Politecnico di Torino

Variants of Newton Method

Numerical Optimization for Large Scale Problems and Stochastic Optimization

Giovanbattista Tarantino 338137

Massimiliano Carli 337728

Samuele Vanini 318684

Contents

1 Abstract

In this report, we aim to develop two variants of Newton's method for solving unconstrained optimization problems, specifically addressing cases where the Hessian matrix computed at any iteration is not Symmetric Positive Definite (SPD). Namely, the Truncated Newton Method (*TNM*) and the Modified Newton Method (*MNM*). The analysis will focus on evaluating the convergence speed, computational efficiency, and robustness of both methods, analyzing their behavior on some reference functions.

2 Newton Method and its variants

2.1 Newton method

Newton's method is a well-known iterative algorithm for solving unconstrained optimization problems. It relies on a quadratic approximation of the objective function around the current iterate x_k, with the search direction determined by solving:

$$\nabla^2 f(x^{(k)}) p_k = -\nabla f(x^{(k)}) \tag{1}$$

where f is the objective function and $x^{(k)}$ denotes the current iterate. Solving for p_k, the new iterate is then computed as $x^{(k+1)} = x^{(k)} + p_k$.

Newton Method is characterized by fast local quadratic convergence, but it requires strong assumptions on the objective function. One of the assumptions relies on the method used to solve 2.1, for instance if Conjugate Gradient Method (CG) is employed, then the Hessian matrix $\nabla^2 f(x^{(k)})$ has to be SPD. If not, we can not guarantee that Newton Method will converge.

To address these limitations, two variants have been developed:

- The **Modified Newton Method**, *MNM*, which aims to modify entries of the Hessian matrix to ensure it is SPD;
- The **Truncated Newton Method**, *TNM*, which exploits the iterative characteristic of the CG method by truncating it at a certain iteration, guaranteeing that the solution p_k will be a descent direction.

2.2 Modified Netwon Method

The Modified Newton Method variant consists in correcting the Hessian matrix, at a given iteration k, by some matrix B_k, such that B_k is both sufficiently positive definite (meaning its smallest eigenvalue is sufficiently away from zero), and sufficiently similar to the original Hessian matrix (which can be measured by means of the Frobenius norm of the difference between the Hessian $H_k = \nabla^2(f(x^{(k)}))$, and its corrected version B_k).

Such correction can be achieved by employing different techniques, in this report we will analyze two of them, that we called (i) Minimal eigenvalue correction and (ii) Diagonalization correction.

Minimal eigenvalue correction consists in defining a matrix E_k such that

$$E_k = \tau_k I$$

where τ_k is given by

$$\tau_k = \max\left(0, \delta - \lambda_{\min}(H_k)\right)$$

and δ is an arbitrary small positive constant.

The corrected Hessian matrix is then obtained as

$$B_k = H_k + E_k$$

This approach can yield a resulting SPD matrix thanks to the property that $\lambda(A + \tau I) = \lambda(A) + \tau$, where $\lambda(A)$ denotes the eigenvalues of matrix A, and $\lambda(A + \tau I)$ denotes the eigenvalues of matrix $A + \tau I$. Therefore, we can compute τ, adjusting the value of δ, to rule the corrected matrix $B_k = H_k + E_k$ desired.

Diagonalization correction proceeds in two steps. First, compute the full diagonalization of H_k, such that $H_k = X\Lambda X^{-1}$, where Λ is a diagonal matrix containing the eigenvalues of H_k on its main diagonal, and X is an orthogonal matrix whose columns are the corresponding eigenvectors.

Second, construct a modified diagonal matrix $\tilde{\Lambda}$, which is identical to Λ, except that all diagonal entries in Λ below a certain threshold δ are replaced by δ itself. The corrected

matrix B_k is then reconstructed as

$$B_k = X\tilde{\Lambda}X^{-1} = X\tilde{\Lambda}X^T$$

where we applied the property for which if a matrix X is orthogonal, then $X^{-1} = X^T$.

It is evident how the latter approach presented is computationally more expensive compared to the former, due to the fact that all eigenvalues of H_k must be computed, while in Minimal eigenvalue we can exploit some technique able to avoid the computation of all eigenvalues of H_k, focusing specifically on the smallest one. Since Diagonalization correction becomes increasingly more expensive as the dimension of H_k increases, we will present results for this approach only for problems of size $n \sim 10^3$.

2.3 Truncated Newton Method

The TNM operates on the system 2.1, by *truncating* whichever algorithm it is used for solving the linear system. If the solver is not able to find an optimal solution, TNM will use as a descent direction the last approximation computed by the solver.

In this analysis, we implemented the Conjugate Gradient method for solving the mentioned system. *CG* is initialized with a starting value equals to $-\nabla f(x_k)$, which by definition is a descent direction. In the case of a failure, that will be noted as a *truncation*, *CG* will return its last computed value.

3 Analysis Description

We will evaluate the performance of both variants on a collection of functions. We will first test on the *2-Dimensional Rosenbrock* function, to be able to report the solution both with table and with plots on relevant information.

Then, we will implement three functions taken from [1]: *Problem 82*, *Extended Rosenbrock* and *Extended Powell badly scaled*.

The analysis will cover three different sizes, $n = \{10^3, 10^4, 10^5\}$, and 11 total starting points: the suggested point x_0, from the function definition in [1], and 10 randomly generated points sampled in the hypercube $[x_0 - 1, x_0 + 1] \times \ldots \times [x_0 - 1, x_0 + 1] \subset \mathbb{R}^n$.

	Problem 82	Extended Rosenbrock	Extended Powell
Exact Hessian	0.0045 s	0.0036 s	0.0071 s
Approximation	0.0112 s	0.0254 s	0.0184 s
Specific Approximation	0.0116 s	0.0263 s	0.0180 s

Table 1: Average execution time for computing the Hessian, based on different functions and methods. Test size: 1000 points, $n = 10^5$, $h = 10^{-12}$

3.1 Hessian Approximation

For each function, both an exact and an approximated Hessian computations were implemented. The exact Hessian was determined analytically by deriving the second-order partial derivatives of the function. For the approximation, we implemented the Sparse Jacobian approximation described in [2], using a custom approach for each function, by exploiting the patterns of the result matrix. We also added the option to use a specific approximation, based on the point x in which the Hessian is being computed.

We can compare those implementations by their execution time. We conducted a test in which we considered 1000 randomly generated points of size $n = 10^5$ and a level of approximation $h = 10^{-12}$. The average execution time for each implementation was then computed for each function.

The results are summarized in Table 1. All of the functions present a tridiagonal symmetric Hessian, enabling a fast calculation by only requiring computation for the main diagonal and the first upper diagonal, which is then mirrored in the first lower diagonal. The two types of approximation gave similar average times, but since their implementations require the gradient of the respective function to be evaluated multiple times (two for Extended Rosenbrock and Extended Powell, three for Problem 82), both still performed worse than the Exact Hessian.

Furthermore, implementing approximated Hessians of any kind when MNM was employed, resulted in no convergence in any of the functions under analysis. We justify this behavior for the effect of having an inexact hessian definition paired with a further modification of the hessian entries. This combination results in a final matrix that can be considered too inconsistent with the original Hessian, impairing the method's capacity to find descent directions.

3.2 Parameters tuning

For the analysis we used the following hyper-parameters:

- $\rho = 0.5$ as the step-length decrease factor during backtracking;
- $c_1 = 10^{-4}$ as the Armijo parameter during backtracking;
- $\epsilon = 10^{-6}$ as the gradient tolerance for convergence for Truncated Newton Method;
- $\epsilon = 10^{-4}$ as the gradient tolerance for convergence for Modified Newton Method;

We then set 5000 as the maximum number of iterations the two methods can perform, and 50 as the maximum number of backtracking steps allowed at each iteration.

The analysis will be structured in the following way:

- *Plain execution*: Both algorithms will be executed with the default setup [1];
- *Preconditioning*: a preconditioning matrix will be calculated using *incomplete Cholesky factorization* or *Incomplete LU factorization* before applying the CG method for solving 2.1;
- *Approximation*: the Sparse Jacobian approximation of the gradient will be applied when calculating the Hessian matrix, with a level of approximation $h = 10^{-12}$ and both with and without specific information about the point x.

Note that in all of the examples, we only reported the solution for *Plain execution* and just the suggested starting point, together with a random one. In any case tables with comprehensive data are available in Appendix A.

4 2-dimensional Rosenbrock

The Rosenbrock function is defined as

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

[1] For MNM, the preconditioning was always necessary for Problem 82, to allow the method to converge

We will optimize this functions with the following starting points:

$$x_0 = (1.2, 1.2), x_1 = (-1.2, 1)$$

The final results can be seen in Table 1, and in Figure 1 we show the surface plot of the function, with the intermediate steps calculated by the methods, and a bar chart showing the backtracking iterations performed.

We observe that for these specific initial guesses, the Hessian matrix remains a positive definite matrix throughout all iterations. As a result, Hessian matrix correction or CG method truncation were never required, resulting in both methods behaving like the pure Newton's Method.

	$f(x_k)$	$\|g(x_k)\|$	*Iteration*	*Execution time*
x_0	1.088e-25	1.436e-11	8	0.0653s
x_1	3.744e-21	4.473e-10	21	0.0392s

Table 1: Results of execution of Newton Method on Rosenbrock 2-dimensional function.

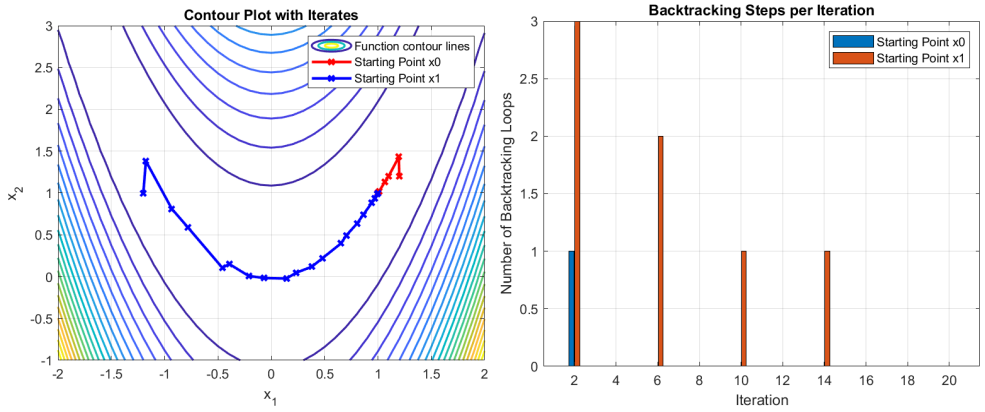

Figure 1: On the left, the contour plot of the two-dimensional Rosenbrock function, showing the convergence of Newton method by both starting points. On the right, the bar plot of the backtracking steps applied in Newton Method for two-dimensional Rosenbrock function.

