FMAN61 - Optimization Project

Quasi-Newton methods with application to penalty and barrier problems

Ludvig Eriksson Brangstrup, *lu1717er-s*;
Erik Inghammar, *er1284in-s*;
Kasper Nordenstam, *ka0884no-s*;

December 2023

Supervisor:
Stefan Diehl

Acknowledgements

Erik and Kasper implemented the optimization program. Erik has been responsible for the main implementation and Kasper for adapting it to solve the problems in the report. Ludvig worked on the algorithm testing and evaluation of the different methods, parameters and was also responsible for the report.

1 Introduction

This report will describe the theory behind, and an implementation of, Quasi-Newton methods for optimization of functions, with and without constrained domains. More specifically, the Davidon–Fletcher–Powell (DFP) and Broyden–Fletcher–Goldfarb–Shanno (BFGS) methods. DFP and BFGS is especially useful for solving optimization problems where computing the second derivative is especially hard or computationally heavy. The core idea of the methods is for every iteration updating an estimate of the inverse Hessian matrix (A matrix of second order derivatives). While DFP was the first widely used optimization method to approximate the Hessian matrix, BFGS later came and offered improved robustness and better convergence properties. The methods will be tested on functions where the theoretical performance is known. They will then be applied to optimization problems, specifically the Rosenbrock function:

$$100\,(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{Rosenbrock}$$

and this problem, from now on referred to as (pen):

$$\begin{aligned} &\text{minimize} \quad e^{x_1 x_2 x_3 x_4 x_5} \\ &\text{subject to} \quad \begin{cases} x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 = 10 \\ x_2 x_3 = 5 x_4 x_5 \\ x_1^3 + x_3^3 = -1, \end{cases} \end{aligned} \tag{pen}$$

where penalty and barrier methods will enforce the constraints.

2 Theory

In general, an optimization problem can be described as

$$\begin{aligned} &\text{minimize} \quad f(\mathbf{x}) \\ &\text{subject to} \quad \mathbf{x} \in S \end{aligned} \tag{P}$$

Where the *feasible region* S is some restriction of $\mathbb{R}^n$, usually a combination of *inequality* and *equality* constraints.
The a prototype algorithm for solving minimization problems is described on page 39 of [1], and reproduced in Algorithm 1. This algorithm takes in a starting point from which you compute a direction in which to search, minimzizes the objective function $f(\boldsymbol{x})$ from (P) along that line and repeats the search procedure until it finds a satisfactory point.

Algorithm 1: Prototype algorithm

Input : starting point $\mathbf{x}_0 \in \mathbb{R}^n$

Output: approximate local minimizer, or a warning that none could be found

$k \leftarrow 1$

repeat

compute a *search direction* $\mathbf{d}_k$

do *line search*: find $\lambda_k \in \mathbb{R}$ that (approximately) minimizes

$F(\lambda) := f(\mathbf{x}_k + \lambda \mathbf{d}_k),\ \lambda > 0$

$\boldsymbol{x}_{k+1} \leftarrow \mathbf{x}_k + \lambda \mathbf{d}_k$

$k \leftarrow k + 1$

until *termination criteria* are satisfied;

2.1 Termination criteria

There are many ways at which we can stop an iteration. Since this project concerns numerical optimization and tries to be as fast as possible we need to keep in mind that we are looking for a *good-enough*. Some conditions on which to stop are:

- $\|\nabla f(\mathbf{x}_k\|) < \varepsilon_{tol}$ gradient tolerance,
- $\|\mathbf{x}_k - \mathbf{x}_{k-N}\| < \varepsilon_{tol}$ step size tolerance
- $\frac{\|\mathbf{x}_k - \mathbf{x}_{k-N}\|}{C + \|\mathbf{x}_{k-N}\|} < \varepsilon_{tol}$ relative step size tolerance,
- $f(\mathbf{x}_{k-N}) - f(\mathbf{x}_k) < \varepsilon_{tol}$ function decrease tolerance,
- $\frac{f(\mathbf{x}_{k-N}) - f(\mathbf{x}_k)}{C + |f(\mathbf{x}_{k-N})|} < \varepsilon_{tol}$ relative function decrease tolerance,
- `N_iter` $\geq$ `MAX_ITER` number of iterations tolerance

The last one is useful as a secondary condition in all while-loops in order to prevent us from getting stuck indefinitely without progress. We have implemented the first two conditions with a shared tolerance, as well as the last conditions. To avoid numerical problems with the finite difference method, we can derive a minimum tolerance. The difference between two function values must be larger than the precision 10^{-16}. The finite differnce is calculated at points -10^{-8} and 10^{-8}. This gives the limit condition on the tolerance t, where an approximately linear function is assumed,

$$|(-t10^{-8}) - (t10^{-8})| = 2t10^{-8} > 10^{-16} \tag{1}$$

So the tolerance is larger than 10^{-8}. However, problems can arise even if this condition is fulfilled if the search direction is close to orthogonal to the gradient. In this case, the finite difference is increased by a factor of 10 until the derivative is negative.

2.2 Line search

In our prototype algorithm 1 we are performing a *line-search* in some direction. This search is a separate problem from finding a suitable line, which will be discussed in separate chapters . The line-search means

finding the value λ_k that minimizes

$$F(\lambda) := f(\mathbf{x}_k + \lambda \mathbf{d}_k),\ \lambda > 0$$

in some direction $\mathbf{d}_k$. This is no trivial problem. The objective function might be very complex to calculate at certain points, the algorithm for finding a minima might also require operators on the objective function which require costly computations. The usual trade-offs between speed and storage need to be taken into account when designing an optimizer.

