02610 Optimization and Data Fitting
Assignment 1

Ramiro Mata

October 16 2016

1 Fitting to Air Pollution Data

In this assignment, we are given measurements of air pollution (NO) gathered from a busy street in Copenhagen over a period of 24 hours. Since the measurements are taken hourly, we would like to fit a model (i.e. fit a smooth curve to the measurements) in order to compute the [NO] at any given point in time over the 0 to 24 hour interval.

We assume that the data given represents the true signal plus the data errors. The latter captures errors in the measurement device (i.e. noise) as well as random variations inherent to the physical process that creates the data. In our case, the physical process is the "automobile-traffic system."

Our data fitting goal is to create a parametric model that captures the behavior of this system, and which gives reliable estimates of the measured data (without being too sensitive to the data errors). We create the model by employing a least squares fit (LSQ) approach, which minimizes the sum of the squared residuals in order to find the best-fit model parameters.

Under the LSQ approach, we make the following assumptions:

Assumption 1 The data and the errors are independent.

Assumption 2 The data errors are "white noise", i.e. random and uncorrelated, have a mean of zero, have the same variance and are normal distributed.

1.1 Matlab Code

We create a function (NOfit.m) that computes the Least Squares Fit of the data given. See Appendix B for the function's code.

1.2 Testing the Software

We test our matlab function for a model of order n = 3, which should not be a reliable model. We obtain the same residual norm of 292.558 as provided by the instructors. Figure 1 below shows how the model approximates the observed data, and lists its solution coefficients ($x* = [186.806 - 44.936 - 93.429]$).

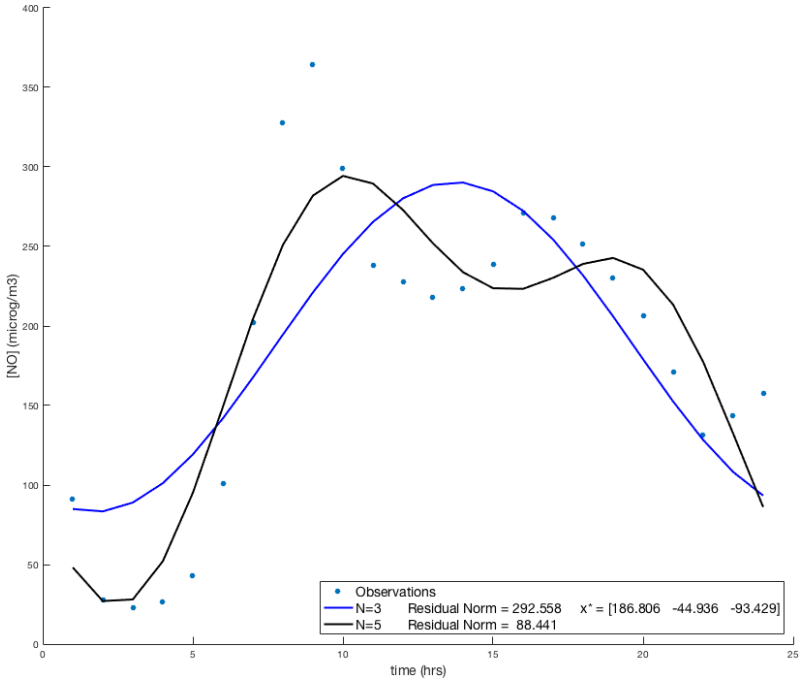

Figure 1: LSQ Model fit for n = 3 (blue line) and n = 5 (black line). The figure shows how the model with 3 features fails to capture the bimodal structure of the observed data. The model with 5 features better captures the behavior, but still fails to provide good estimates of the observed data.

1.3 The Optimal Order of the Fit

We want our model to capture the main behavior of the observed data, which consists of the pure data plus the data error (ideally, we would want our model to capture the pure-data function only). The residuals play an important role in evaluating how well the model does this.

The residuals consist of two things: the data error (e_i) and the approximation error ($\Gamma(t_i) - M(x, t_i)$).

$$r_i = e_i + (\Gamma(t_i) - M(x, t_i)) \tag{1}$$

Because the data error and the approximation error have different statistical properties, and these properties carry over to the residuals, we can inspect the residuals to determine how well the model performs.

We can not calculate the approximation errors because we do not have prior knowledge of the pure-data function. However, in the case where the approximation errors dominate the residuals, we would expect the residuals to behave more like the approximation errors. Generally speaking, the residuals will behave like a sampled signal (i.e. show trends and strong local correlations) and not like noise.

On the other hand, if data errors dominate the residuals, then we would expect the statistical properties of the data errors to carry over to the residuals. In that case, the residuals should behave more like white noise. As such, in our search for the best fit and optimal order n of that fit, we can analyze the residuals pertaining to each model and ask:

1) Do the residuals behave like noise?

2) (Or alternatively) Do our residuals have strong, local correlations and/or trends?

To answer these questions we perform a random signs test and an autocorrelation test.

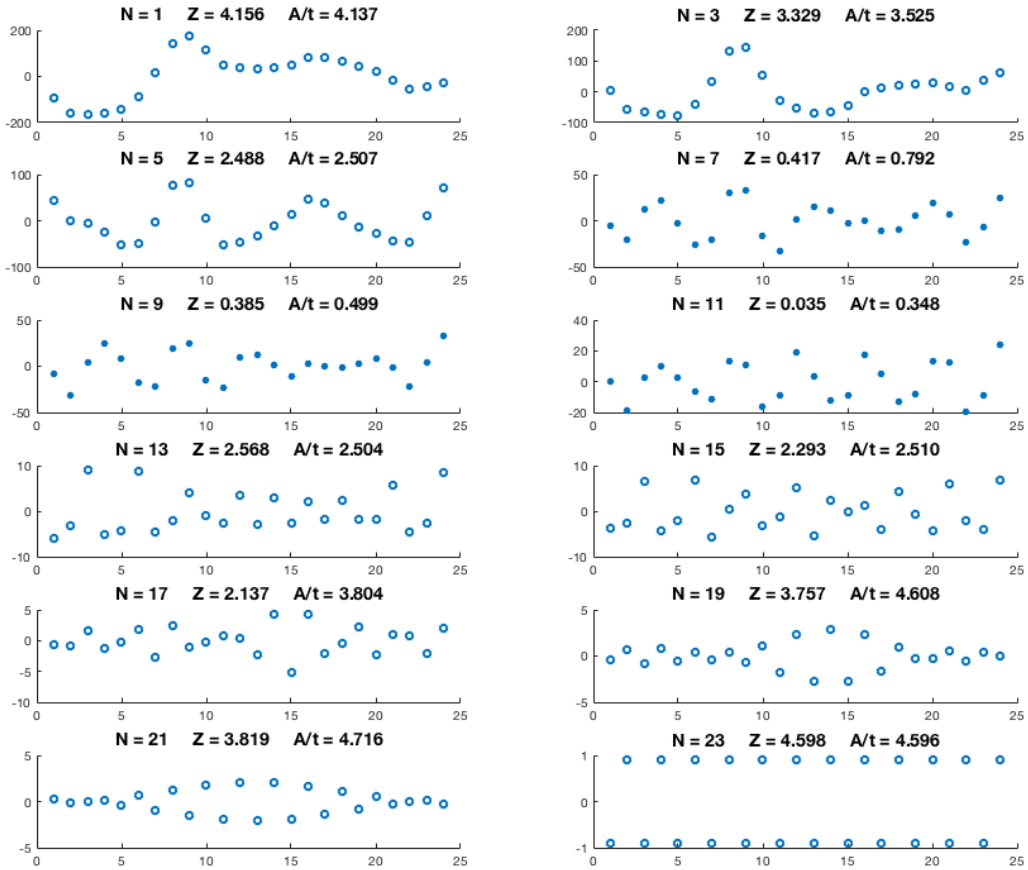

Figure 2: The figures above show the residuals over time for each model of n'th order. The y-axis describes the magnitude of the residuals while the x-axis represents time in hours. Only the models of order 7, 9 and 11 (with filled blue bubbles) meet the criteria for randomness of signs and absence of trends.

Figure 2 shows the residuals as a function of time for models of different orders. The Z-Scores above each plot describe that the randomness of signs can be considered random at a 5 percent significance level if the Z-Score is less than 1.96. Only the models of order 7, 9 and 11 fit this criterion.

The autocorrelation over trend threshold ratio (A/T) is also shown for each model. Values greater than 1 indicate a trend is likely to be present in the residuals. Again, we observe that only the models of order 7, 9 and 11 fit this criterion.

We can see in figure 1 that when n is 3 and 5, the models fails to capture the behavior of the observed data, and hence we can conclude that the approximation errors are too

large (bigger than the data errors). The residuals' tests support this. A visual inspection of figure 2 shows obvious trends and the signs do not look random whatsoever.

Models with n greater than 11 resemble the observed data the most (see figure 4), and have the smallest residuals (see table 1). However, while we want our model to capture the main behavior of the observed data as much as possible, we don't want the model that most resembles the observed data. Choosing a model based on having the smallest residuals endangers suffering from overfitting, where the model adapts to the data errors as well.

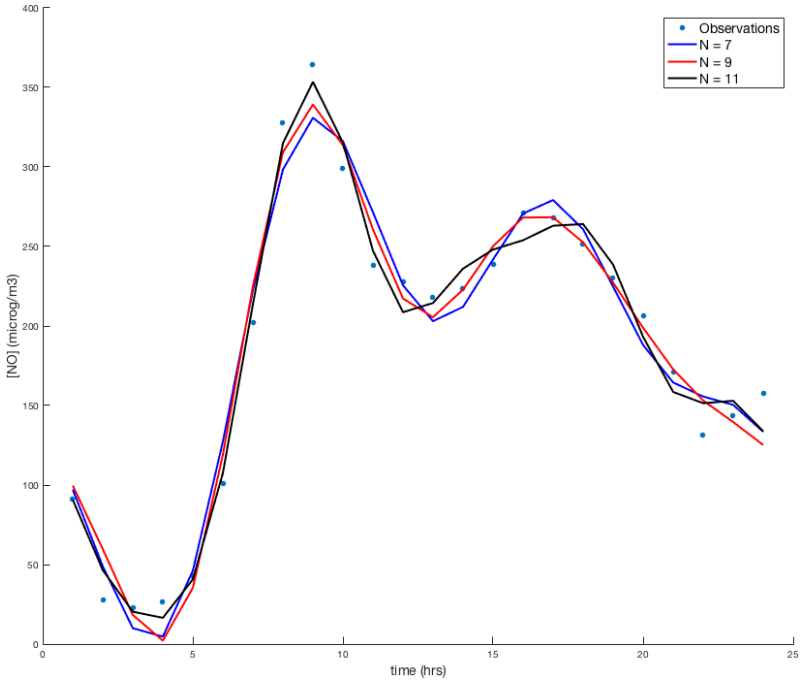

Figure 3: LSQ model fit of n order 7, 9 and 11. The figure shows how these models capture the main behavior of the observed data without following it too closely. This leaves room for data error as opposed to adapting to the data error, which is undesirable.

Indeed, for models n greater than 11, overfitting is a concern. They resemble the observed data too closely (see figure 4). The residuals do not look random either (save possibly for n = 15 albeit the test values say otherwise).

Figure 5 neatly captures this behavior. If we look at the scaled residual norms as a function of n, we should initially expect steep reductions as n increases since the model can better "match" the observed data with each increasing n. This steep decline embodies a diminishing approximation error. Afterwards, with larger n values, a plateu is reached and no longer the approximation error dominates the residuals, but rather the data error does. According to Chapter 2 of Least Squares Data Fitting with Applications, the transition between these two stages indicates the optimal choice for order n.

This transition in figure 5 corresponds to n values of 7, 9 and 11, which coincide with the results of the residuals' tests. Given the arguments of approximation error dominance (for n = 3 and n = 5) and overfitting (for n = 13 and beyond), we conclude that 7, 9 and 11 are good candidates for optimal n order. Based on the residual tests, we choose model n order 11 as our optimal choice.