5 Complete Analysis

5.1 Problem 82

The *Problem 82* function is defined as:

$$F(x) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x)$$

where:

$$f_k(x) = \begin{cases} x_k & : \ k = 1 \\ cos(x_{k-1}) + x_k - 1 & : \ 1 < k <= 0 \end{cases}$$

The suggested starting point is $x_0 = 0.5 \cdot (1, \ldots, 1) \in \mathbb{R}^n$. The Hessian of $F(x)$ is a symmetric tri-diagonal matrix.

Using the suggested starting point, the Hessian matrix was non-PD only one time during the execution, specifically at first iteration. While in the case of randomly chosen starting point, correction and truncation were applied a significantly higher number of times, as shown in Figure 2 and Figure 3.

We noticed that MNM struggled to converge if preconditioning was not employed during CG method, especially for problems of scale $n \sim 10^5$. We therefore always adopted preconditioning with MNM in this problem.

MNM with Minimal eigenvalue correction failed to converge with randomly chosen points when the correction parameter δ was set to values below approximately 10^0. For such lower values of δ, the corrections applied to the Hessian matrix were insufficient to ensure that the final corrected matrix B_k was PD. Therefore, we adopted a higher threshold for δ, despite the possibility for these corrections to result in significantly large values of $\|H_k - B_k\|_F$. However, these adjustments did not prevent the method to convergence, as shown in Table 1.

In the case of MNM with Diagonalization correction we obtained convergence with lower values of δ, namely in the order of 10^{-8}, with a significantly higher execution time for randomly chosen starting points, a comparison of Diagonalization corrections' magnitudes with $\delta = 1$ and $\delta = 1e - 8$ can be seen in Figure 3.

TNM performed well on *Problem 82*, encountering a non-PD matrix during the first iterations of the algorithm, as shown in the Figure 4.

	x_{start}	$f(x_k)$			$\|g(x_k)\|$			*Iteration*			*Execution time (seconds)*			*Truncations*
n	-	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	-
MNM	x_0	3.84e-15	N/A	N/A	8.76e-08	N/A	N/A	5	N/A	N/A	0.2013	N/A	N/A	-
	x_1	3.23e-16	2.11e-14	1.78e-11	2.54e-08	2.05e-07	5.97e-06	11	11	13	0.0534	0.3447	3.5604	-
TNM	x_0	3.84e-15	3.88e-14	3.89e-13	8.76e-08	2.79e-07	8.82e-07	5	5	5	0.0064	0.0083	0.0633	1, 1, 1
	x_1	1.54e-18	7.76e-20	5.08e-15	1.75e-09	3.94e-10	1.01e-07	9	9	10	0.0039	0.0137	0.1338	4, 5, 5

Table 1: Results of execution of MNM and TNM on Problem 82. TNM results are provided with the number of truncation performed.

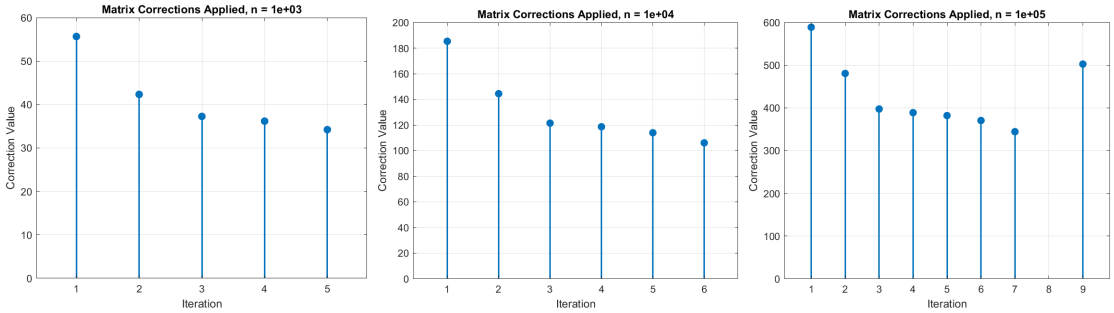

Figure 2: Leaf plot of the corrections applied during the execution of MNM on Problem 82 function, with Minimal eigenvalue correction and $\delta = 1$ on random starting point x_1 for different problem sizes. On the x-axis the iteration at which the correction was applied, on the y-axis the magnitude of that correction, measured as $\|H_k - B_k\|_F$.

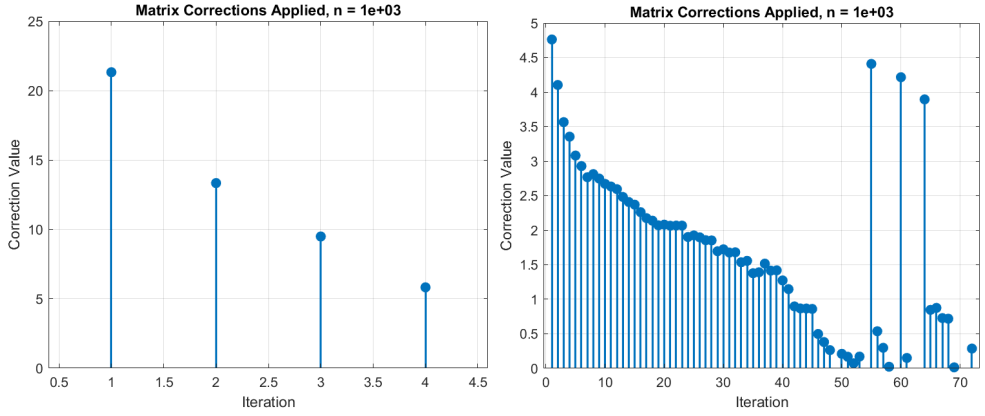

Figure 3: Leaf plot of the corrections applied during the execution of MNM on Problem 82 function with Diagonalization correction on random starting point x_1 for problem of size $n \sim 10^3$. On the x-axis the iteration at which the correction was applied, on the y-axis the magnitude of that correction, measured as $\|H_k - B_k\|_F$. On the left the corrections applied with $\delta = 1$, on the right the corrections applied for $\delta = 1e - 8$.

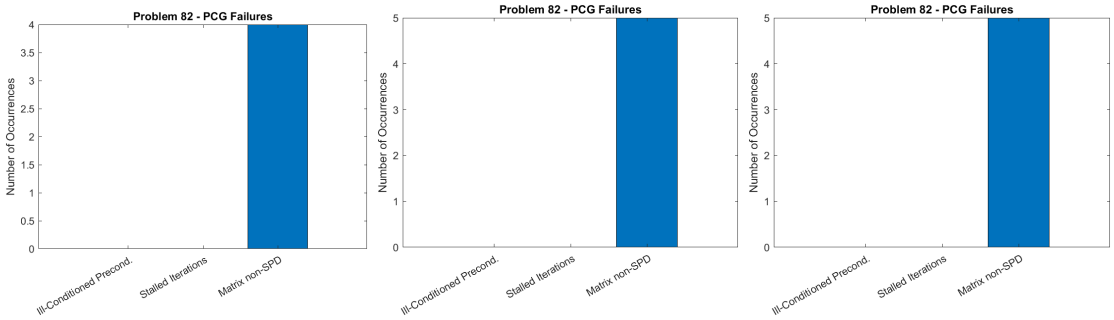

Figure 4: Bar chart of the reason of the truncation applied during the TNM algorithm. From left to right, $(x_1;\ n = 10^3)$, $(x_1;\ n = 10^4)$, $(x_1;\ n = 10^5)$

5.2 Extended Rosenbrock

The *Extended Rosenbrock* function is defined as:

$$F(x) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x)$$

where:

$$f_k(x) = \begin{cases} 10(x_k^2 - x_{k+1}) & : \ k \text{ odd} \\ x_{k-1} - 1 & : \ k \text{ even} \end{cases}$$

The suggested starting point is $x_0 = (-1.2, 1, -1.2, 1, \ldots, -1.2, 1)^T \in \mathbb{R}^n$. The Hessian of $F(x)$ is a symmetric tri-diagonal matrix.

MNM with $\delta = 1e-8$ failed to converge with randomly chosen points, both for Minimal Eigenvalue correction, both for Diagonalization correction. These results can be seen in Table 2.

TNM succesfully converges for both points. We further notice the high sensitivity of the Newton-like methods to the starting positions. While the suggested point x_0 leads to a rapid convergence, having a random point can slow the actual computation. For the case of x_1 in the $n = 10^5$ dimension, it required 510 iterations, 23 truncations and over 25 seconds to reach the optimal solution.

	x_{start}	$f(x_k)$			$\|g(x_k)\|$			*Iteration*			*Execution time (seconds)*			*Truncations*
n	-	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	-
MNM	x_0	9.36e-19	9.36e-18	9.35e-17	5e-09	1.58e-08	5e-08	21	21	21	0.0444	0.0908	0.4818	-
	x_1	N/A			N/A			N/A			N/A			-
TNM	x_0	9.36e-19	9.36e-18	9.36e-17	5e-09	1.58e-08	4.59e-08	21	21	21	0.0175	0.0320	0.3060	0, 0, 0
	x_1	2.28e-20	4.56e-25	3.48e-22	3.66e-09	4.12e-12	5.75e-10	48	89	510	0.0271	0.3675	25.61	7, 15, 23

Table 2: Results of execution of MNM and TNM to Extend Rosenbrock function. TNM results are provided with the number of truncation performed.

