;
    k = 1;
    kmax = 100;
    fknew = fk;

    function converged = is_converged()
        fkstep = fns(xk+a*pk);
        tol = fk + c*a*dfk'*pk;
        converged = fkstep <= tol;
    end

    converged = is_converged();

    while ~converged && k<kmax
        a = rho*a;
        converged = is_converged();
        k = k + 1;
    end

end
```

Code Listing 15: backtracking.m

```
function s = mergestructs(org, update)
    names = fieldnames(update);
    for i=1:length(names)
        name = names{i};
        org = setfield(org, name, getfield(update, name));
    end
    s = org;
end
```

Code Listing 16: mergestructs.m

A.7 Exercise 4.3

```
% Define starting points
x0s = [ -1.2  1.2;
```

```
         1    1.2 ];
numx0 = size(x0s, 2);

% Define algorithms to test
algorithms = {'steepest-descent', 'newton', 'quasi-newton'};
numalgos = length(algorithms);

% Init options for linesearch function
o = struct();
o.tol = 1e-10;

% Init data structures to hold the results
xmins = cell(numx0, numalgos);
norms = cell(numx0, numalgos);
xs = cell(numx0, numalgos);

% Iterate all combinations of starting point and algorithm
for x0ind = 1:numx0
    x0 = x0s(:, x0ind);
    for algoind = 1:numalgos
        o.algorithm = algorithms{algoind};
        [xmin, X, F, DF] = linesearch(@rosenbrock, x0s(:,x0ind), o);
        xmins{x0ind, algoind} = xmin;
        xs{x0ind, algoind} = X;
        % Calculate gradient norms
        norms{x0ind, algoind} = sqrt(DF(1,:).^2 + DF(2,:).^2);
    end
end
```

Code Listing 17: exercise43.m

```
% This script should only be called after exercise43.m have been run
% as this script relies on data calculated by exercise43.m

% Iterate all combinations of starting point and algorithm
for x0ind = 1:numx0
    for algoind = 1:numalgos
        X = xs{x0ind, algoind};
        filenametmpl = sprintf('%s-start%d', algorithms{algoind}, x0ind);
        normarray = norms{x0ind, algoind};
        n = length(normarray);

        % Save gradient norms as latex table
        f = fopen([filenametmpl, '.tex'], 'w');
        for i = (n-4):n
            fprintf(f, '%d & %.3e\n', i-1, normarray(i));
        end
        fclose(f);

        % Plot gradient norms as function of iteration number
        % and save as eps
        set(gca, 'fontsize', 16);
        toplot = max(1, n-30):n;
        semilogy(toplot, normarray(toplot), 'r^', 'markersize', 8, 'linewidth', 2);
        titlestr = sprintf('Algorithm: %s, x_0=(%.02f, %.02f)', ...
                           algorithms{algoind}, x0s(1,x0ind), x0s(2, x0ind));
        title(titlestr, 'fontsize', 16);
        xlabel('Iteration no.'); ylabel('Norm of gradient');
```

```
        saveeps([filenametmpl, '.eps']);

        % Plot the convergence on contour plot
        plot_rosenbrock_convergence(X, titlestr);
        saveeps([filenametmpl, '-contour.eps']);
    end
end
```

Code Listing 18: exercise43_save.m

```
function [f, df, ddf] = rosenbrock(x)
    f = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;

    df = [-400*x(1)*(x(2)-x(1)^2)-2*(1-x(1)) ;
          200*(x(2)-x(1)^2)];

    ddf = [-400*x(2)+1200*x(1)^2+2  -400*x(1) ;
           -400*x(1)  200];
end
```

Code Listing 19: rosenbrock.m

```
function [] = plot_rosenbrock_convergence(Xs, titlestr)
    % Helper function used in the scripts exercise43_save and exercise44_save
    % This function plot contour of the rosenbrock function and the iteration
    % path given in input parameter Xs

    % Determine plot limits based on the iteration points
    margins = 0.2;
    mins = min(Xs, [], 2) - margins;
    maxs = max(Xs, [], 2) + margins;
    x = linspace(max(-2, mins(1)), min(2, maxs(1)), 100);
    y = linspace(max(-1, mins(2)), min(1.5, maxs(2)), 100);

    % Calculate function values for contour plot
    v = -10:0.1:10;
    [X, Y] = meshgrid(x, y);
    f = arrayfun(@(x, y)rosenbrock([x;y]), X, Y);

    % Plot contour
    contour(X, Y, log(f), v, 'linewidth', 1);
    colorbar;
    set(gca, 'fontsize', 16);
    hold on;

    % Plot iteration path
    plot(Xs(1,:), Xs(2,:), 'k^', 'markersize', 10, 'linewidth', 2);
    plot(Xs(1,:), Xs(2,:), 'k-', 'linewidth', 2);
    hold off;
    title(titlestr, 'fontsize', 16);
    xlabel('x_1'); ylabel('x_2');
end
```

Code Listing 20: plot_rosenbrock_convergence.m

A.8 Exercise 4.4

```
function [xs, norms] = exercise44()
    % Define starting points
    x0s = [ -1.2  1.2;
             1    1.2 ];
    numx0 = size(x0s, 2);
    % Define algorithms to test
    algorithms = {'quasi-newton', 'steepest-decent'};
    numalgos = length(algorithms);

    % Create data structures for the results
    norms = cell(numx0, numalgos);
    xs = cell(numx0, numalgos);