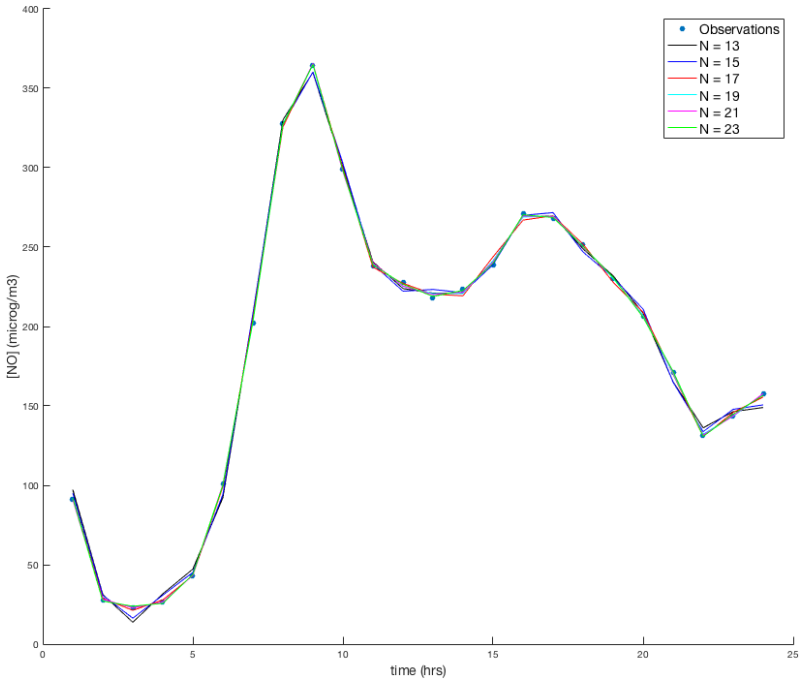

Figure 4: LSQ model fit of n orders 13 to 23. The figure shows how these models follow the observed data too closely and risk adapting the behavior of the data error as well. This can result in overfitting.

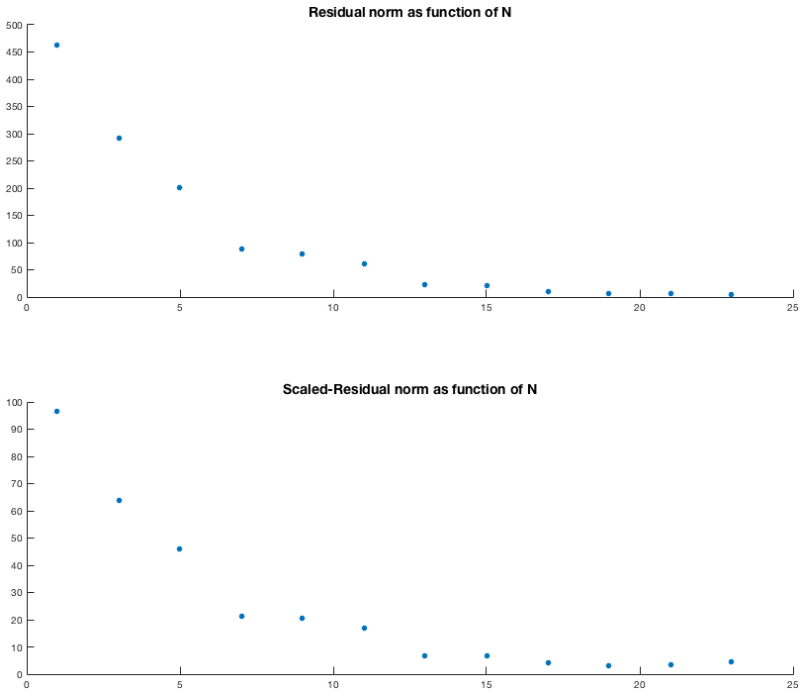

Figure 5: The residual norm and its scaled version here are shown as a function of n. The figure shows how initially (for n = 1,3 and 5) the residuals (and the aproximation error) decrease rapidly as each additional n is able to fit a better model. After n = 7, the residuals flatten and the data error dominates the residuals. The transition between these two stages (from n =7 to n= 11) is an optimal zone for the order of the model.

n	Residual Norm	Scaled Residual Norm
1	463.22	96.59
3	292.56	63.84
5	201.15	46.15
7	88.44	21.45
9	79.27	20.47
11	61.59	17.08
13	22.19	6.69
15	20.34	6.78
17	10.78	4.07
19	6.72	3.00
21	6.00	3.47
23	4.47	4.47

Table 1

1.4 Estimating the Standard Deviation of the Solution Coefficients

The scaled residual norm plays also another important role in assessing the uncertainty of our x* solution coefficients. In fact, if we assume that our residuals of our optimal n are dominated by data errors (as explained in section 1.3), then we can use the scaled residual norm as an estimate of the standard deviation of the noise (sigma).

Section 1.3 and figure 5 explain and depict respectively, that the transition between a diminishing approximation error and the plateu marks the area for an optimal n order, where the residuals are dominated by the data error.

The sigma of the noise in turn, allows us to calculate the standard deviation of our x* solution coefficients with the following equation:

$$\mathrm{Cov}(x^*) = \zeta^2 (A^T A)^{-1} \tag{2}$$

Table 2 lists the x* solution coefficients for model of n order 11 along with their respective standard deviations.

i	x_i^*	Std. Deviation
1	186.81	3.49
2	-44.94	4.93
3	-93.43	4.93
4	-60.89	4.93
5	-7.28	4.93
6	21.81	4.93
7	47.37	4.93
8	7.69	4.93
9	-8.30	4.93
10	-11.54	4.93
11	8.61	4.93

Table 2: Table 2

1.5 Concluding remarks

By analyzing the residuals of the different models, we were able to choose an optimal one. Based on the assumptions of the LSQ fit approach, we knew that the statistical properties of the residuals should behave more like white noise (data error) as opposed to a sampled signal (approximation error). By performing a randomness of sings and an autocorrelation to trend threshold tests, we were able to evaluate the statistical properties of the residuals for each model. This guidance allowed us to pick an optimal model. In addition, the scaled residual norm for our optimal model served as an estimate for the standard deviation of the noise, which in turn allowed us to compute the covariance matrix of the solution coefficients (x*). This allowed us to approximate the standard deviation of each solution coefficient to get an understanding of the uncertainty involved in each coefficient. We can see from table 2 that this uncertainty becomes an issue for the 5th, 6th, and 8th to 11th coefficients. While a model of order n=7 provides as a whole coefficients with less uncertainty (the first seven on table 2), we ultimately opted for n=11 because the magnitude of the residual norm is significantly lower. If computational complexity was a concern, then model n=7 should provide a better alternative. However, with the given optimization task at hand, the computational demands of either of these models are not a concern, and model of n order 11 should provide better estimates to the observed data, yet without adapting to the data error.

2 Algorithms for Nonlinear Unconstrained Optimization

In this part, optimization algorithms will be tested using the Himmelblau function

$$f(x) = f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \tag{3}$$

A contour plot of the Himmelblau function can be seen in figure 6. For finding the stationary points the gradient has to be determined

$$\nabla f(x) = \begin{pmatrix} \frac{\delta}{\delta x_1} f(x) \\ \frac{\delta}{\delta x_2} f(x) \end{pmatrix}$$

$$\nabla f(x) = \begin{pmatrix} 4x_1(x_1^2 + x_2 - 11) + 2(x_1 + x_2^2 - 7) \\ 2(x_1^2 + x_2 - 11) + 4x_2(x_1 + x_2^2 - 7) \end{pmatrix} = \begin{pmatrix} 4x_1^3 + 4x_1x_2 - 42x_1 + 2x_2^2 - 14 \\ 2x_1^2 - 26x_2 + 4x_1x_2 + 4x_2^3 - 22 \end{pmatrix}$$

Solving $\nabla f(x) = \binom{0}{0}$ in WolframAlpha[1] yields the stationary points, which can be seen in table 3. The Hessian is found as the second order partial derivative

$$\nabla^2 f(x) = \begin{pmatrix} \frac{\delta^2}{\delta x_1^2} f(x) & \frac{\delta^2}{\delta x_1 \delta x_2} f(x) \\ \frac{\delta^2}{\delta x_2 \delta x_1} f(x) & \frac{\delta^2}{\delta x_2^2} f(x) \end{pmatrix}$$

$$\nabla^2 f(x) = \begin{pmatrix} \frac{\delta}{\delta x_1} 4x_1^3 + 4x_1x_2 - 42x_1 + 2x_2^2 - 14 & \frac{\delta}{\delta x_1} 2x_1^2 - 26x_2 + 4x_1x_2 + 4x_2^3 - 22 \\ \frac{\delta}{\delta x_2} 4x_1^3 + 4x_1x_2 - 42x_1 + 2x_2^2 - 14 & \frac{\delta}{\delta x_2} 2x_1^2 - 26x_2 + 4x_1x_2 + 4x_2^3 - 22 \end{pmatrix}$$

$$\nabla^2 f(x) = \begin{pmatrix} 12x_1^2 + 4x_2 - 42 & 4(x_1 + x_2) \\ 4(x_1 + x_2) & 12x_2^2 + 4x_1 - 26 \end{pmatrix}$$

By inserting each stationary point into the Hessian and finding the eigenvalues it can be determined whether each stationary point is a local minimum, local maximum or a saddle point. These calculations are elaborated in Appendix A and the results are summarized in table 3 and figure 6.

#	x_1	x_2	type
1	3	2	min
2	-3.77931	-3.28319	min
3	3.58443	-1.84813	min
4	-2.80512	3.13131	min
5	-0.127961	-1.95371	saddle
7	-3.07303	-0.0813530	saddle
8	3.38515	0.0738519	saddle
9	0.0866775	2.88425	saddle
6	-0.270845	-0.923039	max

Table 3

[1] using the command `stationary points | (x1^2+x2-11)^2+(x1+x2^2-7)^2`

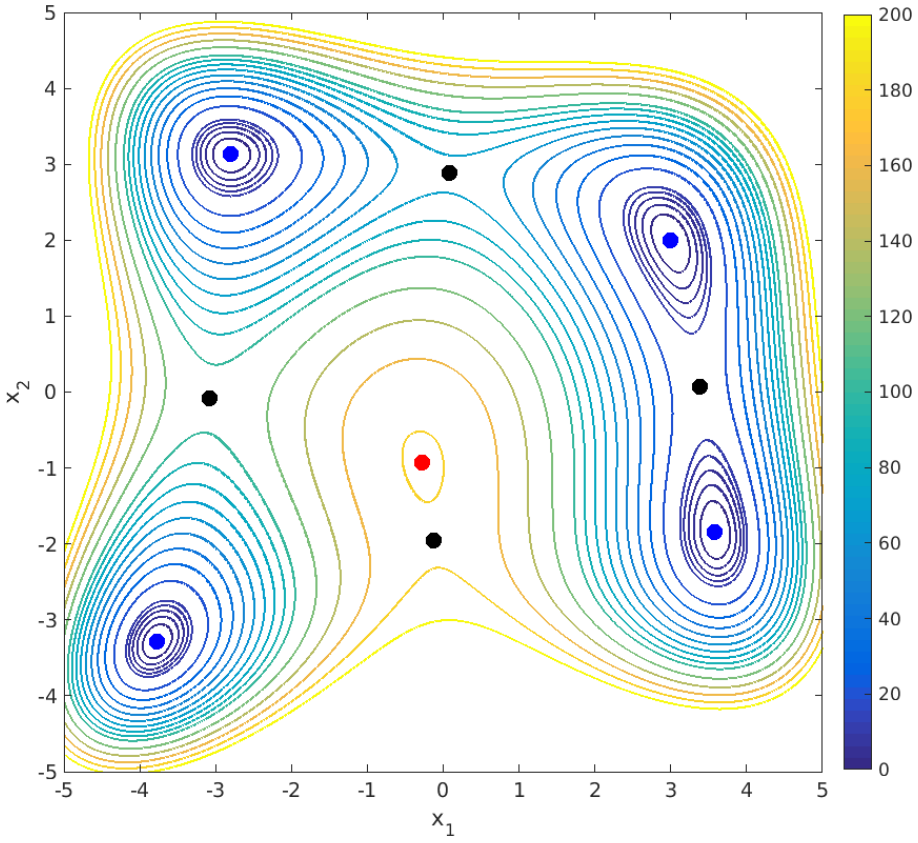

Figure 6: Contour plot of the Himmelblau function. The blue dots are local minima, the black dots are saddle points and the red dot is a local maximum.

2.1 Algorithms used

Minima of the Himmelblau function will be found using four different algorithms – steepest descent, Newton's algorithm, Quasi-Newton algorithm and Gauss-Newton algorithm – and one trust region method – the Levenberg-Marquardt algorithm. Having obtained the search direction the step length in each iteration will be determined using two different approaches – Backtracking and Soft line search.

For each of the employed optimization algorithms a table will be presented with number of iterations, function evaluations and computational time (mean time in ms from 100 repetitions). For the line search methods the table will include data for both the backtracking algorithm and the soft line search.

All code used can be found in Appendix C.

Backtracking: This method determines the maximum step length that gives a substantial decrease along the given search direction. The algorithm, which is based in the Goldstein-Armijo condition, starts with a large step length and reduces it in each iteration while the sufficient decrease condition is not satisfied.

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k' p_k \tag{4}$$

The simplicity of this line search method is probably its main advantage. Being computationally efficient allows to use most of the machine resources to find the optimal search direction.

We wrote the algorithm and will be using it in the line search methods along with a soft line search algorithm in order to compare them and be able to point out their advantages and disadvantages.

Soft line search: Evaluating the function along the direction of search in order to find a step length that will give a sufficient decrease in the function value, before actually moving to the next iteration, has been proved to be greatly efficient in most optimization problems.