2.2.1 Armijo's rule

We have decided to implement an inexact line search method called Armijo's rule as seen in (B.5) which can find the optimal step size λ in a given direction d. The first part of its main loop is the step forward phase:

```
while F(alpha*lambda) < F_0 + epsilon*F_prim_0*alpha*lambda
lambda = alpha * lambda;
N_eval = N_eval + 1;
end
```

Here it checks whether the Armijo's condition is satisfied and if so the algorithm takes a step by increasing λ. The next part of the main loop is backtracking which is done once the Armijo's condition is not satisfied any more. This is done by decreasing the size of λ until Armijo's condition is satisfied once more, and then we have found the optimal λ. The Armijo's condition (2) determines whether λ is acceptable by checking if the decrease in function value is proportional to both λ and the gradient at x. The algorithm can quickly find a *satisfactory* value for the minimizer, whilst only sparingly needing the derivatives of the function. This saves us time and function calls.

$$F(\lambda) \leq F(0) + \epsilon \cdot F'(0) \cdot \lambda \tag{2}$$

2.2.2 Our implementation

A further extension we used was using a *Wolfe's criterion search* as seen in (B.4). This first uses Armijo's rule to find a good starting value for λ and then performs a bisection search to find a better minimizer. The *Wolfe's condition* contains both the Armijo's condition and the curvature condition (3) which ensures that the slope has been reduced sufficiently by λ. If the curvature condition is not satisfied initially, then the first part of the loop is performed:

```
1 if abs(F_prim_lambda) > -sigma * F_prim_0
2     iter = 0;
3     while F_prim_lambda < 0 && iter < MAX_ITER
4         a = lambda;
5         lambda = alpha * lambda;
```

```
        F_prim_lambda = num_gradient(F,lambda);
        N_eval = N_eval +2;
    end
    b = lambda;
    lambda = (a + b)/2;
    F_prim_lambda = num_gradient(F,lambda);
```

which increases λ in order to find a larger interval in which the curvature condition might be satisifed, this is done until either the curvature condition is satisfied or the maximum iterations has been performed. Now that the interval has been determined, the next part of the loop involves a binary search which is performed in order to narrow down the interval until λ satisfies the curvature condition.

$$|F'(\lambda)| \leq -\sigma \cdot F'(0) \tag{3}$$

2.3 Newton's method

The idea of Newton's method is to assume that the function being optimized is well approximated by a a quadratic function. If this is true, it is uniquely defined by the Hessian matrix H. During iterations, the Hessian is computed in each point, before the next point is found as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}_k^{-1}\mathbf{g}_k \tag{4}$$

where $\mathbf{g}_k$ is the gradient. On a quadratic function, this point would be the minimum, so the method should converge in one iteration.

2.4 Quasi-Newton methods

Newton's method unfortunately requires us to compute the Hessian matrix of the problem and invert it, which is computationally intensive. Quasi-Newton methods instead estimate the inverse of the Hessian and update this estimate step-by-step using only the differences of gradients and points, where the updated matrix $\mathbf{D}_{k+1}$ satisfies the *quasi-Newton condition*

$$\mathbf{D}_{k+1}\mathbf{q}_i = \mathbf{p}_i, \quad i = 1, \ldots, k, \tag{QN}$$

Here, D_k is the current approximation of the inverse Hessian matrix. q_i denotes the difference in gradients and p_i is the step taken.

2.4.1 DFP

$$\mathbf{D}_{k+1} := \mathbf{D}_k + \frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} - \frac{\mathbf{D}_k\mathbf{q}_k\mathbf{q}_k^T\mathbf{D}_k}{\mathbf{q}_k^T\mathbf{D}_k\mathbf{q}_k} \tag{DFP}$$

(DFP) ensures that the updated matrix D_{k+1} satisfies the quasi-Newton condition (QN) which ensures that the algorithm converges. The initial matrix D_0 should be positive definite since it ensures that d_k is a strict descent direction so it was chosen as I. Its implementation can be seen in (B.1).

2.4.2 BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is a common optimization algorithm belonging to the Quasi Newton methods family. It seeks to find the minima of a function by iteratively approximating the Hessian matrix. Using the approximation of the inverse Hessian the algorithm can find the direction towards the minima. The iterative process consists of first determining the search direction d_k using the inverse hessian approximation and the numerical gradient at current point x. Next the line search is performed in order to find an optimal step size λ. Next the position is updated based on d_k and λ, and then is the inverse hessian aproxiamtion updated using equation (BFGS)

$$\mathbf{D}_{k+1} := \mathbf{D}_k + \frac{1}{\mathbf{p}_k^T \mathbf{q}_k} \left(\left(1 + \frac{\mathbf{q}_k^T \mathbf{D}_k \mathbf{q}_k}{\mathbf{p}_k^T \mathbf{q}_k} \right) \mathbf{p}_k \mathbf{p}_k^T - \mathbf{D}_k \mathbf{q}_k \mathbf{p}_k^T - \mathbf{p}_k \mathbf{q}_k^T \mathbf{D}_k \right). \tag{BFGS}$$

2.4.3 Comparison between DFP, BFGS

The iterations has the same structure in both DFP and BFGS, the only difference there is how the inverse Hessian approximation is updated, but their behaviours differs. DFP without restart can be efficient but tends to be less robust compared to BFGS, however with restart it can greatly improve the robustness especially in cases where the convergence is slow or if it gets stuck for example in a local minima.