    % Used to get iteration results out of fminunc
    function stop = outfun(x, optim, state)
        if strcmp(state, 'iter')
            X = [X x];
            DF = [DF optim.gradient];
        end
        stop = false;
    end

    % When largescale is off a quasi-newton algorithm is run.
    % Setting HessUpdate to steepdesc gives the steepest descent algo.
    % See http://www.mathworks.se/help/toolbox/optim/ug/fminunc.html
    o = {};
    o{1} = optimset('gradobj', 'on', 'largescale', 'off', 'maxiter', 1000, ...
                    'outputfcn', @outfun, 'maxfuneval', 3000, ...
                    'initialhesstype', 'identity');
    o{2} = optimset(o{1}, 'hessupdate', 'steepdesc');

    % Iterate all combinations of starting points and algorithms
    for x0ind = 1:numx0
        x0 = x0s(:, x0ind);
        for algoind = 1:numalgos
            X = []; DF = [];
            fminunc(@rosenbrock, x0s(:,x0ind), o{algoind});
            xs{x0ind, algoind} = X;
            norms{x0ind, algoind} = sqrt(DF(1,:).^2 + DF(2,:).^2);
        end
    end
end
```

Code Listing 21: exercise44.m

```
% Should only be called after the function exercise44 have been run and the
% results have been saved in the variables xs and norms. This script relies
```

```
% on these two variables being available

% Define starting points
x0s = [ -1.2  1.2;
         1    1.2 ];
numx0 = size(x0s, 2);
% Define algorithms
algorithms = {'quasi-newton', 'steepest-descent'};
numalgos = length(algorithms);

% Iterate all combinations of starting points and algorithms
for x0ind = 1:numx0
    for algoind = 1:numalgos
        X = xs{x0ind, algoind};
        filenametmpl = sprintf('%s-start%d-fminunc', algorithms{algoind}, x0ind);
        normarray = norms{x0ind, algoind};
        n = length(normarray);

        % Save gradient norms as latex table
        f = fopen([filenametmpl, '.tex'], 'w');
        for i = (n-4):n
            fprintf(f, '%d & %.3e\n', i-1, normarray(i));
        end
        fclose(f);

        % Plot gradient norm as function of iteration number
        set(gca, 'fontsize', 16);
        toplot = max(1, n-30):n;
        semilogy(toplot, normarray(toplot), 'r^', 'markersize', 8, 'linewidth', 2);
        titlestr = sprintf('Algorithm: %s (fminunc), x_0=(%.02f, %.02f)', ...
                           algorithms{algoind}, x0s(1,x0ind), x0s(2, x0ind));
        title(titlestr);
        xlabel('Iteration no.'); ylabel('Norm of gradient');
        saveeps([filenametmpl, '.eps']);

        % Plot iteration path on contour plot
        plot_rosenbrock_convergence(X, titlestr);
        saveeps([filenametmpl, '-contour.eps']);
    end
end
```

Code Listing 22: exercise44_save.m

A.9 Exercise 4.5

```
function [xs, norms] = exercise45()
    % Define starting points
    x0s = [ -1.2  1.2;
             1    1.2 ];
    numx0 = size(x0s, 2);

    % Define data structures to hold results
    norms = cell(numx0, 1);
    xs = cell(numx0, 1);

    % Iterate starting points
```

```
    for x0ind = 1:numx0
        x0 = x0s(:,x0ind);
        X = []; DF = [];

        % Calculate results
        [xmin, X, F, DF, A] = gaussnewton(@rosenbrock_sq, x0, struct());
        xs{x0ind,1} = X;
        norms{x0ind,1} = sqrt(DF(1,:).^2 + DF(2,:).^2);

        % Plot and save gradient norm as function of iteration number
        filenametmpl = sprintf('gauss-newton-start%d', x0ind);
        set(gca, 'fontsize', 16);
        semilogy(norms{x0ind,1}, 'r^', 'markersize', 8, 'linewidth', 2);
        titlestr = sprintf('Algorithm: Gauss-Newton, x_0=(%.02f, %.02f)', ...
                           x0(1), x0(2));
        title(titlestr);
        xlabel('Iteration no.'); ylabel('Norm of gradient');
        saveeps([filenametmpl, '.eps']);

        % Plot and save contour plot with iteration path
        plot_rosenbrock_convergence(X, titlestr);
        saveeps([filenametmpl, '-contour.eps']);
    end
end
```

Code Listing 23: exercise45.m

```
function [r, J] = rosenbrock_sq(x)
    r = [10*x(2)-10*x(1)^2;  1-x(1)];
    J = [-20*x(1)  10;
         -1  0];
end
```

Code Listing 24: rosenbrock_sq.m

A.10 Helper function

```
function [] = saveeps(filename)
    print('-depsc', '-loose', filename);
end
```

Code Listing 25: saveeps.m

References

[1] Jorge Nocedal & Stephen J. Wright, *Numerical Optimization*. Springer Science+Business Media, 2nd Edition, 2006.

[2] Kaj Madsen & Hans Bruun Nielsen, *Introduction to Optimization and Data Fitting*. DTU IMM, 1st Edition, 2010.

[3] Hans Bruun Nielsen, *Checking Gradients*. DTU IMM, 1st Edition, 2000, `http://www2.imm.dtu.dk/~hbn/Software/checkgrad.ps`.