Soft line search algorithms try to find a step length that first satisfies the sufficient decrease condition (equation 4) and second prevents the algorithm from taking too short steps (equation 5). While the former makes sure that we are moving towards a minimum, the latter, known as curvature condition, tries to avoid unnecessary iterations knowing that larger steps will lead us faster to the solution.

$$\nabla f(x_k + \alpha p_k)' p_k \geq c_2 \nabla f_k' p_k \tag{5}$$

Line search algorithms start from an initial interval and reduce it in order to get to a point that matches the mentioned criteria, also known as Wolfe conditions.

As we said before, it is not advisable to look too close into the interval because it may waste resources that could better be used to find a more optimal search direction. Accordingly, we decided to include in our code a soft line search algorithm that can be found in IMM Optibox.

2.2 Steepest descent

This algorithm is mainly based in the intuitive thinking that the fastest way to get down a slope is by choosing the direction where it decreases most rapidly, that is, the direction whose slope is minimum.

For a continuous and differentiable function we know that this direction is given by $-\nabla f_k$, which can be shown to be orthogonal to the contours of the function.

The direction, which changes every time depending on the place where the previous point has landed, guarantees to produce a decrease in the function. Knowing this, we can choose our step length in each iteration using a line search algorithm and find a local minimum to the function after some iterations.

We have implemented and tested our steepest descent algorithm from different starting points and using two different line search methods (backtracking and soft line search). Figure 7 shows a contour plot of Himmelblau function where the red and black lines represent the convergence sequence of the algorithm for the two line search methods from three different starting points (1,1), (1,-1) and (-1,0).

The number of iterations and line search evaluations are presented in table 4. We can see from the table and also from the graph in figure 8 that the soft line search algorithm requires fewer iterations and function evaluations to find a minimum. However, the elapsed

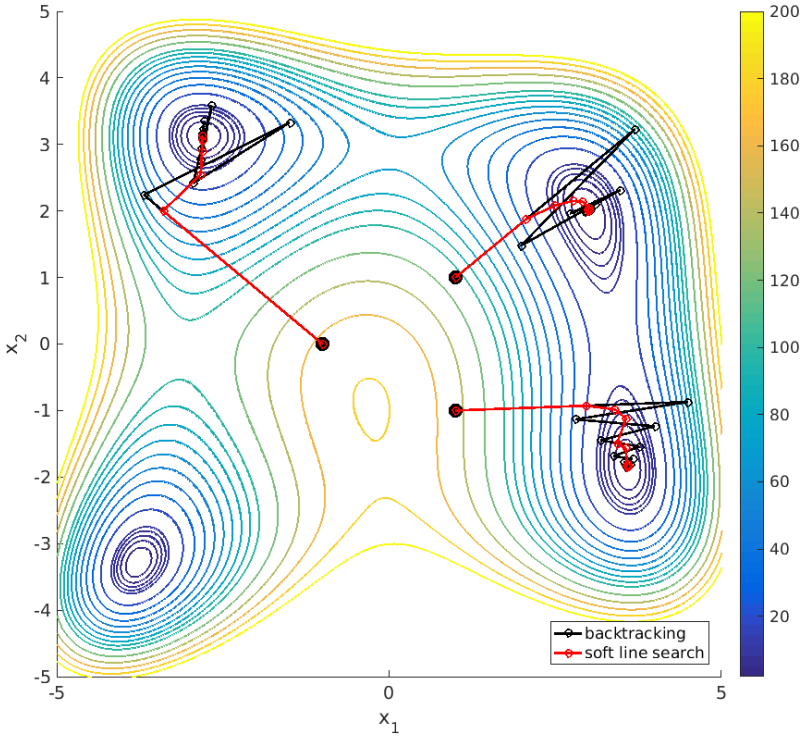

Figure 7: The steepest descent algorithm on the Himmelblau function from the starting point (1,1), (1,-1) and (-1,0).

time is close to the same for both the soft line search and the backtrack algorithm, which means that even though the number of line search evaluations is obviously smaller, soft line search is computationally more demanding than the backtracking algorithm for each step. Even though each iteration is quicker with the backtracking algorithm, the soft line search is still faster for all the examined starting points.

Figure 8 indicates that the steepest descent algorithm has a linear rate of convergence for both backtracking and soft line search, which is consistent with the provided material.

	$x_0 = (1,1)$			$x_0 = (1,-1)$			$x_0 = (-1,0)$		
Method	Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
Backtracking	66	3326	24.0	64	3028	19.2	65	3145	20.4
Soft line search	46	182	17.1	49	232	16.9	21	92	7.4

Table 4: Summary of the steepest descent algorithm on the Himmelblau function

2.3 Newton's algorithm

The method derives from Taylor's quadratic approximation of a function in some neighborhood of a given point.

$$f(x_k + p) \approx f_k + p'\nabla f_k + \frac{1}{2}p'\nabla^2 f_k p \tag{6}$$

If we assume that $\nabla^2 f_k$ is positive definite, Newton direction can be found by minimizing

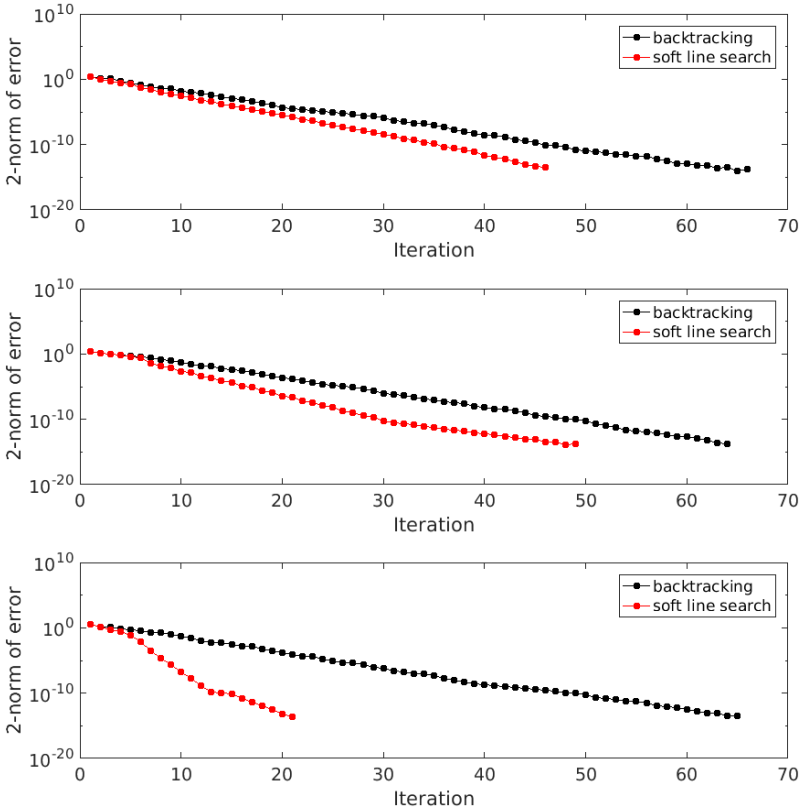

Figure 8: Error plot for the steepest descent algorithm on the Himmelblau function. Plots are from starting points (1,1), (1,-1) and (-1,0) respectively.

the equation above. The minimizer of the approximation is nothing but the next point in the iteration and therefore we can write newton direction as:

$$p_k = -(\nabla^2 f_k)^{-1} \nabla f_k \tag{7}$$

If the Hessian ($\nabla^2 f$) stays always positive definite Newton direction can be used in order to find a local minimizer of the function. The method will be particularly favourable when the function and its quadratic approximation are close to each other. The problem arises when the Hessian matrix is not positive definite, in those cases the direction can be no longer thought to satisfy the descent condition.

Computing the Hessian is known to be computationally demanding and although Newton's method presents usually quadratic rate of convergence this could cause a great problem whenever the resources are poor or limited.

We have implemented the algorithm for Himmelblau function, and again we have used the two line search methods to find the optimal step length. The sequence of convergence for the two methods is represented in figure 9 where the starting points[2] are (-2,0), (1,-1) and (-4,-2). In this particular case we see that the method does not even converge when using soft line search and starting from (1,-1). A major problem of Newton method is its poor rate of global convergence. In this example, Newton direction is pointing towards

[2]Different starting points where chosen for Newton's algorithm compared to the other ones used, since the soft line search did not converge for any of the ones previously chosen.

a positive slope and therefore soft line search algorithm is unable to find a point that satisfies the sufficient decrease condition.

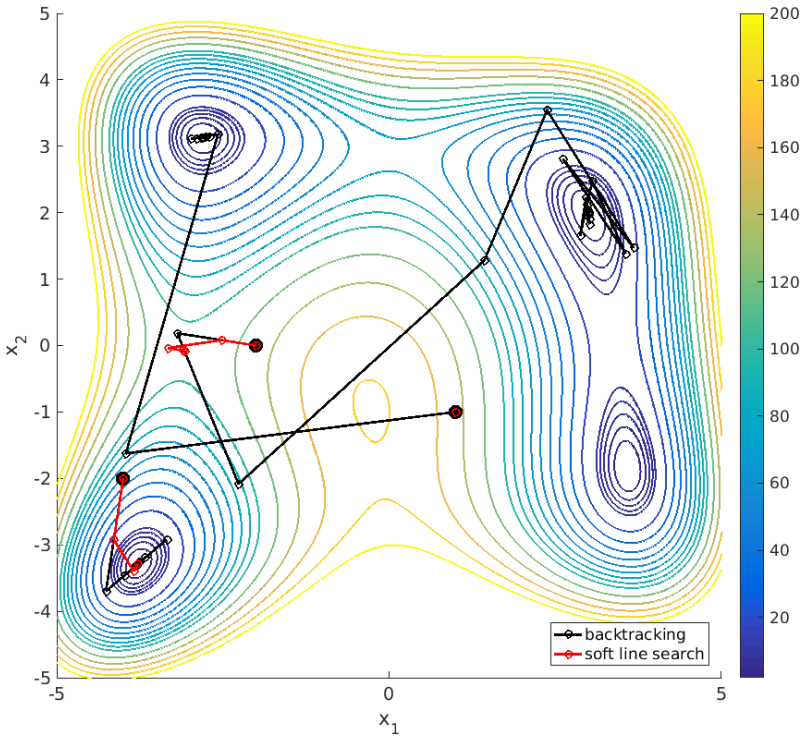

Figure 9: Newton's algorithm on the Himmelblau function from the starting point (-2,0), (1,-1) and (-4,-2).

Using backtracking algorithm we set the initial step length to five in order to see if we could overcome the positive slope. In figure 9 it is seen that the backtracking algorithm converges for all of the starting points, but for (1,-1) and (-2,0) it converges to a stationary point which is far from the starting point.

When starting from (-2,0) the soft line search sequence ends up finding a saddle point. Converging towards a maximizer or a saddle point is known to be another disadvantage of Newton method.

Table 5 summarizes the performance for the three different starting points. Now the timing values are much shorter when the algorithm finds a solution using soft line search. This means that the use of backtracking can be only justified whenever the initial step length needs to be very large to make the algorithm converge.

	$x_0 = (-2, 0)$			$x_0 = (1, -1)$			$x_0 = (-4, -2)$		
Method	Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
Backtracking	70	3054	22.0	61	3051	21.3	57	2822	22.7
Soft line search	6	7	2.0	10000	0	2455.1	6	8	2.2

Table 5: Summary of Newton's algorithm on the Himmelblau function

The convergence rate is illustrated in figure 10. Newton's method present clear quadratic rate of convergence when using the soft line search for the two starting points where it

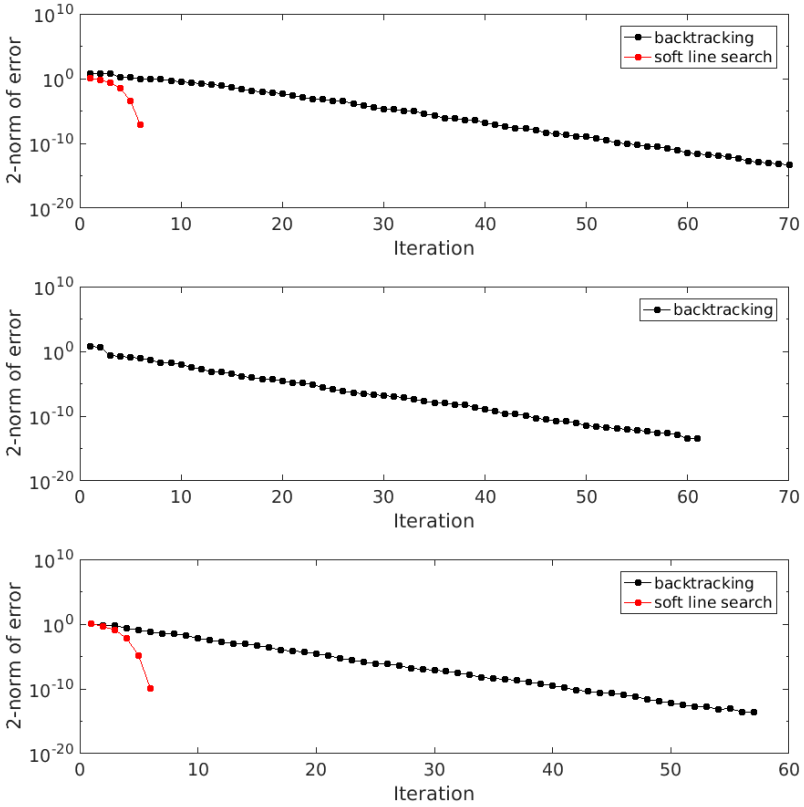

Figure 10: Error plot for Newton's algorithm on the Himmelblau function. Plots are from starting points (-2,0), (1,-1) and (-4,-2) respectively.

does converge – (-2,0) and (-4,-2). For the backtracking algorithm the convergence rate seems linear.