BFGS without restart is robust and efficient in most cases but can have difficulties in complex landscapes, with restart it can have improved convergence in complex landscapes.

2.4.4 Constrained optimization

To solve constrained optimization problems, one may introduce a penalty or barrier function. Penalty functions let the solution be a little bit outside the domain and penalise this, while barrier functions do not let the solution reach the border of the domain. Since the constrained problem treated in this report has equality constraints, meaning that the solution is in the best case on the border of the domain, penalty functions had to be used.
We used a φ starting at 1, and every iteration going up by a factor of 2. The iterative process was terminated when the norm of the difference in the solutions between two iterations was less than the predefined tolerance.

2.5 Testing cases

The implementation will be tested on several cases, where theoretical results are known. These are:

$$f(x_1, ..., x_n) = \mathbf{x}^T A \mathbf{x} \tag{5}$$

where A is a positive definite matrix. To test this out, positive definite matrices where generated by first creating a matrix a, where each entry is drawn from a unit normal distribution. a is then multiplied by its transpose to yield a positive semi definite matrix a^*, as

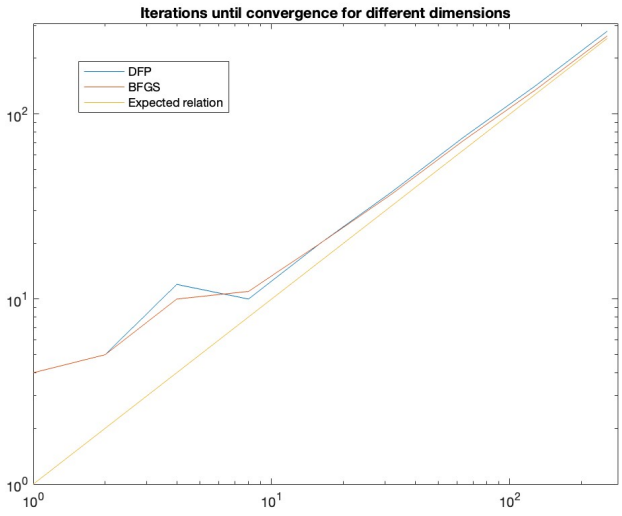

Figure 1: Iterations until convergence

$$\mathbf{x}^T a^* \mathbf{x} = \mathbf{x}^T a^T a \mathbf{x} = (\mathbf{x}a)^T a \mathbf{x} = |a\mathbf{x}|^2 \geq 0. \tag{6}$$

We then get A by adding an identity matrix scaled by 10^{-3} to a^*. Showing that this is positive definite is straight forward.

$$\mathbf{x}^T A \mathbf{x} = \mathbf{x}^T (a^T a + 10^{-3} I)\mathbf{x} = |a\mathbf{x}|^2 + 10^{-6}|\mathbf{x}|^2 > 0, \text{ if } \mathbf{x} \neq \mathbf{0} \tag{7}$$

According to theory, QN methods should find the exact solution within $dim(\mathbf{x})$ iterations using exact line search. Since the line search is not exact, we are using numerical differentiation, and the exact solution does not exist in a numerical world, we can expect slightly more iterations, and this will depend on the tolerance. It is, however, reasonable that with increasing dimension of x we will get some asymptotic property. To evaluate this, 10 random matrices of different size were produced, and for each the solution was found using each of the two algorithms, starting from the **1**-vector, and running to the tolerance 10^{-4}. The number of iterations to convergence is shown in Figure 1. Restart was not used, as this result should only hold without, however, looking at the results, restart would only occur once and not affect the results very much. As can be clearly seen, the number of iterations converges to the dimension of $\mathbf{x}$, in fact, it looks like it might go below, which is likely due to the non-zero tolerance used. BFGS seems to perform better than DFP asymptotically.

$$f(x_1, x_2) = \sqrt{|x_1| + 1} + \sqrt{|x2_2| + 1}, \tag{8}$$

which is a nonconvex function, but one which has only one local extremum, which is the global minima. On this function, the methods should converge to the minimum, as the estimate of the (inverse of the) Hessian should always be positive definite. Both DFP and BFGS converged the optimum on this problem, both

with and without restarts.

3 Implementation

The implementation results in a MATLAB function

`function [x,N_eval,N_iter,normg] = nonlinearmin(f,x0,method,tol,restart,printout)`.

The inputs are f - the function to be *minimized*, x_0 - the initial point, `method` - either `DFP` or `BFGS`, `tol` - the tolerance for terminating the algorithm, and `restart` and `printout` which can be set to either 0 or 1, depending on if restarts should be used and whether printout (of what) is desired. The outputs are x - the found minimizer, `N_eval` - the number of function evaluations, `N_iter` - the number of algorithm iterations, and `normg` - the norm of the gradient at the stopping point.

As stopping criterion we started with gradient stop but to make it work for discontinuous functions we implemented deltax as well.

3.1 Testing

During development we continuously tested different parts of the program in order to be sure that they worked properly. These tests were implemented in `testing.m` **??**, but were subsequently removed once we were satisfied with their state. The final version of `testing.m` can be thought of as our `main` function which runs the (pen)-optimization.

4 Problems

4.1 Rosenbrock's function

Our first task with our completed solver is to minimize the Rosenbrock function for different initial values, using both methods, with and without `restart`. For certain starting values and tolerances this gets us the results seen in Table 1. The optimal point is $0, 0$ with optimal function value 0. The Rosenbrock's function is a popular function used for benchmarking of optimization algorithms. It is a non-convex function that has several challenges for optimization algorithms. It also has a narrow curved valley containing the global minimum, which requires the algorithm to perform delicate steps when in the valley in order to step in the right direction towards the minima. The function is also smooth which allows testing of gradient based algorithms.