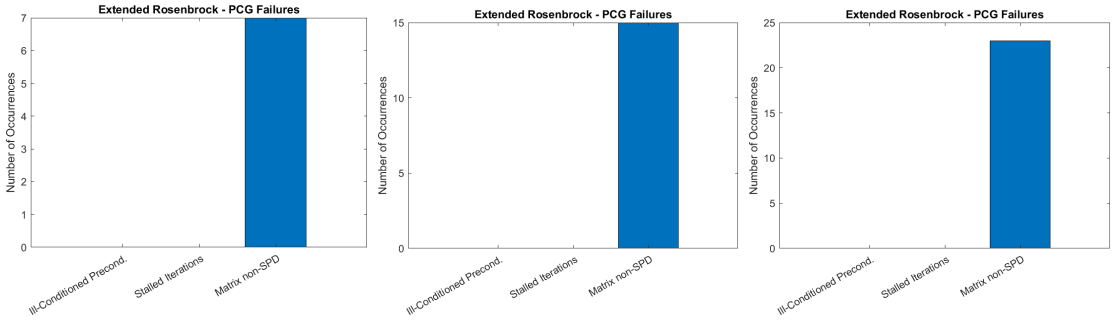

Figure 5: Bar chart of the reason of the truncation applied during the TNM algorithm. From left to right, $(x_1;\ n = 10^3)$, $(x_1;\ n = 10^4)$, $(x_1;\ n = 10^5)$

5.3 Extended Powell badly scaled

The *Extended Powell badly scaled* function is defined as:

$$F(x) = \frac{1}{2}\sum_{k=1}^{n} f_k^2(x)$$

where:

$$f_k(x) = \begin{cases} 1000\, x_k\, x_{k+1} - 1 & : \ k \text{ odd} \\ e^{-x_{k-1}} + e^{-x_k} - 1.0001 & : \ k \text{ even} \end{cases}$$

The suggested starting point for this function is $x_0 = (0, 1, 0, 1, \ldots, 0, 1) \in \mathbb{R}^n$. The Hessian of $F(x)$ is a symmetric tri-diagonal matrix.

As the name suggests, this function is intentionally designed to challenge numerical optimization solvers. Introducing numerical cancellation, ill-conditioned gradients and Hessians. Indeed, such behavior can be observed in the results shown in Table 3. MNM was not able to converge when a random starting point was used as initial guesses, neither with Minima Eigenvalue correction nor with Diagonalization correction. For the suggested starting point x_0 we set $\delta = 10^{-8}$.

TNM only performed some truncations on the CG solver at the start of the algorithm, all of them because the Hessian was non-PD. After that, the Hessian was SPD, making CG converge every time. Nonetheless, TNM failed, due to stagnation.

	x_{start}	$f(x_k)$			$\|g(x_k)\|$			*Iteration*			*Execution time (seconds)*			*Truncations*
n	-	1e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5	-
MNM	x_0	0.00104	0.00253	0.00829	9.86e-05	7.95e-05	8.85e-05	26	29	46	0.067187	0.141547	1.159652	-
	x_1	N/A			N/A			N/A			N/A			-
TNM	x_0	3.88e-07	2.08e-06	1.67e-05	1.35e-07	2.85e-07	8.91e-07	108	114	116	0.055	0.37	2.33	7, 5, 8
	x_1	6.79e+07	3.58e+08	1.2e+10	2.53e+09	3.4e+09	3.62e+10	264	164	171	0.418	2.45	16.5	10, 10, 8

Table 3: Results of execution of MNM and TNM to Extend Powell badly scaled function. For TNM the number of truncation of the *CG* are shown.

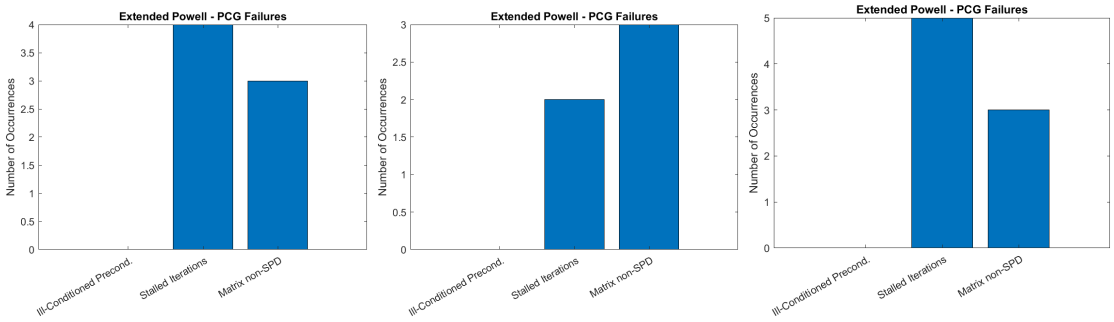

Figure 6: Bar chart of the reason of the truncation applied during the TNM algorithm. From left to right, $(x_0;\ n = 10^3)$, $(x_0;\ n = 10^4)$, $(x_0;\ n = 10^5)$

6 Conclusion

In this report, we analyzed the Truncated Newton Method (TNM) and Modified Newton Method (MNM) for large-scale optimization with non-SPD Hessians. TNM demonstrated superior efficiency, achiving the stopping criteria with fewer iterations and overall less time. MNM, while effective for specific use cases, requires a more careful tuning, often leading to higher computational cost, and poorer results. TNM outperformed MNM in scalability, where MNM frequently failed to converge.

We can conclude that TNM is more versatile and efficient. Future work could explore further Hessian correction techniques, like *modified LDL factorization* [2], or focus on improving the TNM with algorithms that improve the starting guess of the method.

Appendix

Comprehensive tables with results and code developed

A List of all outputs

A.1 Truncated Newton Method

Problem 82

Plain setup: 33 Success, 0 Failures

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	3.84e-15	3.88e-14	3.89e-13	8.76e-08	2.79e-07	8.82e-07	5	5	5	0.0058	0.00857	0.0634	1	1	1
x_1	1.54e-18	7.76e-20	5.08e-15	1.75e-09	3.94e-10	1.01e-07	9	9	10	0.006	0.0159	0.141	4	5	5
x_2	7.77e-15	8.78e-21	3.75e-18	1.25e-07	1.33e-10	2.74e-09	8	9	11	0.00323	0.015	0.149	3	4	6
x_3	6.65e-22	1.05e-16	2.58e-22	3.65e-11	1.45e-08	2.27e-11	9	9	11	0.00499	0.0163	0.166	3	4	5
x_4	2.6e-24	2.3e-24	3.37e-16	2.28e-12	2.14e-12	2.59e-08	9	10	11	0.00489	0.0166	0.162	4	4	5
x_5	4.27e-14	2.45e-20	1.89e-22	2.92e-07	2.21e-10	1.94e-11	8	9	11	0.00216	0.0162	0.189	4	5	6
x_6	1.34e-24	4.71e-16	1.55e-13	1.64e-12	3.07e-08	5.57e-07	9	9	9	0.00286	0.0205	0.132	4	5	5
x_7	3.71e-19	9.52e-14	8.65e-21	8.61e-10	4.36e-07	1.32e-10	8	9	11	0.00263	0.0174	0.141	4	4	6
x_8	2.7e-14	4.46e-14	6.23e-16	2.33e-07	2.99e-07	3.53e-08	8	9	9	0.00271	0.0132	0.124	3	5	5
x_9	2.1e-18	3.93e-13	1.13e-23	2.05e-09	8.86e-07	4.75e-12	8	9	11	0.00281	0.0193	0.169	3	4	5
x_{10}	1.02e-13	3.01e-15	5.49e-24	4.51e-07	7.76e-08	3.31e-12	8	8	10	0.00545	0.0409	0.147	4	4	5

Table 1: TNM results on Problem 82.

Preconditioning: 33 Success, 0 Failures

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	3.84e-15	3.88e-14	3.89e-13	8.76e-08	2.79e-07	8.82e-07	5	5	5	0.00948	0.0139	0.103	1	1	1
x_1	9.03e-15	7.29e-14	6.97e-23	1.34e-07	3.82e-07	1.18e-11	10	16	20	0.0304	0.0988	1.94	3	11	14
x_2	4e-13	1.27e-15	2.93e-19	8.95e-07	5.05e-08	7.66e-10	13	16	20	0.0187	0.235	1.6	8	12	15
x_3	8.36e-22	1.42e-14	6.43e-22	4.09e-11	1.68e-07	3.59e-11	12	16	21	0.0181	0.129	1.92	6	12	15
x_4	1.06e-13	8.04e-23	2.79e-18	4.61e-07	1.27e-11	2.36e-09	14	18	18	0.0333	0.19	1.56	8	13	12
x_5	3.98e-19	4.88e-17	3.85e-13	8.92e-10	9.87e-09	8.78e-07	11	16	20	0.0217	0.186	2.23	5	10	16
x_6	2.7e-17	1.5e-14	2.44e-16	7.35e-09	1.73e-07	2.21e-08	13	17	19	0.034	0.194	2.19	6	13	14
x_7	2.23e-19	7.88e-14	4.78e-16	6.68e-10	3.97e-07	3.09e-08	15	22	23	0.0211	0.223	2.19	10	18	17
x_8	4.2e-22	1.61e-20	1.85e-15	2.9e-11	1.79e-10	6.09e-08	14	18	20	0.0287	0.141	1.74	6	12	14
x_9	1.44e-18	4.67e-14	6.25e-21	1.7e-09	3.05e-07	1.12e-10	13	17	21	0.0241	0.182	2.14	9	12	14
x_{10}	3.47e-24	2.9e-23	1.57e-15	2.63e-12	7.61e-12	5.6e-08	16	17	21	0.0297	0.229	3.91	9	12	17

Table 2: TNM results on Problem 82, with preconditioning applied.

Approximation: 33 Success, 0 Failures

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	3.1e-22	3.13e-21	3.13e-20	2.49e-11	7.91e-11	2.5e-10	6	6	6	0.0217	0.0148	0.0908	6	6	6
x_1	3.17e-19	3.94e-24	4.63e-21	7.96e-10	2.81e-12	9.62e-11	9	10	11	0.00723	0.0203	0.165	9	10	11
x_2	1.66e-14	2e-24	4.92e-19	1.82e-07	2e-12	9.92e-10	8	10	11	0.00345	0.0194	0.169	8	10	11
x_3	2.23e-16	6.41e-22	2.77e-20	2.11e-08	3.58e-11	2.36e-10	9	10	11	0.00349	0.0197	0.169	9	10	11
x_4	8.23e-19	3.81e-21	9.77e-26	1.28e-09	8.73e-11	4.42e-13	9	11	11	0.00526	0.0227	0.17	9	11	11
x_5	1.03e-25	4.09e-25	1.18e-21	4.53e-13	9.04e-13	4.85e-11	10	10	11	0.00461	0.0188	0.202	10	10	11
x_6	5.8e-18	4.51e-19	3.99e-24	3.41e-09	9.5e-10	2.83e-12	9	10	11	0.0107	0.0318	0.232	9	10	11
x_7	3.71e-21	1.11e-13	6.83e-17	8.61e-11	4.71e-07	1.17e-08	9	9	11	0.00303	0.0194	0.172	9	9	11
x_8	7.44e-25	9.64e-21	1.03e-13	1.22e-12	1.39e-10	4.53e-07	9	10	10	0.00337	0.0223	0.148	9	10	10
x_9	4.16e-13	5.1e-24	1.89e-24	9.13e-07	3.19e-12	1.94e-12	8	10	11	0.0037	0.0201	0.16	8	10	11
x_{10}	3.56e-24	9.2e-16	1.15e-13	2.67e-12	4.29e-08	4.8e-07	10	9	10	0.00405	0.0182	0.148	10	9	10

Table 3: TNM results on Problem 82, using the Sparse Jacobian approximation ($h = 10^{-12}$).