2.4 Quasi-Newton algorithm

We have already pointed out that one of the main inconveniences of Newton's method is the large employ of computer resources, mainly derived from computing the Hessian. Quasi-Newton algorithms appeared in order to optimize such demanding computations. These methods try to give an approximation using the gradient in each iteration which is often much faster to calculate. This alternative usually requires less CPU-time while keeping a similar rate of convergence.

In the first iteration the Hessian is substituted by the identity matrix which makes the search direction become the gradient or steepest descent direction. This is not an issue, since the steepest descent normally shows a great rate of global convergence for points which are far from minimums.

The approximations gather from the fact that changes in the gradient (∇f) give information about the Hessian along the search direction. From Taylor's approximation of a region where the Hessian is positive definite, it can be shown that:

$$\nabla^2 f_{k+1}(x_{k+1} - x_k) \approx \nabla f_{k+1} - \nabla f_k \tag{8}$$

Equation (8) is known as the secant condition and can also be written as:

$$B_{k+1}s_k = y_k \tag{9}$$

Where:

$$s_k = x_{k+1} - x_k \tag{10}$$
$$y_k = \nabla f_{k+1} - \nabla f_k \tag{11}$$

This, along with the conditions that B has to preserve the hessian symmetry and that the difference between two successive approximations $(B_k - B_{k+1})$ must have low rank, allowed to formulate different approximations to the Hessian matrix. In our algorithm we used the BFGS updating formula:

$$B_{k+1} = B_k - \frac{B_k s_k s_k' B_k}{s_k' B_k s_k} + \frac{y_k y_k'}{y_k' s_k} \tag{12}$$

Figure 11 illustrates the set of steps that the algorithm takes before finding a minimum from (1,1), (1,-1) and (-1,0). The plot shows that Newton's method lack of convergence has now disappeared and the iteration process is able to determine a local minimum from all the examined starting points.

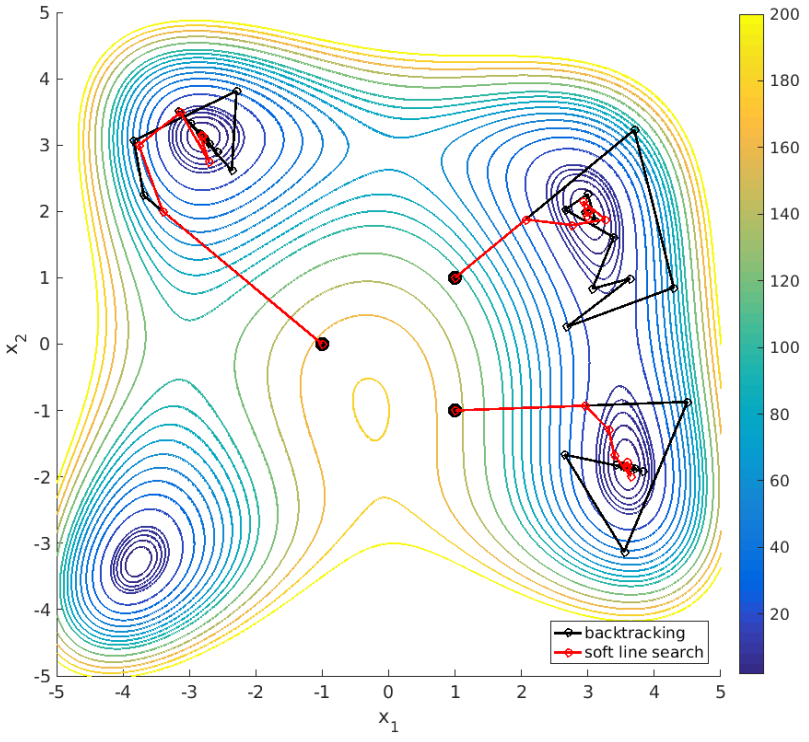

Figure 11: Quasi Newton algorithm on the Himmelblau function from the starting point (1,1), (1,-1) and (-1,0).

From figure 12 we can conclude that, despite the fact that the algorithm is not computing the exact Hessian but an approximation, the rate of convergence appears to be almost quadratic (at least for the soft line search) while the algorithm has the ability to converge from a wider range of starting points. The timing values in table 6 seem to be larger

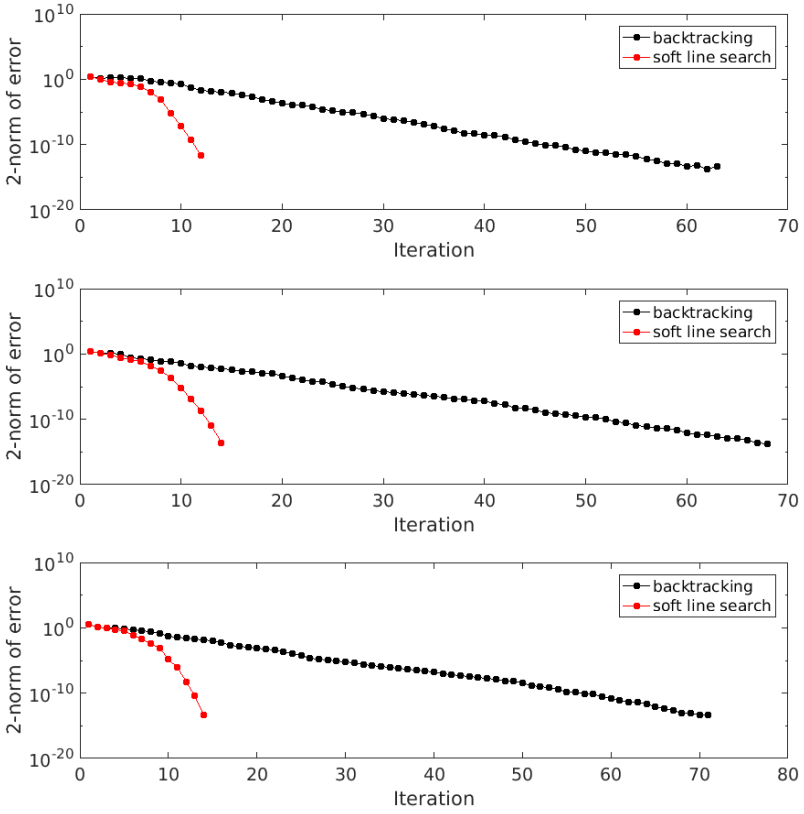

Figure 12: Error plot for Quasi Newton algorithm on the Himmelblau function. Plots are from starting points (1,1), (1,-1) and (-1,0) respectively.

than for Newton algorithm, this is due to the fact that in our code we are just evaluating the Hessian and not computing it in every iteration. However, having an expression for the Hessian matrix may not be a possibility in some problems, where it would have to be calculated every time.

	$x_0 = (1,1)$			$x_0 = (1,-1)$			$x_0 = (-1,0)$		
Method	Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
Backtracking	63	2971	23.1	68	3221	22.3	71	3205	26.6
Soft line search	12	17	4.4	14	21	4.2	14	18	4.9

Table 6: Summary of the Quasi Newton algorithm on the Himmelblau function

2.5 Gauss-Newton algorithm

One important application of the least squares problem is the solution of a nonlinear systems of equations. We need to reformulate our optimization problem so it takes the form of:

$$\min \|r(x)\|_2^2 \tag{13}$$

By visual inspection of Himmelblau function, we see that equation 3 can be written as:

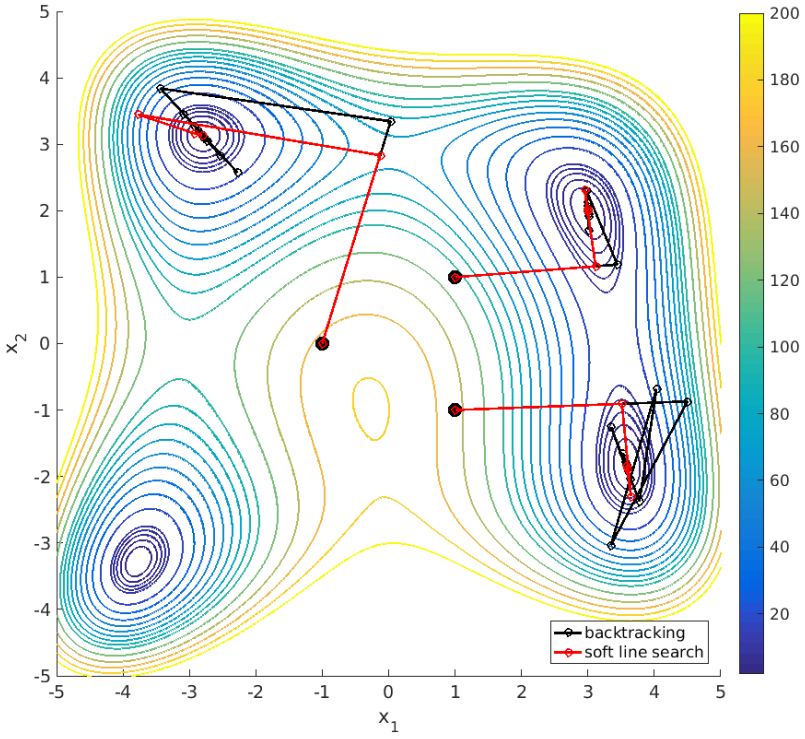

Figure 13: Gauss-Newton algorithm on the Himmelblau function from the starting point (1,1), (1,-1) and (-1,0).

$$f(x) = \|r(x)\|_2^2 \Rightarrow r(x) = r(x_1, x_2) = \begin{pmatrix} x_1^2 + x_2 - 11 \\ x_1 + x_2^2 - 7 \end{pmatrix} \tag{14}$$

It can be easily shown that the gradient can be expressed in terms of the Jacobian matrix as:

$$\nabla f(x) = J(x)' r(x) \tag{15}$$

Deriving the above expression the Hessian takes the form of:

$$\nabla^2 f(x) = J(x)' J(x) + \sum_{i=1}^{m} r_i(x) \nabla^2 r_i(x) \tag{16}$$

Gauss-Newton algorithm can be viewed as a modification of Newton direction, where the Hessian is approximated by neglecting the second term in equation 16. This approximation will be valid whenever the problem is only mildly nonlinear or the residuals at the solution are small.

$$\left(J(x_k)' J(x_k)\right) p_k = -J(x_k)' r(x_k) \Rightarrow p_k = -J(x_k)^{-1} r(x_k) \tag{17}$$

where $J(x_k)$ is the Jacobian evaluated in x_k

$$J(x) = \begin{pmatrix} \frac{dr}{dx_1} & \frac{dr}{dx_2} \end{pmatrix} = \begin{pmatrix} 2x_1 & 1 \\ 1 & 2x_2 \end{pmatrix} \tag{18}$$

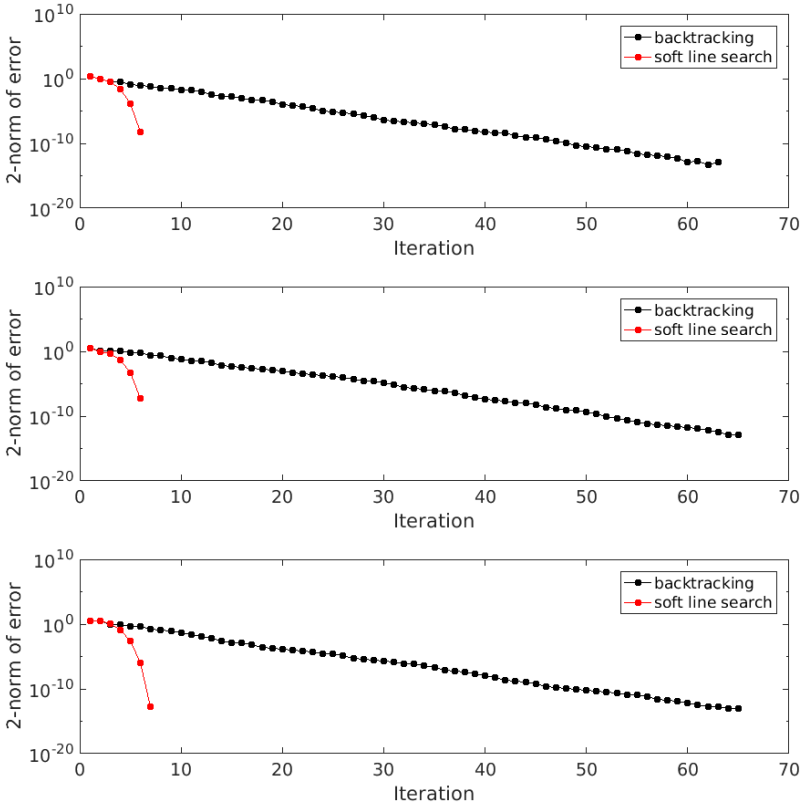

Figure 14: Error plot for Gauss-Newton algorithm on the Himmelblau function. Plots are from starting points (1,1), (1,-1) and (-1,0) respectively.