Table 1: Results for different initial points x_0.

x_0	`tol`	`restart`	(DFP) Result, N_{iter}, N_{eval}, criterion	(BFGS) Result, N_{iter}, N_{eval}, criterion
$\begin{bmatrix}0.8\\0.5\end{bmatrix}$	10^{-3}	no	$\begin{bmatrix}1\\1\end{bmatrix}$, 10, 201, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$	$\begin{bmatrix}1\\1\end{bmatrix}$, 9, 166, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$
$\begin{bmatrix}1.2\\0.5\end{bmatrix}$	10^{-3}	no	$\begin{bmatrix}1.0005\\1.001\end{bmatrix}$, 9, 164, $\lvert\Delta\mathbf{x}\rvert$	$\begin{bmatrix}1\\1\end{bmatrix}$, 11, 200, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$
$\begin{bmatrix}0.8\\0.5\end{bmatrix}$	10^{-6}	no	$\begin{bmatrix}1\\1\end{bmatrix}$, 12, 241, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$	$\begin{bmatrix}1\\1\end{bmatrix}$, 12, 224, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$
$\begin{bmatrix}1.2\\0.5\end{bmatrix}$	10^{-6}	no	$\begin{bmatrix}1\\1\end{bmatrix}$, 13, 240, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$	$\begin{bmatrix}1\\1\end{bmatrix}$, 12, 220, $\lvert\boldsymbol{\nabla}\mathbf{x}\rvert$
$\begin{bmatrix}0.8\\0.5\end{bmatrix}$	10^{-3}	yes	$\begin{bmatrix}0.98164\\0.96338\end{bmatrix}$, 7, 124, $\lvert\Delta\mathbf{x}\rvert$	$\begin{bmatrix}0.9817\\0.96348\end{bmatrix}$, 7, 124, $\lvert\Delta\mathbf{x}\rvert$
$\begin{bmatrix}1.2\\0.5\end{bmatrix}$	10^{-3}	yes	$\begin{bmatrix}1.0002\\1.0004\end{bmatrix}$, 11, 188, $\lvert\Delta\mathbf{x}\rvert$	$\begin{bmatrix}1.0002\\1.0004\end{bmatrix}$, 11, 188, $\lvert\Delta\mathbf{x}\rvert$
$\begin{bmatrix}0.8\\0.5\end{bmatrix}$	10^{-6}	yes	$\begin{bmatrix}1\\1\end{bmatrix}$, 13, 223, $\lvert\Delta\mathbf{x}\rvert$	$\begin{bmatrix}1\\1\end{bmatrix}$, 13, 223, $\lvert\Delta\mathbf{x}\rvert$
$\begin{bmatrix}1.2\\0.5\end{bmatrix}$	10^{-6}	yes	$\begin{bmatrix}1\\1\end{bmatrix}$, 15, 253, $\lvert\Delta\mathbf{x}\rvert$	$\begin{bmatrix}1\\1\end{bmatrix}$, 15, 253, $\lvert\Delta\mathbf{x}\rvert$

As can be seen, the results using restart are consistently worse than without, either stopping too early or using more iterations. This is likely because the Rosenbrock function has a very steep curve in one direction and a very flat curve in an orthogonal direction. This leads the gradient descent step to go in the steep direction, thus slowing convergence. It is not clear to us how it might cause the early stopping, but it might be a matter of numerical chance. The results for the smaller tolerance are more interpretable. It can be noted that the number of function evaluations was on average the same with and without restart.

4.2 Penalty optimization problem

Our second task was to use our `nonlinearmin`-method to find a minimizer to the problem (pen). We were to start at point

$$\mathbf{x}_0 = \Big[-2, 2, 2, -1, -1\Big]^T,$$

also trying some other starting points, and to find a sequence of appropriate penalty parameters.
This problem caused some issues. These may have arised because there seems to be a point where all 3 constraints are fulfilled. This means that all constraints intersect, and it is likely that the domain is not smooth. When applying the penalty, then, we may get a non-differentiable function, which means that we cannot guarantee fulfilling the Wolfe's conditions. This motivated putting the maximum number of iterations inside the Wolfe algorithm. This could also have been due to numerical issues caused by small derivatives.
Note that the numbers of iterations and function evaluations here are the total amounts for all φ tried.

Table 2: Results for different initial points x_0. The tolerance in all cases was 10^{-6}.

x_0	`restart`	(DFP) Result, $f(x)$, N_{iter}, N_{eval}, last φ	(BFGS) Result, $f(x)$, N_{iter}, N_{eval}, last φ
$\begin{bmatrix}-2\\2\\2\\-1\\-1\end{bmatrix}$	no	$\begin{bmatrix}-1.7171\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.05395, 98, 3730, 8192	$\begin{bmatrix}-1.7172\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.053949, 72, 2478, 4096
$\begin{bmatrix}-2\\2\\2\\-1\\-1\end{bmatrix}$	yes	$\begin{bmatrix}-1.7172\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.05395, 88, 2206, 8192	$\begin{bmatrix}-1.7172\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.053949, 71, 2094, 4096
$\begin{bmatrix}0\\0\\0.0001\\-1\\-1\end{bmatrix}$	no	$\begin{bmatrix}-0.69905\\2.7899\\-0.86996\\-0.69672\\0.69672\end{bmatrix}$, 0.43885, 621, 16982, 32768	$\begin{bmatrix}-1.7172\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.05395, 103, 4437, 8192
$\begin{bmatrix}0\\0\\0.0001\\-1\\-1\end{bmatrix}$	yes	$\begin{bmatrix}-1.7171\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.05395, 129, 3324, 8192	$\begin{bmatrix}-1.7172\\1.8272\\1.5957\\-0.76364\\-0.76364\end{bmatrix}$, 0.05395, 105, 2977, 8192