Specific apporximation: 33 Success, 0 Failures

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	3.1e-22	3.13e-21	3.13e-20	2.49e-11	7.91e-11	2.5e-10	6	6	6	0.00756	0.0125	0.0926	6	6	6
x_1	4.57e-24	6.07e-22	9.97e-16	3.02e-12	3.48e-11	4.47e-08	11	11	15	0.0134	0.0416	0.638	7	7	7
x_2	7.49e-17	3.99e-21	4.61e-17	1.22e-08	8.93e-11	9.6e-09	10	12	17	0.00601	0.0562	0.738	7	6	8
x_3	9.87e-20	5.17e-14	5.4e-20	4.44e-10	3.22e-07	3.29e-10	11	11	13	0.0066	0.0347	0.476	9	9	9
x_4	1.72e-17	1.79e-16	7.9e-17	5.87e-09	1.89e-08	1.26e-08	9	11	14	0.00492	0.0435	0.577	8	8	8
x_5	5.18e-20	8.07e-17	1.13e-15	3.22e-10	1.27e-08	4.75e-08	10	10	12	0.00615	0.0281	0.322	7	8	9
x_6	4.75e-21	2.62e-25	3.35e-21	9.75e-11	7.24e-13	8.19e-11	10	14	12	0.00507	0.0638	0.427	7	7	7
x_7	1.88e-22	3.18e-19	1.69e-18	1.94e-11	7.98e-10	1.84e-09	10	13	12	0.00705	0.0515	0.355	7	8	8
x_8	5.63e-23	3.83e-23	1.25e-13	1.06e-11	8.76e-12	5e-07	11	12	11	0.00586	0.0383	0.456	9	9	5
x_9	9.43e-22	6.66e-16	1e-19	4.34e-11	3.65e-08	4.47e-10	10	15	14	0.00554	0.0704	0.486	8	7	8
x_{10}	1.14e-13	5.66e-24	7.19e-20	4.77e-07	3.36e-12	3.79e-10	9	11	12	0.00549	0.0439	0.363	8	7	8

Table 4: TNM results on Problem 82, using the specific Sparse Jacobian approximation ($h = 10^{-12}$).

Extended Rosenbrock

Plain setup: 27 Success, 6 Failures. In all the failure cases, TNM could not satified the Armijo condition, even if the Armijo parameters (ρ and c_1) were tuned.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	9.36e-19	9.36e-18	9.36e-17	5e-09	1.58e-08	4.95e-08	21	21	21	0.00913	0.0314	0.304	0	0	0
x_1	2.28e-20	4.65e-25	3.48e-22	3.66e-09	4.12e-12	5.75e-10	48	89	510	0.0242	0.36	27.5	7	15	23
x_2	1.06e-16	7.37e-16	6.59e+03	2.7e-07	3.31e-07	652	110	159	261	0.0598	0.897	14.7	13	13	18
x_3	1.27e-23	1.55e-20	317	3.37e-11	3.03e-09	6.6	66	133	405	0.0464	0.69	23.4	9	14	19
x_4	2.52e-20	3.37e-18	1.31e+05	4.42e-09	5.34e-08	9.65e+03	88	143	101	0.048	0.798	5.79	15	18	21
x_5	4.64e-21	9.02e-20	1.16e+03	2.16e-09	7.57e-09	318	61	223	284	0.0303	1.03	15.3	12	25	15
x_6	3.87e-20	2.49e-21	1.17e-24	4.65e-09	6.45e-11	1.47e-11	58	108	282	0.0307	0.493	15.4	9	17	11
x_7	1.03e-15	1.28e-16	1.44e-17	7.67e-07	2.79e-07	1.17e-07	46	174	259	0.0247	0.786	13.1	6	16	17
x_8	6.27e-23	3.53e-16	479	1.46e-10	5.55e-07	26	88	106	323	0.0466	0.546	17.8	10	12	18
x_9	4.18e-15	4.89e-21	1.36e+03	5.94e-07	3.91e-10	253	54	113	226	0.0287	0.542	12.7	10	19	12
x_{10}	1.46e-26	1.62e-21	4.1e-20	1.44e-12	9.93e-10	5.44e-10	118	145	239	0.0682	0.651	12.1	14	16	19

Table 5: TNM results on Extended Rosenbrock.

Preconditioning: 3 Success, 30 Failures. Using a preconditioner caused the Hessian matrix to become more unstable.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	9.36e-19	9.36e-18	9.36e-17	5e-09	1.58e-08	5e-08	21	21	21	0.0154	0.0397	0.386	0	0	0
x_1	7.44e+04	4.29e+05	9.2e+04	1.33e+04	5.29e+04	1.2e+03	6	2	27	0.029	0.0596	12.6	6	2	15
x_2	701	1.01e+04	7.71e+04	252	590	365	35	15	15	0.21	0.845	9.4	14	4	5
x_3	7.92e+04	9.66e+03	9.57e+04	1.5e+04	416	417	1	15	26	0.00752	0.885	11.2	1	5	14
x_4	7.3e+04	1.86e+05	7.78e+04	1.43e+04	6.74e+04	444	0	3	20	0.00991	0.142	12.5	0	3	6
x_5	7.55e+04	5.45e+05	7.73e+04	1.42e+04	1.64e+05	393	2	3	15	0.0134	0.0907	10.1	2	3	3
x_6	7.47e+04	7.25e+03	7.81e+04	1.32e+04	170	408	5	20	21	0.0368	1.15	11.8	1	5	8
x_7	780	1.23e+05	7.78e+04	204	2.35e+04	377	19	4	30	0.114	0.0625	14.5	8	4	15
x_8	7.42e+04	7.76e+03	7.84e+04	1.46e+04	138	580	2	17	22	0.0247	0.97	13.5	0	4	6
x_9	5.96e+04	9.48e+03	7.9e+04	3.14e+04	180	863	1	18	24	0.00452	0.842	13.1	1	9	9
x_{10}	7.02e+04	1.18e+05	7.72e+04	1.29e+04	2.94e+04	361	2	5	15	0.00857	0.213	8.8	2	3	5

Table 6: TNM results on Extended Rosenbrock, with preconditioning applied.

Approximation: 0 Success, 33 Failures. The Hessian approximation resulted in a rapid stagnation of TNM.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	377	3.77e+03	3.77e+04	50.8	161	508	8	8	8	0.0192	0.0286	0.262	7	7	7
x_1	761	7.82e+03	7.74e+04	85.9	586	728	22	22	31	0.0125	0.0794	1.09	22	22	31
x_2	748	7.81e+03	7.74e+04	90.9	686	1.44e+03	21	24	22	0.0121	0.0889	0.791	21	24	22
x_3	749	7.67e+03	7.68e+04	51.1	284	1.39e+03	23	22	33	0.0132	0.0749	1.19	23	22	33
x_4	769	7.91e+03	8e+04	145	579	3.49e+03	18	25	26	0.012	0.104	0.895	18	25	26
x_5	765	7.92e+03	7.85e+04	63.9	639	2.06e+03	21	28	25	0.0106	0.101	0.887	21	28	25
x_6	725	7.67e+03	7.72e+04	60.9	312	1.26e+03	21	19	22	0.011	0.0673	0.779	21	19	22
x_7	746	7.87e+03	7.73e+04	98.5	281	1.88e+03	24	26	28	0.0136	0.0962	1.01	24	26	28
x_8	764	7.79e+03	7.8e+04	96.2	274	1.08e+03	24	30	34	0.0155	0.109	1.18	24	30	34
x_9	752	7.82e+03	7.79e+04	73.3	506	3.08e+03	19	25	27	0.0148	0.0946	0.955	19	25	27
x_{10}	801	7.97e+03	8.27e+04	98	428	2.52e+03	22	22	22	0.0119	0.0782	0.752	22	22	22

Table 7: TNM results on Extended Rosenbrock, using the Sparse Jacobian approximation ($h = 10^{-12}$).

Specific approximation: 0 Success, 33 Failures. The Hessian approximation resulted in a rapid stagnation of TNM.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	1.14e+03	1.14e+04	1.14e+05	91.1	288	911	6	6	6	0.013	0.0224	0.198	5	5	5
x_1	945	9.55e+03	9.62e+04	129	273	811	18	22	19	0.0107	0.08	0.664	18	22	19
x_2	1.29e+03	9.54e+03	9.53e+04	54.4	239	236	27	20	21	0.0158	0.0739	0.737	27	20	21
x_3	930	9.58e+03	9.52e+04	24.1	204	527	21	18	20	0.0129	0.0651	0.695	21	18	20
x_4	1.18e+03	9.57e+03	9.58e+04	98.4	146	620	22	19	20	0.0122	0.0728	0.74	22	19	20
x_5	1.3e+03	9.53e+03	9.73e+04	53.9	79.4	260	21	19	19	0.0108	0.0687	0.648	21	19	19
x_6	944	9.59e+03	9.54e+04	53.1	136	791	18	19	22	0.0103	0.0686	0.76	18	19	22
x_7	952	9.54e+03	9.61e+04	43.1	146	1.64e+03	22	19	19	0.0133	0.0704	0.705	22	19	19
x_8	947	9.65e+03	9.82e+04	45.1	120	1.37e+03	18	18	21	0.0114	0.0649	0.755	18	18	21
x_9	1.17e+03	9.5e+03	9.78e+04	56.7	229	775	19	18	23	0.0106	0.0694	0.785	19	18	23
x_{10}	953	9.53e+03	9.77e+04	50	121	807	18	18	20	0.00905	0.0642	0.686	18	18	20

Table 8: TNM results on Extended Rosenbrock, using the Sparse Jacobian approximation ($h = 10^{-12}$).