The method makes the algorithm much more efficient, since once we have computed the gradient we can immediately obtain the approximation to the Hessian. Moreover, as long as the first term in equation 16, $(J(x)'J(x))$, is much larger than the second, the algorithm will preserve Newton's quadratic convergence.

Finally, if $J(x)$ has full rank and the gradient is different from 0, we can assure that the search direction is a descent direction and thus, our method will converge to a minimum.

In figure 13 the convergence sequence can be seen from the three starting points. It is seen that this variation of Newton's algorithm can converge from all the starting points – contrary to Newton's algorithm presented using the Hessian. From figure 14 it is seen, that the quadratic convergence of Newton's algorithm is preserved (at least for the soft line search). The required number of iterations to converge along with the computational time (table 7) is similar to Newton's algorithm and this is because the residuals are close to 0 when near the solution. Consequently, the approximation of the Hessian happens to be acceptable, since the first term of equation 16 is significantly greater than the second term.

	$x_0 = (1,1)$			$x_0 = (1,-1)$			$x_0 = (-1,0)$		
Method	Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
Backtracking	63	2983	20.1	65	2819	19.2	65	3046	19.8
Soft line search	6	10	1.9	6	11	1.9	7	12	2.0

Table 7: Summary of the Gauss-Newton algorithm on the Himmelblau function

2.6 Levenberg-Marquardt

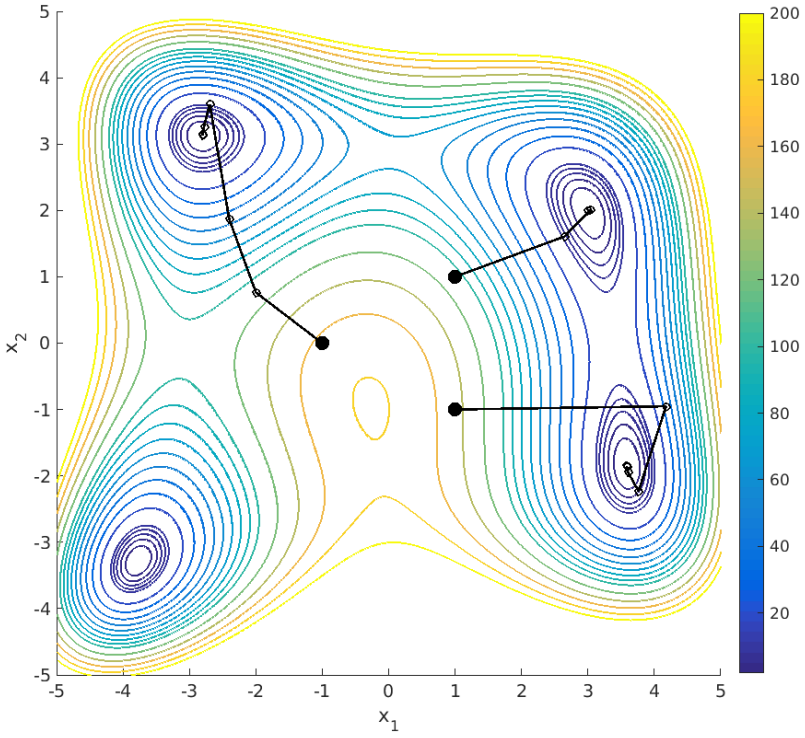

Figure 15: Levenberg-Marquardt algorithm on the Himmelblau function from the starting point (1,1), (1,-1) and (-1,0).

Levenberg-Marquardt method can be treated as one of the many damped newton algorithms.

In order to face the global convergence problem of Newton's method Damped Newton algorithms try to combine the safe and global convergence properties of steepest descent direction, whenever the iteration is far from the solution, and the quadratic convergence rate of Newton's algorithm when the sequence is close to a local minimum.

This is done by modifying the Newton direction as shown in equation 19.

$$p_k = -(\nabla^2 f_k + \mu I)^{-1} \nabla f_k \tag{19}$$

Where the damping parameter (μ), which is always greater than 0, is updated in every iteration.

We see that when $\mu = 0$ the search direction is equal to Newton's direction and as μ gets larger equation 19 looks more similar to steepest descent direction divided by μ. Therefore, ideally, μ will be bigger when the algorithm iterates somewhere far from a local minimum and it will be 0 or close to 0 when being around the solution.

Damped Newton algorithms differ in the way they update μ, specifically Levenberg-Marquardt method makes sure that the sequence moves towards a minimum by checking that the matrix $\nabla^2 f_k + \mu I$ is positive definite before taking the next step. The damped factor is increased when the matrix is not positive definite.

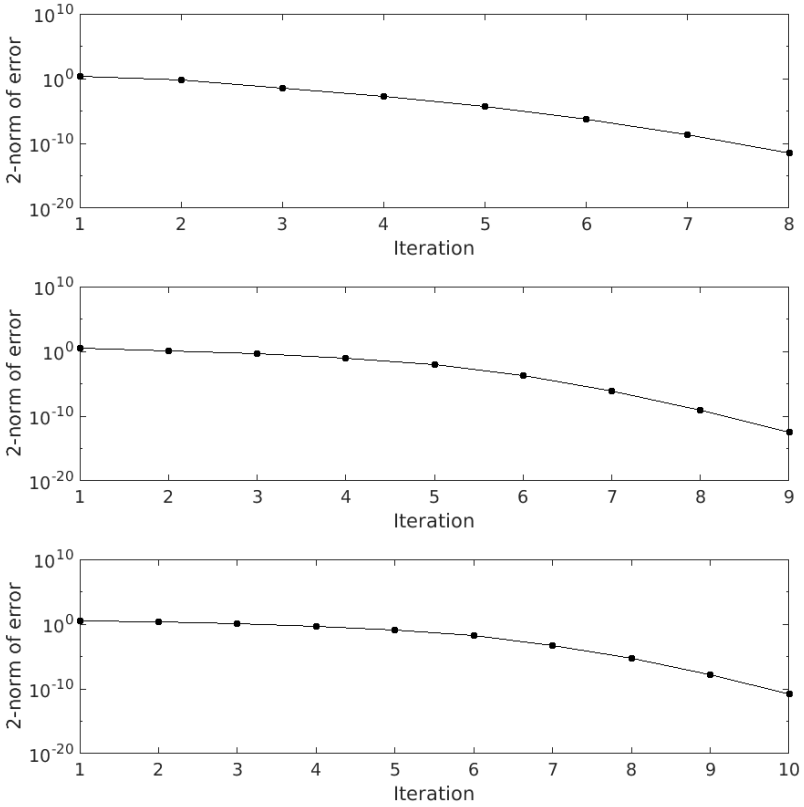

Figure 16: Error plot for Levenberg-Marquardt algorithm on the Himmelblau function. Plots are from starting points (1,1), (1,-1) and (-1,0) respectively.

By updating the damping parameter, Levenberg-Marquardt method tries to influence both search direction and step length, thus the use of a line search algorithm turns to be unnecessary.

Again the optimization problem can be transformed into a least squares problem and then, equation 19 can be written as, where the term $J(x_k)'J(x_k)$ is an approximation to the Hessian:

$$p_k = -(J(x_k)'J(x_k) + \mu I)^{-1} J(x_k) r(x_k) \tag{20}$$

In figure 15 the convergence paths can be seen from the three starting points. The Levenberg-Marquardt seems to converge quadratically for all the starting points (figure 16). Even though it takes a similar number of iterations as some of the line search methods, it is seen (table 8) that the computational time is much lower for the Levenberg-Marquardt algorithm. This is because the algorithm does not need any further method to determine the step length.

$x_0 = (1, 1)$			$x_0 = (1, -1)$			$x_0 = (-1, 0)$		
Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
8	16	0.42	9	18	0.43	10	22	0.55

Table 8: Summary of the Levenberg-Marquardt algorithm on the Himmelblau function

2.7 Comparison of algorithms

As a conclusion for the optimization problem, we have gathered the results of all our algorithms in table 9 along with some outputs obtained from Matlab optimization function fminunc.

We have mentioned that although requiring many iterations, the intuitive steepest descent method is always able to find a solution even when starting from points which are far from local minimums. If we look at the number of iterations and function evaluations of steepest descent we realize that the values are considerably larger than for the rest of the algorithms.

Algorithm	$x_0 = (1, 1)$			$x_0 = (1, -1)$			$x_0 = (-1, 0)$		
	Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
Steepest Descent	46	182	17.1	49	232	16.9	21	92	7.4
Quasi Newton	12	17	4.4	14	21	4.2	14	18	4.9
Gauss-Newton	6	10	1.9	6	11	1.9	7	12	2.0
Levenberg-Marquardt	8	16	0.42	9	18	0.43	10	22	0.55
fminunc	7	8	17.3	8	9	17.7	7	8	16.6
	$x_0 = (-2, 0)$			$x_0 = (1, -1)$			$x_0 = (-4, -2)$		
	Iter	Evals	Time	Iter	Evals	Time	Iter	Evals	Time
Newton	6	7	2.0	10000	0	2455.1	6	8	2.2

Table 9: Summary of the methods discussed. For the line search methods – Steepest descent, Quasi Newton, Gauss-Newton and Newton – only the results from the soft line search are included. The results from fminunc (using Levenberg-Marquardt) from matlab are also included. Notice that all algorithms have the same starting points except Newton's algorithm, since it failed to converge for all of them.

On the other hand, we showed that apart from being computationally demanding Newton method presents a lack of global convergence, a good example of this is the starting point (-1,1) where Newton algorithm is unable to find a solution. However, when iterating around a minimum (starting point -4,-2) the convergence turns out to be quadratic and the number of iterations is low. We said that the iterations could sometimes lead the sequence to a maximum or a saddle point, as it occurs when starting from (-2,0).

Quasi Newton algorithm appeared as an efficient alternative to Newton method, where the Hessian is approximated in every iteration. Obviously, that comes along with a lower convergence rate which can be seen if we compare the number of iterations and function evaluations for both algorithms. Yet, the values are lower than for steepest descent while it potentially employs less resources than Newton method, as it does not compute the Hessian.

Again, we would like to remark that in our algorithms we are using an expression for evaluating the Hessian, which avoids tedious computations. Therefore, the improvement of using Quasi-Newton approximation instead of the exact Hessian matrix cannot be seen by looking at the timing values.

Treating the optimization problem as a least square problem allowed us to write an algorithm that gives an approximation to the Hessian by using the Jacobian matrix. Gauss-Newton algorithm is particularly advisable when the residuals are close to 0, which occurs

when the sequence is near a minimum. It also offers good computation results because there is no need to calculate second derivatives. The low number of iterations and function evaluations for any of the starting points presented in table 9 prove that the algorithm is totally suitable for our minimization problem.

Levenberg-Marquardt combines steepest descent's global convergence properties with Newton's quadratic convergence by adding the damping factor to the Hessian matrix. The factor can be used to control both direction and step length and thus we did not use a line search method in this algorithm. The values show that while keeping the number of iterations and function evaluations low, the algorithm is much faster than the rest as it does not need an inner loop to find the step length in every iteration.

Finally, from the results obtained by running fminunc function (which was set to use a Levenberg-Marquardt method), it can be seen that, although the number of iterations and evaluations can be reduced compared to our implementation, the computational time is much higher.