This problem really showed the differences between different methods. Looking first at the recommended starting vector, all variations converged to what seems to be the optimum, but BFGS did so in fewer iterations and function evaluations than DFP. Introducing restarts improved the speed of both methods,

most dramatically for DFP. With the other starting point, which was chosen because it was a limit case in which some methods failed and some succeeded, DFP without restarts did not find the optimum. Switching to BFGS or/and introducing restarts fixed this. When using restarts, although both methods converged, BFGS still did so in fewer iterations. Since restarts helped in this problem, while they made the solution worse in the first problem, it may be a good idea to try both.

References

[1] Stefan Diehl. *Optimization - a basic course.* Studentlitteratur, 2023.

A Printouts from a minimization

```
Optimizing the function @(x)sqrt(abs(x(1))+1)+sqrt(abs(x(2))+1) from the starting point [10  10]' to the gradient tolerance 1e-06.
Restarts are not used.
Printout has been disabled.
The method applied is DFP.
Local minimum found, change in x less than tolerance
Final point x: [-6.71e-13   -6.71e-13]'.
Gradient norm at this x: 4.7448e-05
```

Figure 2: Example of the printouts during optimization.

B Code

B.1 nonlinearmin.m

Listing 1: The main part of our optimizer style

```
function [x_opt, N_eval, N_iter, normg] = nonlinearmin(f, x0, method, tol,
    restart, printout)
% NONLINEARMIN minimizes a given function using DFP or BFGS quasi-Newton
    algorithms.
%
%   [x, N_eval, N_iter, normg] = nonlinearmin(f, x0, method, tol, restart,
    printout)
%
% INPUTS:
%   f         - Function handle for the objective function to be minimized.
%   x0        - Initial point chosen by the user.
%   method    - String specifying the method: 'dfp' or 'bfgs'.
%   tol       - User-defined tolerance for termination.
%   restart   - Flag indicating whether to use restart (1 for yes, 0 for no).
%   printout  - Flag indicating whether to display intermediate results (1 for
    yes, 0 for no).
```

```
13 %
14 % OUTPUTS:
15 %    x          - The point at which the minimum is estimated.
16 %    N_eval     - Total number of function evaluations.
17 %    N_iter     - Total number of iterations.
18 %    normg      - Norm of the gradient at the output point.
19 %
20 % EXAMPLE:
21 %    To minimize the function stored in the m-file func.m with starting point
       (1, 2, 3, 4),
22 %    you may use the command:
23 %    [x, N_eval, N_iter, normg] = nonlinearmin(@func, [1, 2, 3, 4], 'DFP', 1e
       -6, 1, 1);
24 %
25 % NOTE:
26 %    The function f should be defined as a MATLAB function or anonymous
       function.
27
28 disp("Optimizing the function " + func2str(f) + " from the starting point ["
       ...
29     + num2str(x0') + "]' to the gradient tolerance " + num2str(tol) + ".")
30 if restart
31     disp("Restarts are used.")
32 else
33     disp("Restarts are not used.")
34 end
35 if ~printout
36     disp("Printout has been disabled.")
37 end
38 switch lower(method)
39     case 'dfp'
40         % Equation found on p.73 in Diehl, S. 'Optimization - a
41         % basic course'.
42         updateD = @(D_k, p_k, q_k) D_k + (p_k*(p_k')) / (p_k' * q_k) - ...
43             (D_k*q_k*(q_k')*D_k) / (q_k' * D_k * q_k);
44         disp("The method applied is DFP.")
45     case 'bfgs'
46         % Equation found on p.76 in Diehl, S. 'Optimization - a
47         % basic course'.
48         updateD = @(D_k, p_k, q_k) D_k +                          ...
```

```
49              1/(p_k'*q_k)*(                                          ...
50                  (1 + (q_k'*D_k*q_k)/(p_k' * q_k))*p_k*p_k'          ...
51                  - D_k * q_k * (p_k') - p_k*(q_k')*D_k               ...
52              );
53          disp("The method applied is BFGS.")
54      otherwise
55          error("non-implemented method: %s. only 'dfp' and 'bfgs' are
                implemented", method)
56  end
57  % setup
58  MAX_ITER = 500;
59  freq = length(x0);
60
61  % initialization
62  N_eval=0;
63  D_k_plus = eye(length(x0));
64  x_opt = x0; % current best guess for optimizer.
65  x_old = x0 -1; % just initialization
66  N_iter = 0; % number of iterations
67  grad_k_plus = num_gradient(f, x_opt);
68  N_eval = N_eval +2*numel(x_opt);
69  tic
70
71  if printout
72      lambda_k = 0;
73      N_eval = N_eval +1;
74      print_out(N_iter, x_opt, f(x_opt), norm(grad_k_plus), N_eval, lambda_k,
            toc)
75  end
76
77  while N_iter < MAX_ITER
78      grad_k = grad_k_plus;
79      D_k = D_k_plus;
80
81      % Search direction
82      d_k = - D_k * grad_k;
83
84      % line search
85      [lambda_k, N_eval, fx] = wolfe_linsearch(f, x_opt, d_k, N_eval);
86      %
```

```
87
88      x_old = x_opt;
89      x_opt = x_old + lambda_k*d_k;
90
91      grad_k_plus = num_gradient(f, x_opt);
92      N_eval = N_eval +2*numel(x_opt);
93
94      % p,q
95      p_k = x_opt - x_old;
96      q_k = grad_k_plus - grad_k;
97
98      N_iter = N_iter + 1; % we've iterated once again.
99
100     if printout
101         print_out(N_iter, x_opt, fx, norm(grad_k), N_eval, lambda_k, toc)
102     end
103
104     % glitchy stops
105     if p_k == 0
106         disp("Stopped due to no change in x")
107         break
108     elseif q_k == 0
109         disp("Stopped due to no change in gradient")
110         break
111     elseif p_k' * q_k == 0
112         disp("Stopped due to change in gradient orthogonal to change in x")
113         break
114     end
115
116     % non-glitchy stops
117     if N_iter == MAX_ITER
118         disp("Maximum iterations reached")
119         break
120     elseif norm(grad_k_plus) <= tol
121         disp("Local minimum found, gradient less than tolerance")
122         break
123     elseif norm(x_opt-x_old) <= tol
124         disp("Local minimum found, change in x less than tolerance")
125         break
126     end
```

```
127
128     D_k_plus = updateD(D_k, p_k, q_k);
129
130     if restart && mod(N_iter, freq) == 0
131         D_k_plus = eye(length(x0));
132     end
133
134     % another sort of glitchy stop
135     if sum(D_k_plus == Inf, 'all') > 0
136         disp("Stopped due to discontinuity at minimum")
137         break
138     end
139 end
140
141 normg = norm(grad_k_plus);
142 disp("Final point x: [" + num2str(x_opt') + "]'.")
143 disp("Gradient norm at this x: " + string(normg))
144 disp(" ")
145 end
146
147 function N_eval = evals_add(N_eval, k)
148     N_eval = N_eval +k;
149 end
```