Extended Powell Badly scaled

Plain setup: 3 Success, 30 Failures. TNM stagnated for all random points.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	3.88e-07	2.08e-06	1.67e-05	1.35e-07	2.85e-07	8.91e-07	108	114	116	0.0295	0.207	2.18	7	5	8
x_1	1.33e+08	5.31e+08	1.84e+10	2.82e+09	3.14e+09	3.55e+10	45	48	39	0.0463	0.395	3.75	10	10	8
x_2	7.53e+07	1.03e+09	3.86e+10	1.5e+09	3.22e+10	6.44e+10	40	45	52	0.0372	0.384	5.06	6	11	7
x_3	7.12e+07	1.66e+09	2.51e+10	1.57e+09	1.23e+10	5.52e+10	41	45	53	0.0391	0.42	5.22	9	6	8
x_4	3.32e+07	8.32e+08	1.8e+10	8.14e+08	5.15e+09	1.19e+11	48	39	34	0.0445	0.33	3.16	7	11	12
x_5	4.53e+07	2.06e+09	1.59e+10	1.07e+09	2.18e+10	2.42e+10	55	45	38	0.0507	0.402	3.57	7	7	9
x_6	6.3e+07	2.2e+09	3.12e+10	1.34e+09	1.91e+10	7.42e+10	47	46	41	0.0449	0.419	4.05	7	8	7
x_7	1.32e+08	8.76e+08	1.9e+10	3.25e+09	9.31e+09	2.09e+10	41	49	29	0.0384	0.431	2.71	9	10	11
x_8	7.28e+07	1.89e+09	1.46e+10	1.31e+09	1.09e+10	1.96e+10	43	51	32	0.0415	0.455	2.99	6	7	8
x_9	1.11e+08	9.55e+08	1.83e+10	3.43e+09	5.69e+09	2.57e+10	40	45	30	0.0385	0.394	2.83	6	8	9
x_{10}	2.98e+07	2.12e+09	2.76e+10	6.38e+08	4.21e+10	5.41e+10	30	45	53	0.0282	0.39	5.26	9	8	8

Table 9: TNM results on Extended Powell.

Preconditioning: 3 Success, 30 Failures. TNM stagnated for all random points. In the case of x_0, employing a preconditioner resulted in a reduction in the number of iterations required to reach convergence, but increased the number of truncations required.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	9.49e-24	4.94e-23	1.76e-22	3.75e-07	1.44e-08	1.88e-11	87	87	87	0.0907	0.216	2.39	10	12	8
x_1	1.21e+10	1.78e+10	4.76e+11	2.7e+09	2.4e+10	1.61e+10	0	10	7	0.0201	0.788	6.02	0	0	0
x_2	1.1e+10	2.86e+10	1.29e+11	2.57e+09	1.41e+11	8.65e+10	0	10	11	0.00844	0.768	9.36	0	0	0
x_3	1.17e+10	3.1e+10	2.87e+10	2.66e+09	3.64e+10	2.07e+10	0	8	17	0.00961	0.673	14.6	0	0	0
x_4	1.13e+10	8.86e+10	1.21e+11	2.61e+09	1.68e+12	1.97e+11	0	2	6	0.00918	0.236	6.01	0	0	0
x_5	1.02e+10	5.03e+10	4.8e+11	2.49e+09	2.21e+11	1.62e+10	0	8	7	0.00836	0.614	6	0	0	0
x_6	1.05e+10	4.4e+10	9.73e+11	2.5e+09	1.87e+10	1.97e+15	1	9	4	0.0166	0.695	4.47	1	0	0
x_7	1.07e+10	4.06e+10	2e+11	3.09e+10	3.22e+12	1.53e+11	9	10	4	0.0745	0.84	4.25	2	0	0
x_8	1.11e+10	6.5e+10	1e+11	2.6e+09	6.35e+10	1.38e+10	0	12	10	0.00923	1.07	9.6	0	0	0
x_9	1.17e+10	1.08e+11	1.99e+11	2.7e+09	8.6e+10	8.72e+11	0	5	6	0.00924	0.46	6.08	0	0	0
x_{10}	9.85e+09	4.97e+10	6.71e+10	3.36e+09	1.2e+10	1.59e+12	9	8	32	0.0774	0.744	29.5	0	0	0

Table 10: TNM results on Extended Powell badly scaled, with preconditioning applied.

Approximation: 0 Success, 33 Failures. The Hessian approximation resulted in the stagnation of TNM.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	10.9	109	1.09e+03	0.973	3.08	9.73	9	9	9	0.0264	0.051	0.578	9	9	9
x_1	6.25e+04	1.55e+05	1.57e+06	2.18e+06	1.87e+06	4.58e+06	198	341	352	0.151	2.24	24.9	198	341	352
x_2	7.64e+03	1.4e+05	1.54e+06	4.75e+05	1.42e+06	4.44e+06	391	369	368	0.349	2.44	26.6	391	369	368
x_3	1.27e+05	1.73e+05	2.95e+06	5.95e+06	1.98e+06	4.73e+06	333	354	290	0.277	2.87	20.3	333	354	290
x_4	3.16e+04	1.85e+05	1.53e+06	7.9e+05	2.2e+06	4.63e+06	358	339	362	0.299	2.86	25.3	358	339	362
x_5	1.08e+05	1.8e+05	1.66e+06	5.64e+06	1.97e+06	4.76e+06	274	344	350	0.255	2.85	25.3	274	344	350
x_6	4.55e+04	2.46e+05	1.52e+06	1.87e+06	1.39e+06	4.38e+06	342	303	359	0.276	2.64	25.5	342	303	359
x_7	2.1e+04	1.44e+05	1.25e+06	1.03e+06	1.32e+06	4.85e+06	287	352	394	0.244	2.93	28	287	352	394
x_8	1.31e+04	2.49e+05	1.46e+06	4.01e+05	3.31e+06	4.53e+06	335	299	367	0.245	2.5	26.5	335	299	367
x_9	6.07e+04	6.93e+06	1.46e+06	4.02e+06	2.7e+07	4.57e+06	354	203	367	0.291	1.52	26.7	354	203	367
x_{10}	9.55e+03	1.24e+05	1.55e+06	3.71e+05	1.2e+06	4.72e+06	393	378	362	0.321	2.61	27.4	393	378	362

Table 11: TNM results on Extended Powell badly scaled, using the Sparse Jacobian approximation ($h = 10^{-12}$).

Specific approximation: 0 Success, 33 Failures. The Hessian approximation resulted in the stagnation of TNM.

	Function Value			Norm of Gradient			Iterations			Execution Time (seconds)			Truncation		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	51.1	511	5.11e+03	5.88e+04	1.86e+05	5.88e+05	47	47	47	0.053	0.285	2.88	46	46	46
x_1	3.69e+05	4.2e+06	4.53e+07	2.54e+06	1.31e+07	3.27e+07	233	228	225	0.188	1.66	17.4	233	228	225
x_2	8.43e+04	4.19e+06	4.91e+07	1.98e+06	2.32e+07	3.93e+07	353	239	220	0.263	1.7	17.1	353	239	220
x_3	3.46e+05	4.27e+06	4.79e+07	5.68e+06	1.52e+07	3.43e+07	239	233	222	0.191	1.68	17.4	239	233	222
x_4	3.29e+05	4.02e+06	4.68e+07	4.69e+06	1.19e+07	4.46e+07	248	231	224	0.194	1.66	17.4	248	231	224
x_5	5.37e+05	3.76e+06	4.58e+07	6.25e+06	9.15e+06	3.79e+07	213	234	224	0.165	1.67	17.4	213	234	224
x_6	4.69e+05	4.28e+06	4.61e+07	3.14e+06	2.04e+07	4.13e+07	213	225	225	0.175	1.6	17.5	213	225	225
x_7	3.72e+05	3.76e+06	4.68e+07	3.95e+06	9.12e+06	3.79e+07	227	234	223	0.176	1.7	17.4	227	234	223
x_8	2.93e+05	3.68e+06	4.5e+07	2.94e+06	9.27e+06	5.34e+07	243	233	226	0.197	1.7	17.5	243	233	226
x_9	1.66e+05	4.48e+06	4.57e+07	3.87e+06	1.22e+07	3.48e+07	255	226	224	0.199	1.63	17.3	255	226	224
x_{10}	3.09e+05	4.43e+06	4.78e+07	5.23e+06	2.07e+07	4.76e+07	236	226	222	0.183	1.63	17	236	226	222

Table 12: TNM results on Extended Powell badly scaled, using the specific Sparse Jacobian approximation ($h = 10^{-12}$).

A.2 Modified Newton Method

A.2.1 Minimal eigenvalue correction

Problem 82[2]

Preconditioning: 33 Success, 0 Failures.

	Function Value			Norm of Gradient			Iterations			Execution Time		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	9.88e-19	1.06e-17	1.07e-16	1.41e-09	4.6e-09	1.46e-08	5	5	5	0.349	0.0669	0.235
x_1	1.74e-09	3.02e-15	3.2e-15	5.89e-05	7.77e-08	8e-08	9	11	12	0.0846	0.113	0.768
x_2	1.9e-09	4.42e-12	5.72e-15	6.17e-05	2.97e-06	1.07e-07	8	10	12	0.0339	0.1	0.769
x_3	1.43e-11	8.45e-12	1.49e-15	5.34e-06	4.11e-06	5.46e-08	9	11	13	0.0384	0.114	0.904
x_4	1.67e-10	4.83e-17	5.07e-11	1.83e-05	9.83e-09	1.01e-05	10	11	12	0.038	0.12	0.859
x_5	1.38e-16	1.52e-15	6.11e-10	1.66e-08	5.5e-08	3.5e-05	10	11	11	0.0418	0.104	0.889
x_6	9.59e-14	1.69e-16	2.83e-15	4.38e-07	1.84e-08	7.53e-08	10	11	12	0.0429	0.113	0.863
x_7	4.77e-12	3.28e-15	3.81e-17	3.09e-06	8.1e-08	8.73e-09	9	11	12	0.0375	0.127	0.942
x_8	1.64e-10	1.63e-15	5.4e-10	1.81e-05	5.7e-08	3.29e-05	9	11	11	0.0394	0.115	0.816
x_9	3.52e-12	4.58e-16	4.76e-11	2.65e-06	3.03e-08	9.75e-06	9	11	11	0.0354	0.109	0.873
x_{10}	4.93e-14	4.88e-10	8.22e-15	3.14e-07	3.13e-05	1.28e-07	10	10	12	0.0566	0.128	0.784

Table 13: MNM results on Problem 82, with preconditioning applied. Minimal eigenvalue correction with $\delta = 1$.

[2] *Plain setup* and *Specific Approximation* executions are not available. MNM never converged for exhausting limited runtime.