A Numerical verification of gradient and Hessian

In this section the gradient is evaluated for each of the 9 stationary points to verify that $\nabla f(x) = \binom{0}{0}$. Furthermore, the Hessian is also evaluated in each stationary point to determine whether the point is a local minimum, local maximum or a saddle point. The calculations are conducted in Matlab and can be seen in Appendix C.8

Stationary point #1: $(x_1, x_2) = (3.00, 2.00)$

$$\nabla f(x) = \begin{pmatrix} 0.00 \\ 0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} 74.00 & 20.00 \\ 20.00 & 34.00 \end{pmatrix}$$

Eigenvalues of Hessian: $25.72, 82.28$

The Hessian is positive definite since both eigenvalues are positive. This means that the stationary point $(x_1, x_2) = (3.00, 2.00)$ is a local minimmum

Stationary point #2: $(x_1, x_2) = (-3.78, -3.28)$

$$\nabla f(x) = \begin{pmatrix} 0.00 \\ -0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} 116.27 & -28.25 \\ -28.25 & 88.23 \end{pmatrix}$$

Eigenvalues of Hessian: $70.71, 133.79$

The Hessian is positive definite since both eigenvalues are positive. This means that the stationary point $(x_1, x_2) = (-3.78, -3.28)$ is a local minimmum

Stationary point #3: $(x_1, x_2) = (-2.81, 3.13)$

$$\nabla f(x) = \begin{pmatrix} -0.00 \\ -0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} 64.95 & 1.30 \\ 1.30 & 80.44 \end{pmatrix}$$

Eigenvalues of Hessian: $64.84, 80.55$

The Hessian is positive definite since both eigenvalues are positive. This means that the stationary point $(x_1, x_2) = (-2.81, 3.13)$ is a local minimmum

Stationary point #4: $(x_1, x_2) = (3.58, -1.85)$

$$\nabla f(x) = \begin{pmatrix} 0.00 \\ -0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} 104.79 & 6.95 \\ 6.95 & 29.32 \end{pmatrix}$$

Eigenvalues of Hessian: $28.69, 105.42$

The Hessian is positive definite since both eigenvalues are positive. This means that the stationary point $(x_1, x_2) = (3.58, -1.85)$ is a local minimmum

Stationary point #5: $(x_1, x_2) = (-3.07, -0.08)$

$$\nabla f(x) = \begin{pmatrix} -0.00 \\ 0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} 71.00 & -12.62 \\ -12.62 & -38.21 \end{pmatrix}$$

Eigenvalues of Hessian: $-39.65, 72.44$

The Hessian is indefinite since the eigenvalues have opposite sign. This means that the stationary point $(x_1, x_2) = (-3.07, -0.08)$ is a saddle point

Stationary point #6: $(x_1, x_2) = (-0.13, -1.95)$

$$\nabla f(x) = \begin{pmatrix} -0.00 \\ 0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} -49.62 & -8.33 \\ -8.33 & 19.29 \end{pmatrix}$$

$$\text{Eigenvalues of Hessian: } -50.61, 20.28$$

The Hessian is indefinite since the eigenvalues have opposite sign. This means that the stationary point $(x_1, x_2) = (-0.13, -1.95)$ is a saddle point

Stationary point #7: $(x_1, x_2) = (0.09, 2.88)$

$$\nabla f(x) = \begin{pmatrix} -0.00 \\ -0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} -30.37 & 11.88 \\ 11.88 & 74.17 \end{pmatrix}$$

$$\text{Eigenvalues of Hessian: } -31.71, 75.51$$

The Hessian is indefinite since the eigenvalues have opposite sign. This means that the stationary point $(x_1, x_2) = (0.09, 2.88)$ is a saddle point

Stationary point #8: $(x_1, x_2) = (3.39, 0.07)$

$$\nabla f(x) = \begin{pmatrix} -0.00 \\ -0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} 95.81 & 13.84 \\ 13.84 & -12.39 \end{pmatrix}$$

$$\text{Eigenvalues of Hessian: } -14.14, 97.55$$

The Hessian is indefinite since the eigenvalues have opposite sign. This means that the stationary point $(x_1, x_2) = (3.39, 0.07)$ is a saddle point

Stationary point #9: $(x_1, x_2) = (-0.27, -0.92)$

$$\nabla f(x) = \begin{pmatrix} 0.00 \\ 0.00 \end{pmatrix}, \quad \nabla^2 f(x) = \begin{pmatrix} -44.81 & -4.78 \\ -4.78 & -16.86 \end{pmatrix}$$

$$\text{Eigenvalues of Hessian: } -45.61, -16.07$$

The Hessian is negative definte since both eigenvalues are negative. This means that the stationary point $(x_1, x_2) = (-0.27, -0.92)$ is a local maximum

B Code for Part 1

B.1 OptimalN.m

```
%We first create the x-axis vector (time in hours) and the y-axis vector
%(observations):
t = 1:24; t=t';
y = [91.22 28.04 22.91 26.65 42.96 101.05 202.36 328.02 364.12 299.23 ...
    238.00 227.49 218.03 223.62 238.75 271.26 267.72 251.32 230.04 ...
    206.69 170.77 131.67 143.85 157.57]';

%The A matrices and X* vectors corresponding to each n can be found by
%getting subsets from the A matrix and X* vector for n=23. That is, the
%X* vector corresponding to n=3, for instance, is the first 3 elements of
%the X* vector of n=23.

%Therefore, we can simply calculate the A matrix and the X* vector for
%n=23, and extract subsets from them to find the matrices and vectors for
%lower values of n:

[X23 R23 A23] = NOfit(t, y, 23);


%We cannot do the same with the residual vectors (R) since their values
%differ with each different order of n.

%Therefore, we create matrix R with column vectors storing the residuals
%for each different odd value of n from n=3 to 23.
R = zeros(24,11);
for i=1:11
    n= 2*i + 1;
    R(:,i)=NOfitR(t,y,n);
end
R1 = NOfitR(t,y,1);
R = [R1 R];  %We add residuals of n=1

%We multiply x_star(s) with matrices A to obtain the approximations to the
%observations (y_hat) for each value of n. We store them in Y-hat matrix:
Y_hat = zeros(24,11);
for i=1:11
    j = 2*i +1;
    Y_hat(:, i) = A23(:,1:j) * X23(1:j,:);
end

%%
% Part1_2: TESTING THE SOFTWARE
%
%List the found solution x* for n=3 and plot the least squares fit M(x*,t)
%together with the data points.


scatter(t,y);
hold on;
title('LSQ Model Fit');
xlabel('Time (hrs)');
ylabel('NO Concentration (microg/m3)');
plot(t,Y_hat(:,1), 'k', 'LineWidth', 2);     % N = 3
plot(t,Y_hat(:,2), 'k');                     % N = 5
plot(t,Y_hat(:,3), 'b', 'LineWidth', 2);     % N = 7
```

```
plot(t,Y_hat(:,4), 'r', 'LineWidth', 2);      % N = 9
plot(t,Y_hat(:,5), 'c', 'LineWidth', 2);      % N = 11
plot(t,Y_hat(:,6), 'k');                      % N = 13
plot(t,Y_hat(:,7), 'k');                      % N = 15
plot(t,Y_hat(:,8), 'k');                      % N = 17
plot(t,Y_hat(:,9), 'k');                      % N = 19
plot(t,Y_hat(:,10), 'k');                     % N = 21
plot(t,Y_hat(:,11), 'k');                     % N = 23
hold off;

%X* solution for N = 3: [186.8058;-44.9364;-93.4298]
 X_star_N3 = X23(1:3);

%%
% Part1_3: OPTIMAL ORDER OF FIT
%
%We calculate the z-scores for randomness of signs and the autocorrelation/
%trend treshold ratio:
zscores = zeros(12,1);
for i = 1:12
    zscores(i,1) = runTest(R(:,i));
end

autocorr = zeros(12,1);
for i = 1:12
    autocorr(i,1) = autocorrelationTest(R(:,i));
end

%%Plotting residuals for different values of n with their corresponding
%%z-score (for randomness of signs) and autocorrelation/trend treshold
%%ratio.
subplot(6,2,1);                                     %N=1
scatter(t,R(:,1));
title('N = 1     Z = 4.156     A/t = 4.137');


subplot(6,2,2);                                     %N=3
scatter(t,R(:,2));
title('N = 3     Z = 3.329     A/t = 3.525');

subplot(6,2,3);
scatter(t,R(:,3));                                  %N=5
title('N = 5     Z = 2.488     A/t = 2.507');

subplot(6,2,4);
scatter(t,R(:,4));                                  %N=7
title('N = 7     Z = 0.417     A/t = 0.792');

subplot(6,2,5);
scatter(t,R(:,5));                                  %N=9
title('N = 9     Z = 0.385     A/t = 0.499');

subplot(6,2,6);
scatter(t,R(:,6));                                  %N=11
title('N = 11     Z = 0.035     A/t = 0.348');

subplot(6,2,7);
scatter(t,R(:,7));                                  %N=13
title('N = 13     Z = 2.568     A/t = 2.504');

subplot(6,2,8);
```

```
scatter(t,R(:,8));                                  %N=15
title('N = 15     Z = 2.293     A/t = 2.510');

subplot(6,2,9);
scatter(t,R(:,9));                                  %N=17
title('N = 17     Z = 2.137     A/t = 3.804');

subplot(6,2,10);
scatter(t,R(:,10));                                 %N=19
title('N = 19     Z = 3.757     A/t = 4.608');

subplot(6,2,11);
scatter(t,R(:,11));                                 %N=21
title('N = 21     Z = 3.819     A/t = 4.716');

subplot(6,2,12);
scatter(t,R(:,12));                                 %N=23
title('N = 23     Z = 4.598     A/t = 4.596');

%%
%Part1_3b: SCALED-RESIDUAL NORM
%
%We calculate the 2 norm of the residuals and the scaled residuals for
%different values of n:
Rnorm = zeros(12,1);
for i=1:12
    Rnorm(i,1) = norm(R(:,i));    %Residual Norm as a function of n
end

N = 1:2:23; N = N';
M = 24; MN = M - N;           %MN is the denominator (m - n) of S* equation

SRnorm = zeros(12,1);
for i=1:12
    SRnorm(i,1) = Rnorm(i,1)       /  sqrt(MN(i,1));  %Scaled Residual Norm
end

%We plot Residuals and Scaled Residuals as function of N
SResidualsPlot = figure;
subplot(2,1,1);
scatter(N, Rnorm);
title('Residual norm as function of N');

subplot(2,1,2);
scatter(N, SRnorm);
title('Scaled-Residual norm as function of N');

%%
% Part1_4: ESTIMATING STD DEV OF SOLUTION COEFFICIENTS (x*)
%
%We compute the std deviations for the elements of x* (for x* of n = 11):
cov_x11 = (SRnorm(6,1))^2 * (A23(:,1:11)'*A23(:,1:11))^-1;
sigma_x11 = sqrt(diag(cov_x11));
```

B.2 NOfit.m

```
function [ x_star, r_star, A ] = NOfit( t , y , n )

%NOFIT is a function that computes the least squares fit of the following
```

```
%model: M(x,t) = x1 +x2 sin(wt)+x3 cos(wt)+x4 sin(2wt)+x5 cos(2wt)+···
%given the inputs of t, y and n. Here w = 2pi/24 is the period.

%INPUTS: t is a column vector representing the x-axis (time) of the time
%          series data in hours
%        y is a column vector with the observed measurements of [NOx]
%        n is the order of the fit model desired; that is, n is the number
%          of features desired to be included in the approxmation model.
%

%OUTPUTS: The function outputs:
%                  x_star: a column vector holding the n unknown parameters
%                  r_star: a column vector holding the residuals
%                  A     : matrix A is added to ease computations in
%                          OptimalN.m script. It is the A matrix of the
%                          normal equations to solve the LSQ fit problem.


%We define matrix A with each column representing a basis
%function (eg. f1(t1)=x1, f2(t1)=x2*sin(wt)) and with rows representing
%the different inputs (t1, t2,...,t24) for each basis function.


w = 2*pi/24;  W = (1:length(t))'*w;
N = n-1;
A = zeros(length(t),N);
close all;
scatter(t,y);


for ii=1:2:N
    A(:,ii)=sin(W*(ceil(ii/2)));
end

for ii=2:2:N
    A(:,ii)=cos(W*(ii/2));
end


A=[ones(length(t),1) A];
%We solve x_star by using the normal equation
x_star = linsolve((A'*A), (A'*y));
r_star = y - A*x_star;

hold on
plot(t,A*x_star)
end
```

B.3 autocorrelationTest.m

```
function [ ratio ] = autocorrelationTest( r )
%AUTOCORRELATIONTEST This function computes the values of autocorrelation
%and trend treshold for any input of r residuals. This helps assess whether
%short sequences of consecutive residuals are correlated.
%
%   INPUT: r is the residuals vector
%   OUTPUT: the autocorrelation to trend treshold ratio
%
%NOTE: If the absolute value of the autocorrelation exceeds the trend
```

```
%treshold, a trend is likely to be present in the residuals. Therefore,
%values greater than 1 suggest that a trend is likely to be present in the
%residuals.

m = length(r);
v = zeros(1, m-1);

for i=1:m-1
    v(1,i) = r(i)*r(i+1);
end

auto = abs(sum(v));

trend = (r'*r) / sqrt(m-1);

ratio = auto/trend;
end
```