B.2 print_out.m

Listing 2: Code that properly formats our printouts style

```
1  function print_out(N_itr, x_itr, fx, n_grad, ls_fun_evals, lambda, time)
2  % PRINT_ITR Print the current iteration level in the form specified in the
3  % manual.
4  %
5  %   print_itr(N_itr, x_itr, fx, n_grad, ls_fun_evals, lambda)
6  %
7  % INPUTS:
8  %   N_itr       : the current iteration level.
9  %   x_itr       : the current x vector.
10 %   fx          : the value of the function at the point x.
11 %   n_grad      : the norm of the gradient of the function f at point x.
12 %   ls_fun_eval : ?
13 %
14
```

```
if 0 == N_itr
    disp("iteration     x                f(x)          norm(grad)    ls fun evals
        lambda        runtime")
end

disp(string(N_itr) + blanks(13 - strlength(string(N_itr))) ...
    + string(x_itr(1)) + blanks(14 - strlength(string(x_itr(1)))) ...
    + string(fx) + blanks(13 - strlength(string(fx))) ...
    + string(n_grad) + blanks(13 - strlength(string(n_grad))) ...
    + string(ls_fun_evals) + blanks(15 - strlength(string(ls_fun_evals))) ...
    + string(lambda) + blanks(14 - strlength(string(lambda))) ...
    + string(time))
for i = 2:length(x_itr)
    disp(blanks(13) + string(x_itr(i)))
end
```

B.3 num_gradient.m

```
function gradient = num_gradient(func, x, varargin)
%NUM_DIFF Calculate the gradient of a function
%   Returns the gradient of a function calculated using the central
%   difference formula for each direction of the vector.
%
%   NOTE: x must be strictly Rn, if x is a matrix this calculator will not
%   work properly.
%
%   nabla_f = num_gradient(f, x0, ['h', H_VALUE])
%
%   Inputs:
%       f           the function to be differentiated.              (Rn->R)
%       x0          the point at which to evaluate the gradient     (Rn)
%       varargin    Optional, Can be used to set the default step of the
%                       central difference method. Default h = 1e-6.
%
%   Outputs:
%       gradient    the gradient of f at the point x.               (Rn)

% TODO: Add logic to adjust h based on the function values of f. If f->0, h
% should become even smaller.

% TODO: Make this function take in a matrix and perform column-by-column
% gradients.

```

```
h = 1e-8;
n = numel(x);
if ~isempty(varargin)
    for i=1:2:numel(varargin)
        if strcmp(varargin{i},'h')
            h = varargin{i+1};
        end
    end
end

% Initialize gradient vector
gradient = zeros(size(x));

% Calculate partial derivatives using finite differences
for i = 1:n
    x_plus_h = x;
    x_minus_h = x;
    x_plus_h(i) = x_plus_h(i) + h;
    x_minus_h(i) = x_minus_h(i) - h;

    % Calculate the partial derivative with respect to the i-th variable
    partial_derivative = (func(x_plus_h) - func(x_minus_h)) / h * 0.5;

    % Update the gradient vector
    gradient(i) = partial_derivative;
end

end
```