Approximation: 0 Success, 33 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

	Function Value			Norm of Gradient			Iterations			Execution Time		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	30.9	310	3.1e+03	5.61	17.7	56.1	1	1	1	0.0426	0.0616	0.3
x_1	84.2	812	8.18e+03	10.7	32.9	105	1	1	1	0.0336	0.0574	0.285
x_2	77.5	806	8.16e+03	10.4	32.9	104	1	1	1	0.0321	0.0583	0.265
x_3	85.6	814	8.16e+03	10.3	33.1	105	1	1	1	0.032	0.0565	0.263
x_4	81.5	820	8.22e+03	10.3	33.4	104	1	1	1	0.0334	0.058	0.277
x_5	78.6	807	8.23e+03	10.2	33	105	1	1	1	0.0294	0.0546	0.261
x_6	77.3	812	8.2e+03	10.3	33	105	1	1	1	0.0327	0.0531	0.276
x_7	81.4	835	8.17e+03	10.6	33.3	104	1	1	1	0.0305	0.0572	0.261
x_8	83.4	821	8.19e+03	10.8	32.9	104	1	1	1	0.0309	0.0578	0.266
x_9	79.8	820	8.2e+03	10.2	33	105	1	1	1	0.0305	0.0575	0.271
x_{10}	82.6	833	8.19e+03	10.2	33.2	104	1	1	1	0.0326	0.0621	0.275

Table 14: MNM results on Problem 82, using the Sparse Jacobian approximation ($h = 10^{-12}$). Minimal eigenvalue correction with $\delta = 1$.

Extended Rosenbrock[3]

Plain setup: 3 Success, 30 Failures. Random points produced non-SPD matrices that MNM was not able to correct.

	Function Value			Norm of Gradient			Iterations			Execution Time		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	9.36e-19	9.36e-18	9.36e-17	5e-09	1.58e-08	5e-08	21	21	21	0.0553	0.0628	0.366
x_1	986	1.11e+04	1.09e+05	123	1.56e+03	3.12e+03	4	3	3	0.201	1.72	25.2
x_2	940	1.13e+04	1.09e+05	37.7	2e+03	2.91e+03	5	3	3	0.286	1.43	22.7
x_3	960	1.14e+04	1.09e+05	40.9	1.92e+03	2.4e+03	6	3	3	0.239	1.34	23.5
x_4	857	1.1e+04	1.09e+05	50.4	1.42e+03	2.62e+03	6	3	3	0.303	1.79	22.1
x_5	1.01e+03	1.13e+04	2.92e+05	91.8	2.07e+03	2.5e+04	4	3	2	0.207	1.38	15.7
x_6	1.04e+03	1.14e+04	1.09e+05	74.6	2.27e+03	3.33e+03	4	3	3	0.212	1.56	19.6
x_7	913	1.09e+04	1.09e+05	46.5	1.03e+03	2.82e+03	6	3	3	0.249	1.59	23.4
x_8	1.01e+03	1.04e+04	2.84e+05	81.5	145	2.4e+04	4	4	2	0.247	1.48	14.2
x_9	1.11e+03	1.1e+04	3.27e+05	70.4	1.5e+03	3.04e+04	4	3	2	0.213	1.54	15
x_{10}	961	1.05e+04	3.19e+05	42	153	2.95e+04	5	4	2	0.227	1.46	14.9

Table 15: MNM results on Extended Rosenbrock. Minimal eigenvalue correction with $\delta = 1e-8$

[3] *Preconditioning* and *Specific Approximation* executions are not available. MNM never converged for exhausting limited runtime.

Approximation: 0 Success, 33 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

	Function Value			Norm of Gradient			Iterations			Execution Time		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	318	3.21e+03	3.31e+04	38.5	123	409	166	87	62	0.313	0.787	3.99
x_1	1.42e+04	3.76e+05	2.59e+06	5.06e+03	6.54e+04	1.26e+05	3	2	5	0.0365	0.0796	1.41
x_2	1.47e+04	4.69e+05	2.52e+06	5.31e+03	7.74e+04	1.18e+05	4	3	1	0.0384	0.11	0.79
x_3	5.78e+04	4.95e+05	2.24e+06	1.25e+04	7.64e+04	1e+05	1	2	1	0.0361	0.0853	0.769
x_4	6.08e+04	3.25e+05	2.36e+06	2.02e+04	5.75e+04	1.08e+05	1	2	1	0.0309	0.0812	0.999
x_5	5.23e+04	5.28e+05	2.28e+06	1.49e+04	8.44e+04	1.05e+05	2	2	1	0.036	0.0997	0.936
x_6	3.83e+04	5.62e+05	2.78e+06	9.58e+03	9.44e+04	1.35e+05	2	2	1	0.0334	0.105	1.08
x_7	3.54e+04	2.65e+05	2.43e+06	1.62e+04	4.32e+04	1.15e+05	1	5	2	0.0372	0.118	1.07
x_8	7.24e+04	3.89e+05	2.2e+06	2.96e+04	3.21e+04	1.01e+05	5000	2	1	8.01	0.182	1.05
x_9	5.72e+04	3.37e+05	2.66e+06	1.27e+04	5.95e+04	1.28e+05	1	1	1	0.0361	0.077	1.75
x_{10}	2.34e+04	4.22e+05	2.56e+06	9.95e+03	3.72e+04	1.23e+05	1	2	1	0.0319	0.103	0.848

Table 16: MNM results on Extended Rosenbrock, the Sparse Jacobian approximation ($h = 10^{-12}$). Minimal eigenvalue correction with $\delta = 1e - 8$

Extended Powell Badly scaled[4]

Plain setup: 3 Success, 30 Failures. Random points produced non-SPD matrices that MNM was not able to correct.

	Function Value			Norm of Gradient			Iterations			Execution Time		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	6.39e-19	1.2e-16	7.78e-22	9.09e-05	6.14e-05	5.24e-07	87	86	88	0.106	0.332	2.42
x_1	1.21e+10	1.11e+11	1.11e+12	2.7e+09	8.19e+09	2.59e+10	0	0	0	0.0285	0.0372	0.171
x_2	1.1e+10	1.1e+11	1.11e+12	2.57e+09	8.14e+09	2.58e+10	0	0	0	0.0394	0.0803	0.15
x_3	1.17e+10	1.11e+11	1.11e+12	2.66e+09	8.2e+09	2.59e+10	0	0	0	0.03	0.051	0.161
x_4	1.13e+10	1.11e+11	1.13e+12	2.61e+09	8.19e+09	2.61e+10	0	0	0	0.0278	0.039	0.143
x_5	1.02e+10	1.11e+11	1.12e+12	2.49e+09	8.14e+09	2.6e+10	0	0	0	0.0285	0.0374	0.129
x_6	1.05e+10	1.11e+11	1.1e+12	2.5e+09	8.21e+09	2.57e+10	0	0	0	0.0259	0.0394	0.137
x_7	1.11e+10	1.11e+11	1.11e+12	2.57e+09	8.25e+09	2.58e+10	0	0	0	0.0269	0.0373	0.134
x_8	1.11e+10	1.11e+11	1.11e+12	2.6e+09	8.23e+09	2.58e+10	0	0	0	0.0269	0.0376	0.129
x_9	1.17e+10	1.1e+11	1.11e+12	2.7e+09	8.07e+09	2.59e+10	0	0	0	0.0251	0.0381	0.132
x_{10}	1.06e+10	1.13e+11	1.11e+12	2.5e+09	8.26e+09	2.58e+10	0	0	0	0.0255	0.0375	0.126

Table 17: MNM results on Extended Powell badly scaled. Minimal eigenvalue correction with $\delta = 1e-8$

[4] *Approximation* and *Specific Approximation* executions are not available. MNM never converged for exhausting limited runtime.

Preconditioning: 3 Success, 30 Failures. Random points produced non-SPD matrices that MNM was not able to correct.

	Function Value			Norm of Gradient			Iterations			Execution Time		
	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5	10^3	10^4	10^5
x_0	3.57e-22	1.75e-24	1.51e-24	3.14e-07	1.16e-07	3.26e-09	90	89	90	0.0714	0.275	2.13
x_1	1.2e+10	1.11e+11	1.11e+12	2.69e+09	8.18e+09	2.59e+10	1	2	1	0.0436	0.199	2.1
x_2	1.1e+10	1.1e+11	1.11e+12	2.57e+09	8.14e+09	2.58e+10	0	3	1	0.0389	0.311	1.53
x_3	1.17e+10	1.11e+11	1.11e+12	2.66e+09	8.19e+09	2.59e+10	0	1	0	0.0289	0.105	0.485
x_4	1.11e+10	1.11e+11	1.13e+12	2.59e+09	8.19e+09	2.61e+10	1	0	3	0.033	0.0567	4.18
x_5	1.02e+10	1.1e+11	1.12e+12	2.49e+09	8.15e+09	2.6e+10	0	1	1	0.0372	0.121	2.02
x_6	1.05e+10	1.11e+11	1.1e+12	2.5e+09	8.21e+09	2.57e+10	0	0	0	0.0281	0.0683	0.582
x_7	1.11e+10	1.11e+11	1.11e+12	2.57e+09	8.25e+09	2.58e+10	0	0	2	0.0305	0.0666	3.78
x_8	1.09e+10	1.11e+11	1.11e+12	2.63e+09	8.23e+09	2.58e+10	3	0	0	0.0547	0.0765	0.527
x_9	1.17e+10	1.1e+11	1.11e+12	2.7e+09	8.07e+09	2.59e+10	0	0	0	0.0296	0.0613	0.743
x_{10}	1.02e+10	1.13e+11	1.11e+12	2.45e+09	8.26e+09	2.58e+10	2	0	0	0.0393	0.0504	0.64

Table 18: MNM results on Extended Powell badly scaled, with preconditioning applied. Minimal eigenvalue correction with $\delta = 1e - 8$

A.2.2 Diagonalization correction

Due to the high computational cost of calculating the eigenvalues of the Hessian, only the $n = 10^3$ cases are reported.

Problem 82[5]

Preconditioning: 11 Success, 0 Failures

Table 19: MNM results on Problem 82, with preconditioning applied. Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	8.35e-16	4.09e-08	5	0.607
x_1	1.31e-12	1.62e-06	77	4.55
x_2	7.83e-10	3.96e-05	62	4.15
x_3	5.08e-12	3.19e-06	67	4.31
x_4	2.05e-10	2.03e-05	79	4.88
x_5	7.9e-16	3.97e-08	56	3.72
x_6	8.23e-10	4.06e-05	75	4.72
x_7	6.22e-11	1.12e-05	53	5.44
x_8	1.01e-11	4.5e-06	76	4.84
x_9	3.62e-17	8.51e-09	47	3.61
x_10	1.65e-09	5.75e-05	50	3.31

[5] *Plain setup* execution is not available. MNM never converged for exhausting limited runtime.