B.4 runTest.m

```
function [ z ] = runTest( r )
%RUNTEST runTest is a function used to assess the randomness of signs (+
%and -) in residuals.
%   INPUT: A column vector of residuals wished to be analyzed
%   OUTPUT: A Z-score, that if less than 1.96, the residuals' signs can be
%           considered random at a 5% significance level.

m=length(r);
pvector = zeros(1,m); nvector = zeros(1,m);
runs = 1;

for i=1:m
    pvector(i) = r(i) >= 0;
    nvector(i) = r(i) <= 0;
end

npos = sum(pvector); nneg = sum(nvector);

for ii=1:m-1
    if ~(pvector(ii) == pvector(ii+1))
        runs = runs +1;
    end
end

mu_runs = (2 * npos * nneg / m) + 1;
std_runs = sqrt(((mu_runs - 1) * (mu_runs - 2) / (m - 1)));

z = abs((runs - mu_runs)) / std_runs;
end
```

C Code for Part 2

C.1 himmelblau.m

```
function [f,g,H]= himmelblau(X)
%The flags are used in order to avoid unnecessary computations
x1=X(1);
x2=X(2);
f = (x1^2+x2-11)^2 + (x1+x2^2-7)^2;
g = [4*(x1^2+x2-11)*x1+2*(x1+x2.^2-7); 2*(x1^2+x2-11)+4*(x1+x2^2-7)*x2];
H = [12*x1^2+4*x2-42,       4*(x1+x2);
      4*(x1+x2),      4*x1+12*x2^2-26];
```

C.2 steepest_descent.m

```
clear; clc; close all;
path(path,'../../immoptibox');
path(path,'../../exportfig');

x0 = [1,1; 1,-1; -1,0]; %List of starting points
N = size(x0,1); %Number of starting points
tol = 1e-12; %tolerance of solution
maxIter = 10000; %maximum number of iterations before terminating
iter1 = zeros(1,N); iter2 = zeros(1,N); %Vectors for number of iterations
evals1 = zeros(1,N); evals2 = zeros(1,N); %Vectors for number of func evals
rep = 100; %Number of repetitions
T1 = zeros(rep,N); T2 = zeros(rep,N); %Vectors for convergence time

e1 = cell(1,N); e2 = cell(1,N);

a=5; %Backtracking step length

%Contour plot
fig = figure('Position', [100, 0, 1000,1000]);
hold on
himmelblauContours

for i = 1:N
    plot(x0(i,1),x0(i,2),'k.','markersize',40)


    for j = 1:rep
        iter1(i) = 0; evals1(i) = 0;
        iter2(i) = 0; evals2(i) = 0;

        %Backtracking
        T = tic;
        X1 = x0(i,:);
        [F1, g1] = himmelblau(X1);

        while norm(g1,'inf')>tol && iter1(i) < maxIter
            p=-g1;
            [xn, fn, gn,evals] = backtracking(@himmelblau,X1(end,:),F1(end),p,a);
            X1 = [X1; xn]; F1 = [F1; fn];
            g1 = gn;
            iter1(i) = iter1(i) + 1;
            evals1(i) = evals1(i) + evals;
```

```
        end
        T1(j,i) = toc(T);


        %line search optibox
        T = tic;
        X2 = x0(i,:);
        [F2, g2] = himmelblau(X2);

        while norm(g2,'inf')>tol && iter2(i) < maxIter
            p=-g2;
            [xn, fn, gn,info] = linesearch(@himmelblau,X2(end,:)',F2(end),g2,p);
            X2= [X2; xn']; F2 = [F2; fn];
            g2 = gn;
            iter2(i) = iter2(i) + 1;
            evals2(i) = evals2(i) + info(3);
        end
        T2(j,i) = toc (T);
    end

    h1 = plot(X1(:,1),X1(:,2),'k-o','linewidth',2);
    h2 = plot(X2(:,1),X2(:,2),'r-o','linewidth',2);

    e1{i} = sqrt(sum((X1 - X1(end,:)).^2,2));
    e2{i} = sqrt(sum((X2 - X2(end,:)).^2,2));
end
T1 = mean(T1); T2 = mean(T2);
%%
legend([h1,h2],'backtracking','soft line search','location','southeast')

export_fig '../img/steepestDescent.png'

errorplot('../img/steepestDescentError.png',e1,e2,iter1,iter2,N,maxIter);

saveTable('steepestDescent.tex',x0,iter1,iter2,evals1,evals2,T1,T2)
```

C.3 newton.m

```
clear; clc; close all;
path(path,'../../immoptibox');
path(path,'../../exportfig');

x0 = [-2,0; 1,-1; -4,-2]; %List of starting points
N = size(x0,1); %Number of starting points
tol = 1e-12; %tolerance of solution
maxIter = 10000; %maximum number of iterations before terminating
iter1 = zeros(1,N); iter2 = zeros(1,N); %Vectors for number of iterations
evals1 = zeros(1,N); evals2 = zeros(1,N); %Vectors for number of func evals
rep = 100; %Number of repetitions
T1 = zeros(rep,N); T2 = zeros(rep,N); %Vectors for convergence time

e1 = cell(1,N); e2 = cell(1,N);

a=5; %Backtracking step length

%Contour plot
fig = figure('Position', [100, 0, 1000,1000]);
hold on
himmelblauContours
```

```
colorbar
axis image

for i = 1:N
    plot(x0(i,1),x0(i,2),'k.','markersize',40)

    for j = 1:rep
        iter1(i) = 0; evals1(i) = 0;
        iter2(i) = 0; evals2(i) = 0;
        %Backtracking
        T = tic;
        X1 = x0(i,:);
        [F1,g1,H1] = himmelblau(X1);
        while norm(g1,'inf')>tol && iter1(i) < maxIter
            p=-H1\g1;
            [Xn, Fn, gn,evals] = backtracking(@himmelblau,X1(end,:),F1(end),p,a);
            X1= [X1; Xn]; F1 = [F1; Fn];
            g1 = gn;
            [~,~,H1] = himmelblau(X1(end,:));
            iter1(i) = iter1(i) + 1;
            evals1(i) = evals1(i) + evals;
        end
        T1(j,i) = toc(T);

        %line search optibox
        T = tic;
        X2 = x0(i,:);
        [F2,g2,H2] = himmelblau(X2);

        while norm(g2,'inf')>tol && iter2(i) < maxIter
            p=-H2\g2;
            [xn, fn, gn,info] = linesearch(@himmelblau,X2(end,:)',F2(end),g2,p);
            X2= [X2; xn']; F2 = [F2; fn];
            g2 = gn;
            [~,~,H2] = himmelblau(X2(end,:));
            iter2(i) = iter2(i) + 1;
            evals2(i) = evals2(i) + info(3);
        end
        T2(j,i) = toc(T);
    end
    h1 = plot(X1(:,1),X1(:,2),'k-o','linewidth',2);
    h2 = plot(X2(:,1),X2(:,2),'r-o','linewidth',2);

    e1{i} = sqrt(sum((X1 - X1(end,:)).^2,2));
    e2{i} = sqrt(sum((X2 - X2(end,:)).^2,2));
end
T1 = mean(T1); T2 = mean(T2);

legend([h1,h2],'backtracking','soft line search','location','southeast')

export_fig '../img/newton.png'

errorplot('../img/newtonError.png',e1,e2,iter1,iter2,N,maxIter);

saveTable('newton.tex',x0,iter1,iter2,evals1,evals2,T1,T2)
```

C.4 quasi_newton.m

```
clear; clc; close all;
```

```
path(path,'../../immoptibox');
path(path,'../../exportfig');

x0 = [1,1; 1,-1; -1,0]; %List of starting points
N = size(x0,1); %Number of starting points
tol = 1e-12; %tolerance of solution
maxIter = 10000; %maximum number of iterations before terminating
iter1 = zeros(1,N); iter2 = zeros(1,N); %Vectors for number of iterations
evals1 = zeros(1,N); evals2 = zeros(1,N); %Vectors for number of func evals
rep = 100; %Number of repetitions
T1 = zeros(rep,N); T2 = zeros(rep,N); %Vectors for convergence time

e1 = cell(1,N); e2 = cell(1,N);

a=5; %Backtracking step length

%Contour plot
fig = figure('Position', [100, 0, 1000,1000]);
hold on
himmelblauContours

for i = 1:N
    plot(x0(i,1),x0(i,2),'k.','markersize',40)

    for j = 1:rep
        iter1(i) = 0; evals1(i) = 0;
        iter2(i) = 0; evals2(i) = 0;

        %Backtracking
        T = tic;
        X1 = x0(i,:);
        [F1,g1] = himmelblau(X1);
        B = eye(size(x0,2));

        while norm(g1,'inf')>tol && iter1(i) < maxIter
            p=-B\g1;
            [xn, fn, gn, evals] = backtracking(@himmelblau,X1(end,:),F1(end),p,a);
            s = xn - X1(end,:);
            y = gn - g1;
            Bs = B*s';
            B = B - (Bs/dot(s,Bs))*Bs' + (y/dot(y,s))*y';
            X1= [X1; xn]; F1 = [F1; fn];
            g1 = gn;
            iter1(i) = iter1(i) + 1;
            evals1(i) = evals1(i) + evals;
        end
        T1(j,i) = toc(T);

        %line search optibox
        T = tic;
        X2 = x0(i,:);
        [F2,g2] = himmelblau(X2);
        B = eye(size(x0,2));

        while norm(g2,'inf')>tol && iter2(i) < maxIter
            p=-B\g2;
            [xn, fn, gn, info] = linesearch(@himmelblau,X2(end,:),F2(end),g2,p);
            s = xn' - X2(end,:);
            y = gn - g2;
            Bs = B*s';
            B = B - (Bs/dot(s,Bs))*Bs' + (y/dot(y,s))*y';
```

```
            X2 = [X2; xn']; F2 = [F2; fn];
            g2 = gn;
            iter2(i) = iter2(i) + 1;
            evals2(i) = evals2(i) + info(3);
        end
        T2(j,i) = toc(T);
    end
    h1 = plot(X1(:,1),X1(:,2),'k-o','linewidth',2);
    h2 = plot(X2(:,1),X2(:,2),'r-o','linewidth',2);

    e1{i} = sqrt(sum((X1 - X1(end,:)).^2,2));
    e2{i} = sqrt(sum((X2 - X2(end,:)).^2,2));
end
T1 = mean(T1); T2 = mean(T2);

legend([h1,h2],'backtracking','soft line search','location','southeast')

export_fig '../img/quasiNewton.png'

errorplot('../img/quasiNewtonError.png',e1,e2,iter1,iter2,N,maxIter);

saveTable('quasiNewton.tex',x0,iter1,iter2,evals1,evals2,T1,T2)
```

C.5 Gauss_Newton.m

```
clear; clc; close all;
path(path,'../../immoptibox');
path(path,'../../exportfig');

x0 = [1,1; 1,-1; -1,0]; %List of starting points
N = size(x0,1); %Number of starting points
tol = 1e-12; %tolerance of solution
maxIter = 10000; %maximum number of iterations before terminating
iter1 = zeros(1,N); iter2 = zeros(1,N); %Vectors for number of iterations
evals1 = zeros(1,N); evals2 = zeros(1,N); %Vectors for number of func evals
rep = 100; %Number of repetitions
T1 = zeros(rep,N); T2 = zeros(rep,N); %Vectors for convergence time

e1 = cell(1,N); e2 = cell(1,N);

a=5; %Backtracking step length

J = @(x1,x2)[2*x1, 1 ;
    1, 2*x2];
r = @(x1,x2)[x1^2+x2-11 ; x1+x2^2-7];

%Contour plot
fig = figure('Position', [100, 0, 1000,1000]);
hold on
himmelblauContours

for i = 1:N
    plot(x0(i,1),x0(i,2),'k.','markersize',40)

    for j = 1:rep
        iter1(i) = 0; evals1(i) = 0;
        iter2(i) = 0; evals2(i) = 0;
        %Backtracking
        T = tic;
```

```
        X1 = x0(i,:);
        [F1] = himmelblau(X1);
        J1=J(X1(1),X1(2));
        r1 = r(X1(1),X1(2));

        while norm(J1*r1,'inf')>tol && iter1(i) < maxIter
            p=-J1\r1;
            [xn, fn,~,evals] = backtracking(@himmelblau,X1(end,:),F1(end),p,a);
            X1= [X1; xn]; F1 = [F1; fn];
            J1 = J(X1(end,1),X1(end,2));
            r1 = r(X1(end,1),X1(end,2));
            iter1(i) = iter1(i) + 1;
            evals1(i) = evals1(i) + evals;
        end
        T1(j,i) = toc(T);

        %line search optibox
        T = tic;
        X2 = x0(i,:);
        F2 = himmelblau(X2);
        J2 = J(X2(1),X2(2));
        r2 = r(X2(1),X2(2));

        while norm(J2*r2,'inf')>tol && iter2(i) < maxIter
            %        p=(J2'*J2)^-1*J2'*(-r2);
            p=-J2\r2;
            [xn, fn,~,info] = linesearch(@himmelblau,X2(end,:),F2(end),J2*r2,p);
            X2= [X2; xn']; F2 = [F2; fn];
            J2 = J(X2(end,1),X2(end,2));
            r2 = r(X2(end,1),X2(end,2));
            iter2(i) = iter2(i) + 1;
            evals2(i) = evals2(i) + info(3);
        end
        T2(j,i) = toc (T);
    end
    h1 = plot(X1(:,1),X1(:,2),'k-o','linewidth',2);
    h2 = plot(X2(:,1),X2(:,2),'r-o','linewidth',2);

    e1{i} = sqrt(sum((X1 - X1(end,:)).^2,2));
    e2{i} = sqrt(sum((X2 - X2(end,:)).^2,2));
end
T1 = mean(T1); T2 = mean(T2);

legend([h1,h2],'backtracking','soft line search','location','southeast')

export_fig '../img/gaussNewton.png'

errorplot('../img/gaussNewtonError.png',e1,e2,iter1,iter2,N,maxIter);

saveTable('gaussNewton.tex',x0,iter1,iter2,evals1,evals2,T1,T2)
```