B.4 wolfe_linsearch.m

Listing 3: Method performing a linear search satisfying the Wolfe's condition style

```
function [lambda, N_eval, F_0] = wolfe_linsearch(func, x, d, N_eval, varargin)
%WOLF_LINSEARCH Perform a line search to satisfy the Wolfe condition
%   The Wolfe conditions creates an interval of acceptable points to be
%   used when performing an inexact line search. They are:
%       F(lambda) <= F(0) + epsilon*F'(0)*lambda;
%       abs(F'(lambda))<= -sigma*F'(0);
%   Where:
%       F(l) = f(x + l*d), f:Rn->R, x, d: Rn, l: R.
%       0 < epsilon <= sigma < 1
%
%   This algorithm uses Armijo's method to estimate an initial lambda0 and
```

```
12 %   then further tries to find a value that satisfies the Wolfe conditions.
13 %
14 %   lambda = wolfe_linsearch(func, x, d, ...
15 %       ('lambda', LAMBDA_GUESS), ('epsilon',EPSILON_VALUE'), ...
16 %       ('sigma', SIGMA_VALUE), ('alpha', ALPHA_VALUE)
17 %   )
18 %
19 %   Inputs:
20 %       func         objective function
21 %       x            point from which to line search.
22 %       d            direction in which to search.
23 %       'lambda',l   Optional, can be used to guess an initial
24 %                        lambda value.
25 %       'epsilon',e  Optional, used to specify the tolerance of the search.
26 %                        Default is 0.1. 0 < epsilon <= sigma < 1.
27 %       'sigma',s    Optional, used to specify sigma, the tolerance. Default
28 %                        is 0.2.
29 % 0 < epsilon <= sigma < 1.
30 %       'alpha',a    Optional, specify alpha, the update factor. alpha > 1.
31 %                        Default is 2.
32
33 % TODO: Update default values once the method works.
34 epsilon = 0.1;
35 sigma = 0.2;
36 alpha = 5;
37 MAX_ITER = 100;
38
39 % Unpack the optional values.
40 if ~isempty(varargin)
41     for i = 1:2:numel(varargin)
42         switch varargin{i}
43             case 'alpha'
44                 if (varargin{i+1} <= 1)
45                     error('alpha must be larger than 1');
46                 end
47                 alpha = varargin{i+1};
48             case 'epsilon'
49                 epsilon = varargin{i+1};
50             case 'lambda'
51                 lambda = varargin{i+1};
```

```
52              case 'sigma'
53                  sigma = varargin{i+1};
54          end
55      end
56      clear variable value;
57  end
58
59  % Main method begins here.
60  % Method taken from page 54 of Diehl, S. 'Optimization - a basic course'
61
62  % Compute a new lambda from Armijos method.
63  F = @(l) func(x + l*d);
64
65  [N_eval, F_0, lambda, F_prim_0] = armijo(func,x,d,N_eval,varargin{:});
66
67  F_prim_lambda = num_gradient(F,lambda);
68  N_eval = N_eval +2;
69  a = 0;
70  if abs(F_prim_lambda) > -sigma * F_prim_0
71      iter = 0;
72      while F_prim_lambda < 0 && iter < MAX_ITER
73          a = lambda;
74          lambda = alpha * lambda;
75          F_prim_lambda = num_gradient(F,lambda);
76          N_eval = N_eval +2;
77          if abs(F_prim_lambda) <= -sigma * F_prim_0
78              break;
79          end
80
81          iter = iter +1;
82          if iter == MAX_ITER
83              disp("Wolfe stopped on maximum number of iterations")
84          end
85      end
86      b = lambda;
87      lambda = (a + b)/2;
88      F_prim_lambda = num_gradient(F,lambda);
89      N_eval = N_eval +2;
90      iter = 0;
91      while abs(F_prim_lambda) > - sigma * F_prim_0 && iter < MAX_ITER
```

```
92          if F_prim_lambda < 0
93              a  = lambda;
94          else
95              b = lambda;
96          end
97          lambda = (a+b)/2;
98          F_prim_lambda = num_gradient(F,lambda);
99          N_eval = N_eval +2;
100
101         iter = iter +1;
102         if iter == MAX_ITER
103             disp("Wolfe stopped on maximum number of iterations")
104         end
105     end
106 end
107 end
```

B.5 armijo.m

Listing 4: Method used for performing an Armijo's algorithm line search style

```
1  function [N_eval, F_0, varargout] = armijo(f, x, d, N_eval, varargin)
2  %ARMIJO Performs a line search using Armijo's algorithm
3  %
4  %   Performs an inexact line search using Armijo's algorithm along the
5  %   direction d. TODO: explain more in-depth
6  %
7  %   lambda = armijo(f, x, d, ...
8  %           [(lambda, VALUE_LAMBDA),('alpha', VALUE_ALPHA), ('epsilon',
       VALUE_EPSILON)])
9  %
10 %   Inputs:
11 %       f(x)        A function of x for which we are to minimize along a
12 %                       line d.                                  (Rn->R)
13 %       x           The point at which we are currently standing and from
14 %                       which we would like to line search in a direction.
15 %                                                                (Rn)
16 %       d           The direction along which we line search     (Rn)
17 %       'lambda'    Optional, an initial guess for the lambda value.
18 %                       Default is 1e-2.                         (R)
19 %       'alpha'     Optional, specify the value of the updated lambda
20 %                       factor. Default is 2.                    (R)
```

```
21 %       'epsilon'   Optional, specify the relative step size. Default is 0.2.
22 %                                                                     (R)
23 %   Output:
24 %       lambda      The value of lambda that minimizes f along line d.
25 %                                                                     (R)
26
27 % Initialize alpha, epsilon, lambda
28 alpha = 2;
29 epsilon = 0.2;
30 lambda = 1e-2;
31
32 % Check for input arguments
33 if ~isempty(varargin)
34     for i = 1:2:numel(varargin)
35         switch varargin{i}
36             case 'alpha'
37                 alpha = varargin{i+1};
38             case 'epsilon'
39                 epsilon = varargin{i+1};
40             case 'lambda'
41                 lambda = varargin{i+1};
42             otherwise
43                 error(['invalid input:', variable])
44         end
45     end
46     clear variable value;
47 end
48
49 F = @(l) f(x + l*d);
50
51 % save these to avoid unnecessary function calls.
52 F_0 = F(0);
53 N_eval = N_eval +1;
54 F_prim_0 = num_gradient(F,0);
55
56 h = 1e-8;
57 while F_prim_0 > 0
58     h = h *10;
59     F_prim_0 = num_gradient(F,0,'h',h);
60
```

```
61      if h > 1e-3
62          x = -0.1:0.001:0.1;
63          y = zeros(size(x));
64          for i = 1:length(x)
65              y(i) = F(x(i));
66          end
67          figure
68          plot(x, y)
69          title(num_gradient(F, 0))
70
71          keyboard
72
73          error("Bad search direction, D probably close to singular. Try using
                restarts.")
74      end
75  end
76
77  % Algorithm 3, page 53 in Diehl, S. 'Optimization - A basic course'
78  N_eval = N_eval +1;  % this one refers to the first check below
79  while F(alpha*lambda) < F_0 + epsilon*F_prim_0*alpha*lambda
80      % step forwards
81      lambda = alpha * lambda;
82
83      N_eval = N_eval +1;  % this one refes to the next iteration
84  end
85  N_eval = N_eval +1;  % this one refers to the first check below
86  while F(lambda) > F_0 + epsilon*F_prim_0*lambda
87      % backtrack
88      lambda = lambda/alpha;
89
90      N_eval = N_eval +1;  % this one refes to the next iteration
91  end
92
93  switch nargout
94      case 3
95          varargout{1} = lambda;
96      case 4
97          varargout{1} = lambda;
98          varargout{2} = F_prim_0; % this is used in wolfe_linsearch
99  end
```