Approximation: 0 Success, 11 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

Table 20: MNM results on Problem 82, using the Sparse Jacobian approximation ($h = 10^{-12}$). Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	4.36	3.2	1	0.939
x_1	221	25.6	1	0.144
x_2	206	24.5	1	0.128
x_3	210	24.8	1	0.13
x_4	209	24.8	1	0.151
x_5	206	24.7	1	0.146
x_6	206	24.8	1	0.138
x_7	214	25.2	1	0.201
x_8	224	26.3	1	0.158
x_9	206	24.6	1	0.208
x_10	205	24.4	1	0.174

Specific approximation: 0 Success, 11 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

Table 21: MNM results on Problem 82, using the specific Sparse Jacobian approximation ($h = 10^{-12}$). Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	4.36	3.2	1	1.01
x_1	168	19.4	1	0.139
x_2	154	18.3	1	0.113
x_3	171	19.6	1	0.123
x_4	160	18.8	1	0.13
x_5	151	17.8	1	0.119
x_6	154	18.3	1	0.122
x_7	167	19.5	1	0.13
x_8	168	19.5	1	0.134
x_9	159	19.1	1	0.141
x_10	164	18.9	1	0.129

Extended Rosenbrock[6]

Preconditioned: 11 Success, 0 Failures

Table 22: MNM results on Extended Rosenbrock, with preconditioning applied. Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	9.36e-19	5e-09	21	0.0419
x_1	1.68e-12	3.55e-05	86	0.959
x_2	2.86e-19	1.49e-08	80	0.747
x_3	2.15e-16	3.87e-07	92	0.891
x_4	1.42e-16	2.85e-07	98	1.01
x_5	5.14e-10	3.31e-05	92	1.11
x_6	1.79e-18	3.77e-08	83	0.937
x_7	6.11e-15	2e-06	97	1.09
x_8	2.21e-16	2.02e-07	94	0.956
x_9	1.48e-19	1.42e-09	100	1.04
x_10	1.81e-14	9.15e-08	86	0.937

[6] *Plain setup* execution is not available. MNM never converged for exhausting limited runtime.

Approximation: 0 Success, 11 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

Table 23: MNM results on Extended Rosenbrock, using the Sparse Jacobian approximation ($h = 10^{-12}$). Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	1.18e+03	52.1	1	0.0557
x_1	7.02e+04	1.47e+04	1	0.0527
x_2	6.98e+04	1.45e+04	1	0.0531
x_3	7.41e+04	1.52e+04	1	0.0538
x_4	6.77e+04	1.43e+04	1	0.0507
x_5	7.14e+04	1.48e+04	1	0.051
x_6	7.06e+04	1.46e+04	1	0.0511
x_7	6.72e+04	1.43e+04	1	0.0508
x_8	6.94e+04	1.47e+04	1	0.0532
x_9	7.08e+04	1.48e+04	1	0.053
x_10	6.56e+04	1.41e+04	1	0.0499

Specific approximation: 0 Success, 11 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

Table 24: MNM results on Extended Rosenbrock, using the specific Sparse Jacobian approximation ($h = 10^{-12}$). Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	1.14e+03	91.1	2	0.075
x_1	7.21e+04	1.47e+04	1	0.0538
x_2	7.17e+04	1.45e+04	1	0.0556
x_3	7.58e+04	1.52e+04	1	0.0586
x_4	6.92e+04	1.43e+04	1	0.055
x_5	7.33e+04	1.48e+04	1	0.0525
x_6	7.21e+04	1.46e+04	1	0.0541
x_7	6.88e+04	1.43e+04	1	0.0528
x_8	7.11e+04	1.47e+04	1	0.0531
x_9	7.28e+04	1.48e+04	1	0.0535
x_10	6.75e+04	1.41e+04	1	0.0526

Extended Powell badly scaled

Plain setup: 1 Success, 10 Failures. MNM only converged for the suggested starting point.

Table 25: MNM results on Extended Powell badly scaled. Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	0.000984	9.31e-05	24	0.103
x_1	1.21e+10	2.7e+09	0	0.0399
x_2	1.1e+10	2.57e+09	0	0.0404
x_3	1.17e+10	2.66e+09	0	0.0406
x_4	1.13e+10	2.61e+09	0	0.0415
x_5	1.02e+10	2.49e+09	0	0.0402
x_6	1.05e+10	2.5e+09	0	0.0398
x_7	1.11e+10	2.57e+09	0	0.037
x_8	1.11e+10	2.6e+09	0	0.0384
x_9	1.17e+10	2.7e+09	0	0.0384
x_10	1.06e+10	2.5e+09	0	0.0404

Preconditioning: 1 Success, 10 Failures. Using a preconditioner caused the Hessian matrix to become more unstable.

Table 26: MNM results on Extended Powell badly scaled, with preconditioning applied. Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	9.23e-22	8.82e-08	86	0.123
x_1	1.21e+10	2.7e+09	0	0.0406
x_2	1.1e+10	2.57e+09	0	0.0375
x_3	1.17e+10	2.66e+09	0	0.0385
x_4	1.13e+10	2.61e+09	0	0.0383
x_5	1.02e+10	2.49e+09	0	0.0385
x_6	1.05e+10	2.5e+09	0	0.0372
x_7	1.11e+10	2.57e+09	0	0.0405
x_8	1.11e+10	2.6e+09	0	0.0398
x_9	1.17e+10	2.7e+09	0	0.0394
x_10	1.06e+10	2.5e+09	0	0.0371

Approximation: 0 Success, 11 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

Table 27: MNM results on Extended Powell badly scaled, using the Sparse Jacobian approximation ($h = 10^{-12}$). Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	85.1	2.32e+05	2	0.0993
x_1	1.21e+10	2.7e+09	0	0.044
x_2	1.1e+10	2.57e+09	0	0.042
x_3	1.17e+10	2.66e+09	0	0.0451
x_4	1.13e+10	2.61e+09	0	0.041
x_5	1.02e+10	2.49e+09	0	0.0401
x_6	1.05e+10	2.5e+09	0	0.0393
x_7	1.11e+10	2.57e+09	0	0.0398
x_8	1.11e+10	2.6e+09	0	0.0402
x_9	1.17e+10	2.7e+09	0	0.0386
x_10	1.06e+10	2.5e+09	0	0.0376

Specific approximation: 0 Success, 11 Failures. The Hessian approximation produced a non-SPD matrix that MNM was not be able to correct.

Table 28: MNM results on Extended Powell badly scaled, using the specific Sparse Jacobian approximation ($h = 10^{-12}$). Diagonalization correction with $\delta = 1e - 8$.

	Function Value	Norm of Gradient	Iterations	Execution Time
x_0	85.1	2.32e+05	2	0.0787
x_1	1.21e+10	2.7e+09	0	0.0415
x_2	1.1e+10	2.57e+09	0	0.0443
x_3	1.17e+10	2.66e+09	0	0.0389
x_4	1.13e+10	2.61e+09	0	0.0417
x_5	1.02e+10	2.49e+09	0	0.0411
x_6	1.05e+10	2.5e+09	0	0.0407
x_7	1.11e+10	2.57e+09	0	0.0425
x_8	1.11e+10	2.6e+09	0	0.0419
x_9	1.17e+10	2.7e+09	0	0.04
x_10	1.06e+10	2.5e+09	0	0.0407