C.6 Levenberg_Marquardt.m

```
clear; clc; close all;
path(path,'../../exportfig');

x0 = [1,1; 1,-1; -1,0]; %List of starting points
N = size(x0,1); %Number of starting points
M = size(x0,2);
```

```
tol = 1e-12; %tolerance of solution
maxIter = 10000; %maximum number of iterations before terminating
iter1 = zeros(1,N); %Vectors for number of iterations
evals1 = zeros(1,N); %Vector for number of func evals
rep = 100; %Number of repetitions
T1 = zeros(rep,N); %Vectors for convergence time

e1 = cell(1,N);

a=5; %Backtracking step length

%Contour plot
fig = figure('Position', [100, 0, 1000,1000]);
hold on
himmelblauContours

for i = 1:N
    plot(x0(i,1),x0(i,2),'k.','markersize',40)

    for j = 1:rep
        iter1(i) = 0; evals1(i) = 0;
        T = tic;
        X=x0(i,:);
        mu=0.2;
        [F,g,H]=himmelblau(X);
        H=H+mu*eye(M);

        while norm(g,'inf')>tol && iter1(i)<maxIter
            [L,a]=chol(H,'lower');
            while a~=0 %adjust mu until H positive definite
                mu=2*mu;
                H=H+mu*eye(M);
                [L,a]=chol(H,'lower');
            end
            p=L'\(L\(-g)); %calculate direction H*h=-g
            xn=X(end,:)+p';
            [fn]=himmelblau(xn);
            evals1(i) = evals1(i) + 1;
            q = F(end,:)+p'*g+(1/2).*p'*H*p;
            ro=(F(end,:)-fn)/(F(end,:)-q);
            if F(end,:)-fn>0
                X=[X;xn];
                F=[F;fn];
                iter1(i)=iter1(i)+1;
                mu=mu*max(1/3,1-(2*ro-1)^3);
            else
                mu=mu*2;
            end
            [f,g,H]=himmelblau(X(end,:));
            evals1(i) = evals1(i) + 1;
            H=H+mu*eye(M);
        end
        T1(j,i) = toc(T);
    end
    h1 = plot(X(:,1),X(:,2),'k-o','linewidth',2);

    e1{i} = sqrt(sum((X - X(end,:)).^2,2));
end
T1 = mean(T1);
export_fig '../img/LM.png'
```

```
% Error plot
figure('Position', [100, 0, 1000,1000]);
for i = 1:N
    subplot(3,1,i);
    if iter1(i)<maxIter; semilogy(e1{i},'k.-','markersize',20); end
    xlabel('Iteration','Fontsize',14)
    ylabel('2-norm of error','Fontsize',14)
    set(gca,'fontsize',14);
end
set(gcf, 'Color', 'w');
export_fig('../img/LMError.png')

% Save table
T1 = 1000*T1;
file = fopen('../tables/LM.tex','w');
fprintf(file,'\\begin{tabular}{ccc|ccc|ccc} \\hline \\hline \n');
fprintf(file,'\\multicolumn{3}{c}{$x_0 = (%d,%d)$} & \\multicolumn{3}{c}{$x_0 = (%d,%d)$} & \\m
fprintf(file,'Iter & Evals & Time & Iter & Evals & Time & Iter & Evals & Time \\\\ \\hline \n')
fprintf(file,'%d & %d & %.2g & %d & %d & %.2g & %d & %d & %.2g \\\\ \n',iter1(1),evals1(1),T1(1
fprintf(file,'\\hline \\hline \n');
fprintf(file,'\\end{tabular} \n');
fclose(file);
```

C.7 matlab_optimization.m

```
clear; clc;
x0 = [1,1; 1,-1; -1,0]; %List of starting points
N = size(x0,1); %Number of starting points
rep = 100; %Number of repetitions
opts = optimoptions('fminunc','SpecifyObjectiveGradient',true,'OptimalityTolerance',1e-12);
file = fopen('fminuncOutput.txt','w');

for i = 1:N
    T1 = zeros(rep,1);
    for j = 1:rep
        T = tic;
        [x1,fval1,exitflag1,output1]=fminunc(@himmelblau,x0(i,:),opts);
        T1(j) = toc(T);
    end
    T1 = mean(T1)*1000;
    fprintf(file,'Start: (%d,%d) Converged to: (%.2f,%.2f) Iterations: %d Evals: %d Time: %.1f
end
T1 = mean(T1);
fclose(file);
```

C.8 evalStatPoints.m

```
X = [3, 2; -3.77931, -3.28319; -2.80512, 3.13131;
    3.58443, -1.84813; -3.07303, -0.081353; -0.127961, -1.95371;
    0.0866775, 2.88425; 3.38515, 0.0738519; -0.270845, -0.923039];

file = fopen('../Latex/evalStatPoints.tex','w');

for i = 1:length(X)
    [~,g,H] = himmelblau(X(i,:));
    E = eig(H);
    fprintf(file,['\\paragraph{Stationary point \\#%d:} $(x_1,x_2) = ' ...
```

```
        '(%.2f,%.2f)$\n'],i,X(i,1),X(i,2));
    fprintf(file,'\\begin{gather*}\n');
    fprintf(file,['\\nabla f(x) = \\begin{pmatrix} %.2f \\\\ %.2f ' ...
        '\\end{pmatrix}, \\quad \n'],g(1),g(2));
    fprintf(file,['\\nabla^2 f(x) = \\begin{pmatrix} %.2f & %.2f \\\\ ' ...
        '%.2f & %.2f \\end{pmatrix} \\\\ \n'],H(1,1),H(1,2),H(2,1),H(2,2));
    fprintf(file,'\\text{Eigenvalues of Hessian: } %.2f, %.2f',E(1),E(2));
    fprintf(file,'\\end{gather*}\n');
    if sum(E>0) == 2
        fprintf(file,['The Hessian is positive definite since both ' ...
            'eigenvalues are positive. This means that the stationary ' ...
            'point $(x_1,x_2) = (%.2f,%.2f)$ is a local minimmum'], ...
            X(i,1),X(i,2));
    elseif sum(E>0) == 1
        fprintf(file,['The Hessian is indefinite since the eigenvalues '...
            'have opposite sign. This means that the stationary point '...
            '$(x_1,x_2) = (%.2f,%.2f)$ is a saddle point'],X(i,1),X(i,2));
    elseif sum(E>0) == 0
        fprintf(file,['The Hessian is negative definte since both ' ...
            'eigenvalues are negative. This means that the stationary ' ...
            'point $(x_1,x_2) = (%.2f,%.2f)$ is a local maximum'], ...
            X(i,1),X(i,2));
    end
end

fclose(file);
```

C.9 backtracking.m

```
%Backtracking determines the maximum step length that gives a sufficient
%decrease for a specific search direction.

%Input parameters:
%fun=Handle to the function that needs to be minimized
%x=Current x
%f=Current f(x)
%p=Step vector (search direction)
%a=Initial step length

%Ouput parameters:
%xn: new x. xn=x+a*p
%fn: new f. f(xn)
%gn: new g. g(xn)
%evals: number of function evaluations used

function [xn,fn,gn,evals] = backtracking(fun,x,f,p,a)
p = p/norm(p);
phi = @(a)(fun(x'+a*p)); %computes the value of the function for x+a*p
                         %where a=steplenght and p=direction
stop=false;
evals=0;
iter=0;
maxIter=100;
c=0.9; %Expected decrease coefficient.
t=0.7; %step length reduction coefficient.

% The step length is being reduced in each iteration while the
% Goldstein-Armijo condition it is not satisfied.
while ~stop && iter<maxIter
```

```
    evals = evals + 1;
    if phi(a)> c*f %Goldstein-Armijo condition. Determines the expected decrease.
    a=t*a;
    else stop=true;
    end
    iter=iter+1;
end
xn=x+a*p'; %computes the new x
[fn, gn] = fun(xn);
evals = evals + 1;
```

C.10 errorplot.m

```
function errorplot(fileName,e1,e2,iter1,iter2,N,maxIter)

figure('Position', [100, 0, 1000,1000]);
for i = 1:N
    subplot(3,1,i);
    if iter1(i)<maxIter; semilogy(e1{i},'k.-','markersize',20); end
    hold on
    if iter2(i)<maxIter; semilogy(e2{i},'r.-','markersize',20); end
    xlabel('Iteration','Fontsize',14)
    ylabel('2-norm of error','Fontsize',14)
    legend('backtracking','soft line search')
    set(gca,'fontsize',14);
end
set(gcf, 'Color', 'w');
export_fig(fileName)
```

C.11 himmelblauContours.m

```
function himmelblauContours

Himmelblau = @(x1,x2)(x1.^2+x2-11).^2 + (x1+x2.^2-7).^2;

% Compute the function values
x1 = linspace(-5,5,1000);
x2 = linspace(-5,5,1000);
[X1,X2]=meshgrid(x1,x2);
F = Himmelblau(X1,X2);

% Make contour plot
v = [0:2:10 10:10:100 100:20:200];
contour(X1,X2,F,v,'linewidth',2);

xlabel('x_1','Fontsize',14)
ylabel('x_2','Fontsize',14)
set(gca,'fontsize',14);
set(gcf, 'Color', 'w');
colorbar
axis image


%Xmin = [3, 2; -3.77931, -3.28319; -2.80512, 3.13131; 3.58443, -1.84813];
%Xsaddle = [-3.07303, -0.081353; -0.127961, -1.95371; 0.0866775, 2.88425; 3.38515, 0.0738519];
%Xmax = [-0.270845, -0.923039];
```

```
% Plot stationary points
%plot(Xmin(:,1),Xmin(:,2),'b.','markersize',40)
%plot(Xsaddle(:,1),Xsaddle(:,2),'k.','markersize',40)
%plot(Xmax(:,1), Xmax(:,2),'r.','markersize',40)
%set(gcf, 'Color', 'w');
%export_fig '../img/himmelblauContours.png'
```

C.12 muplot.m

```
function muplot(fileName,sup1,sup2,iter1,iter2,N,maxIter)

figure('Position', [100, 0, 1000,1000]);
for i = 1:N
    subplot(3,1,i);
    if iter1(i)<maxIter; plot(sup1{i},'k.-','markersize',20); end
    hold on
    if iter2(i)<maxIter; plot(sup2{i},'r.-','markersize',20); end
    xlabel('Iteration','Fontsize',14)
    ylabel('\mu','Fontsize',14)
    legend('backtracking','soft line search')
    set(gca,'fontsize',14);
end
set(gcf, 'Color', 'w');
export_fig(fileName)
```

C.13 saveTable.m

```
function saveTable(tblName,x0,iter1,iter2,evals1,evals2,T1,T2)
T1 = 1000*T1; T2 = 1000*T2;

file = fopen(['../tables/' tblName],'w');
fprintf(file,'\\begin{tabular}{l|ccc|ccc|ccc} \\hline \\hline \n');
fprintf(file,'& \\multicolumn{3}{c}{$x_0 = (%d,%d)$} & \\multicolumn{3}{c}{$x_0 = (%d,%d)$} & \
fprintf(file,'Method & Iter & Evals & Time & Iter & Evals & Time & Iter & Evals & Time \\\\ \\h

fprintf(file,'Backtracking & %d & %d & %.1f & %d & %d & %.1f & %d & %d & %.1f \\\\ \n',iter1(1)
fprintf(file,'Soft line search & %d & %d & %.1f & %d & %d & %.1f & %d & %d & %.1f \\\\ \n',iter

fprintf(file,'\\hline \\hline \n');
fprintf(file,'\\end{tabular} \n');
fclose(file);
```