B.6 impose˙conditions.m

Listing 5: Code used for imposing conditions for optimization style

```
function fwcons = impose_conditions(fwocons, phi, constype, eqcons, ineqcons)
if strcmp(constype, "penalty")
    consf = @(x) phi*x^2;
elseif strcmp(constype, "barrier")
    consf = @(x) (x<0)*phi*(-1/x) + (x>=0)*realmax;
    if ~isempty(ineqcons)
        error("Inequality constraints cannot be combined with barrier
            functions.")
    end
else
    error("Only the options 'penalty' and 'barrier' are available.")
end

cons = @(x) 0;
neqcons = length(eqcons);
for i = 1:neqcons
    coni = eqcons{i};
    cons = @(x) cons(x) + consf(coni(x));
end
nineqcons = length(ineqcons);
for i = 1:nineqcons
    coni = ineqcons{i};
    cons = @(x) cons(x) + consf(coni(max(0, x)));
end

fwcons = @(x) fwocons(x) + cons(x);
end
```

B.7 rosenbrock.m

```
function y=rosenbrock(x)
% Rosenbrock's function
y=100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

B.8 testing.m

Listing 6: Code testing various parts of our optimizer style

```
clear
```

```
clc

%% A selection of possible inputs
%fquad = @(x) sum((x-1).^2);
fposdef = @(x, A) x'*(A*A' + 10^-3*eye(size(A)))*x;
fconv = @(x) abs(x(1)) + abs(x(2));
fnconv = @(x) sqrt(abs(x(1))+1) + sqrt(abs(x(2))+1);

% Rosenbrock function
fros = @(x) rosenbrock(x);  % dim(x) = 2

% (pen)
fpen = @(x) exp(prod(x));  % dim(x) = 5
con1 = @(x) sum(x.^2) - 10;
con2 = @(x) x(2)*x(3) - 5*x(4)*x(5);
con3 = @(x) x(1)^3 + x(3)^3 +1;
eqcons = {con1, con2, con3};
phi = 1;
pen = imposecons(fpen, phi, 'penalty', eqcons, {});
% evaluates pretty fast, 0.003 seconds

dfp = 'dfp';
bfgs = 'bfgs';

%% Set inputs

tol = 10^-6;
restart = 0;  % 1, 0
method = dfp;
printout = 0;

f = fnconv;
x0 = [10  10]';

% f = pen;
% x0 = [-2, 2, 2, -1, -1]';  % givet av Stefan
%x0 = [0, 0, 0.0001, -1, -1]';  % bara bfgs n r r tt punkt! b da med
    restart

% iters = zeros(2, 9);
```

```
41 % dims = 2.^[0 1 2 3 4 5 6 7 8];
42 % for i = 1:9
43 %     dim = dims(i)
44 %     x0 = 1*ones(dim, 1);
45 %     A = randn(length(x0));
46 %     f = @(x) fposdef(x, A);
47 %     [~, ~, iters(1, i), ~] = nonlinearmin(f, x0, dfp, tol, restart, printout
       );
48 %     [~, ~, iters(2, i), ~] = nonlinearmin(f, x0, bfgs, tol, restart,
       printout);
49 % end
50 % figure
51 % loglog(dims, iters)
52 % hold on
53 % loglog([1 2^8], [1 2^8])
54 % title("Iterations until convergence for different dimensions")
55 % legend("DFP", "BFGS", "Expected relation")
56
57 % f = fros;
58 %% x0 = [0.8 0.5]';
59 % x0 = [1.2 0.5]';
60
61 %% Run
62 nonlinearmin(f, x0, method, tol, restart, printout);
63 %[x_opt, N_evaltot, N_itertot, lastphi] = penwrapper(f, x0, method, tol,
       restart, printout, eqcons, {})
```