B Matlab code implementation

B.1 Modified Newton Method

```
function [xk, fk, gradfk_norm, k, failure, flag, xseq, btseq, corrseq, fseq,
↪ gradnormseq] = ...
    modifiedNM(...
    f, gradf, Hessf, x0, kmax, tolgrad, ...
    c1, rho, btmax, precond, h, specific_approx, hess_approx,
    ↪ correction_technique, varargin)
%   MODIFIEDNM Modified Newton's Method with Hessian Correction Techniques
%
%   This function solves unconstrained optimization problems using
%   modified Newton's method with various Hessian correction
%   techniques to handle non-positive definite Hessians.
%
%   Syntax:
%   [xk, fk, gradfk_norm, k, failure, flag, xseq, btseq, corrseq, fseq,
↪ gradnormseq] = ...
%       modifiedNM(f, gradf, Hessf, x0, kmax, tolgrad, ...
%       c1, rho, btmax, precond, h, specific_approx, hess_approx,
↪ correction_technique, varargin)
%
%   Input Parameters:
%       - f                 : Function handle for the objective function.
%       - gradf             : Function handle for the gradient of the
↪ objective function.
%       - Hessf             : Function handle for the Hessian of the
↪ objective function (or empty for approximation).
%       - x0                : Initial guess for the solution.
%       - kmax              : Maximum number of iterations.
%       - tolgrad           : Tolerance for the norm of the gradient.
%       - c1                : Armijo condition parameter (0 < c1 < 1).
%       - rho               : Backtracking step reduction factor (0 < rho <
↪ 1).
%       - btmax             : Maximum number of backtracking steps.
%       - precond           : Boolean indicating whether to use
↪ preconditioning.
```

```
27      %       - h                  : Step size for numerical Hessian approximation
        ↪  (if needed).
28      %       - specific_approx    : Boolean indicating usage of gradient of f for
        ↪  exact Hessian approximation.
29      %       - hess_approx        : Function handle for Hessian approximation
        ↪  (ignored if Hessf is not empty).
30      %       - correction_technique : String specifying the correction method among
        ↪  {'minima', 'diag'}.
31      %       - varargin           : Additional parameters for the chosen correction
        ↪  technique.
32      %
33      %   Correction Parameters:
34      %       - 'minima': Tolerance for minimal eigenvalue correction ('toleig').
35      %       - 'diag'  : Tolerance for diagonalization correction ('toleig').
36      %
37      %   Output Parameters:
38      %       - xk                 : Final solution vector.
39      %       - fk                 : Objective function value at the solution.
40      %       - gradfk_norm        : Norm of the gradient at the solution.
41      %       - k                  : Total number of iterations performed.
42      %       - failure            : Boolean indicating whether the method failed.
43      %       - flag               : Message describing the termination condition.
44      %       - xseq               : Sequence of solution vectors across iterations.
45      %       - btseq              : Sequence of backtracking step counts.
46      %       - corrseq            : Sequence of corrections applied to the Hessian.
47      %       - fseq               : Sequence of objective function values across
        ↪  iterations.
48      %       - gradnormseq        : Sequence of gradient norms across iterations.
49      %
50      %
51      %   Notes:
52      %       - If Hessf is empty, numerical approximation of the Hessian is used.
53      %       - Correction techniques preserve sparsity when applied.
54
55      % Parse additional parameters from varargin
56      correction_params = varargin;
57
58      % Import all various matrix corrections
```

```
59      addpath(fullfile(pwd, 'matrix_corrections/'));
60
61      % Define function handle for correction, based on the user choice
62      switch correction_technique
63          case 'minima'
64              correction = @(X) minimal_eigenvalue_correction(X,
                ↪  correction_params{:});
65          case 'diag'
66              correction = @(X) diagonalization_correction(X, correction_params{:});
67          otherwise
68              error('Unknown correction technique: %s', correction_technique);
69      end
70
71      % Function handle for the armijo condition
72      farmijo = @(fk, alpha, c1_gradfk_pk) ...
73          fk + alpha * c1_gradfk_pk;
74
75      % Solution sequence tracking variable initialization
76      xseq = zeros(length(x0), kmax);
77      btseq = zeros(1, kmax);
78      corrseq = zeros(1, kmax);
79      fseq = zeros(1, kmax);
80      gradnormseq = zeros(1, kmax);
81
82      % Failure traickig variables initialization
83      flag = '';
84      failure = false;
85
86      % Starting values initialization
87      k = 0;
88      xk = x0;
89      fk = f(xk);
90      gradfk = gradf(xk);
91      gradfk_norm = norm(gradfk);
92
93      fseq(1) = fk;
94      gradnormseq(1) = gradfk_norm;
95
```

```
96      % Check whetere to use hessian approximation or not
97      if isempty(Hessf)
98          % If specific_approx = true, the function will exploit the approximatio on
            ↪  h * abs(xk)
99          Hessf = @(x) hess_approx(x, h, specific_approx, gradf, gradfk);
100     end
101
102     % Stop when stopping criteria is met
103     while k < kmax && gradfk_norm >= tolgrad
104
105         % Compute Hessian (sparse)
106         Hk = Hessf(xk);
107
108         try
109             Bk = Hk;
110             R = ichol(Bk); % Attempt (incomplete) Cholesky factorization, if not
                ↪  P.D. error will raise
111         catch
112             % If it fails again then Bk is not P.D.
113             Bk = correction(Hk); % Correct Bk using the choosen approach (will
                ↪  preserve sparsity)
114             try
115                 R = ichol(Bk); % Retry Cholesky, if still not P.D. error will
                    ↪  raise
116             catch
117                 failure = true;
118                 flag = 'Corrected matrix Bk is not S.P.D.';
119                 break;
120             end
121         end
122
123         % If no precodition is required, ignore it (overwrite the cholesky
            ↪  factorization with empty matrix)
124         if ~precond
125             R = [];
126         end
127
128         % Continue with the common Newton method with backtracking
```

```
129         [pk, ~, ~, ~, ~] = pcg(Bk, -gradfk, [], [], R, R');
130
131         % Reset the value of alpha
132         alpha = 1;
133
134         % Compute the candidate new xk
135         xnew = xk + alpha * pk;
136
137         % Compute the value of f in the candidate new xk
138         fnew = f(xnew);
139
140         % Backtracking auxiliar variables
141         c1_gradfk_pk = c1 * gradfk' * pk;
142         bt = 0;
143
144         % Backtracking strategy
145         while bt < btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
146             % Reduce the value of alpha
147             alpha = rho * alpha;
148
149             % Update xnew and fnew w.r.t. the reduced alpha
150             xnew = xk + alpha * pk;
151             fnew = f(xnew);
152
153             % Increase the counter by one
154             bt = bt + 1;
155         end
156
157         % If backtracking met stopping criteria raise an error
158         if bt == btmax && fnew > farmijo(fk, alpha, c1_gradfk_pk)
159             flag = sprintf([...
160                 'Failure: Could not satisfy Armijo. ' ...
161                 'Details: ' ...
162                 '  Iteration: %d, ' ...
163                 '  New function value (fnew): %e, ' ...
164                 '  Armijo condition value: %e, ' ...
165                 '  Step length (alpha): %e' ...
166                 ], ...
```

```
167                 k, fnew, farmijo(fk, alpha, c1_gradfk_pk), alpha);
168             failure = true;
169             break;
170         end
171
172         % Update xk, fk, gradfk_norm
173         xk = xnew;
174         fk = fnew;
175         gradfk = gradf(xk);
176         gradfk_norm = norm(gradfk);
177
178         % Increase the step by one
179         k = k + 1;
180
181         % Store current xk in xseq
182         xseq(:, k) = xk;
183         % Store bt iterations in btseq
184         btseq(k) = bt;
185         % Store the correction applied
186         corrseq(k) = norm(Hk - Bk, 'fro');
187
188         % Store function value
189         fseq(k) = fk;
190         % Store gradient norm value
191         gradnormseq(k) = gradfk_norm;
192     end
193
194     % Flag output
195     if ~failure
196         if gradfk_norm < tolgrad % Newton method converged
197             flag = sprintf('Satysfied the tollerance in %d iteration', k);
198         else % Newton method did not converge
199             flag = sprintf(['Failure: mnm did %d iteration but did not converge.
                ↪  ' ...
200                 'Norm of the gradient = %.3g'], k, gradfk_norm);
201             failure = true;
202         end
203     end
```

```
204
205     % "Cut" xseq and btseq to the correct size
206     xseq = xseq(:, 1:k);
207     btseq = btseq(1:k);
208
209     % "Add" x0 at the beginning of xseq (otherwise the first el. is x1)
210     xseq = [x0, xseq];
211
212     end
```

B.1.1 Minimal eigenvalue correction

```
function Bk = minimal_eigenvalue_correction(Hk, toleig)
    % Function to apply minimal eigenvalue correction while preserving sparsity

    if nargin < 2 || isempty(toleig) % Set default value for toleig
        toleig = 1e-8;
    end

    if ~issymmetric(Hk) % Check if the matrix is symmetric
        error('Matrix is not symmetric');
    end

    % Compute the smallest eigenvalue
    min_eig = eigs(Hk, 1, 'smallestreal', 'IsSymmetricDefinite', false, 'tol',
    ↪  1e-8, 'MaxIterations', 500);

    % Compute the correction term
    tauk = max(0, toleig - min_eig);

    % Add correction while preserving sparsity
    Bk = Hk + tauk * speye(size(Hk));
end
```

B.1.2 Diagonalization correction

```
function Bk = diagonalization_correction(Hk, toleig)
    % Function to apply diagonalization correction while preserving sparsity

    if nargin < 2 || isempty(toleig) % Set default value for toleig
        toleig = 1e-8;
    end

    if ~issymmetric(Hk) % Check if the matrix is symmetric
        error('Matrix is not symmetric');
    end

    n = size(Hk, 1);

    % Compute eigendecomposition (use eigs() - rather than eig() - with size of Hk since we need *all* eigenvalues, note: eig() with sparse matrices can't return eigenvectors)
    [V, D] = eigs(Hk, n, 'largestabs', 'IsSymmetricDefinite', false, 'tol', 1e-8, 'MaxIterations', 500);

    % Threshold eigenvalues
    D = spdiags(max(diag(D), toleig), 0, n, n); % Ensure eigenvalues are >= toleig and enforce sparsity (eigs() do not preserve sparsity)
    V = sparse(V); % Enforce sparsity on V (eigs() do not preserve sparsity)

    % Reconstruct the matrix
    Bk = V * D * V';
end
```

B.2 Truncated Newton Method

```
function [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
    x_sequence, backtrack_sequence, pcg_sequence] = ...
    truncatedNM(f, grad_f, hess_f, x_initial, max_iteration, ...
    tollerance, c1, rho, max_backtrack, do_pcg_precond, ...
    h, specific_approx, hess_approx)
%TRUNCATEDNM - Truncated Newton's Method
%
%   Syntax
%       [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
%           x_sequence, backtrack_sequence, pcg_sequence] =
%       truncatedNM(f, grad_f, hess_f, x_initial, max_iteration, ...
%           tollerance, c1, rho, max_backtrack, do_pcg_precond)
%
%   Input Parameters:
%       f - Describe the function to minimaze
%           function handle
%       grad_f - Compute the gradient of f
%           function handle
%       hess_f - Compute the hessian of f
%           function handle
%       x_initial - Starting point for TNM
%           column vector
%       max_iteration - Maximum number iterations that TNM can perform
%           positive scalar integer
%       tollerance - Tollerance for stopping criteria (absolute residual)
%           positive scalar
%       c1 - Parameter for Armijo condition
%           positive scalar
%       rho - Parameter for Armijo condition
%           positive scalar
%       max_backtrack Maximum iterations for backtracking
%           positive scalar integer
%       do_pcg_precond - Indicates if apply preconditioning to Hessian;
%           boolean
%       h - Level of approximation
```

```
36  %           positive scalar
37  %       specific_approx - Indicates if apply specific approximation;
38  %           boolean
39  %       hess_approx - Compute the approximated hessian
40  %           function handle
41  %
42  %   Output:
43  %       x_found - Solution found by TNM
44  %           column vector;
45  %       f_x - Function value at x_found
46  %           positive scalar
47  %       norm_grad_f_x - Norm of the gradient (if failure == false, should be 0)
48  %           positive scalar
49  %       iteration - Number of iteration performed
50  %           positive scalar integer
51  %       failure - Indicates if a failure happend
52  %           boolean
53  %       flag - TNM output description
54  %           string
55  %       x_sequence - Value of x computed at each iteration
56  %           matrix
57  %       backtrack_sequence - Number of iteration for backtrack at each outer
    ↪  iteration
58  %           row vector
59  %       pcg_sequence - pcg performance at each iteration (iteration, flag,
    ↪  preconditioning type)
60  %           matrix
61
62  % Starting values initialization
63  x_k = x_initial;
64  f_xk = f(x_k);
65  grad_f_xk = grad_f(x_k);
66  norm_grad_f_xk = norm(grad_f_xk);
67
68  % Check if hessian approximation is needed
69  if isempty(hess_f)
70      hess_f = @(x) hess_approx(x, h, specific_approx, grad_f, grad_f_xk);
71  end
```

```
72
73  % x_k, pcg and backtracking sequences
74  x_sequence = zeros(length(x_initial), max_iteration);
75  pcg_sequence = zeros(3, max_iteration); % [iteration + flag + precond] saved at
    ↪  each iteration
76  backtrack_sequence = zeros(max_iteration);
77
78  % PCG default parameters
79  precond = [];
80  pcg_tol = 1e-6;
81  pcg_maxit = 50;
82
83  % Stagnation variables
84  stagnation = 0;
85  max_stagnation = 5;
86  stagn_threshold = 1e-2;
87
88  % Iteration variables
89  i = 0; % current iteration
90  failure = false;
91
92  % -- Loop --
93  while i < max_iteration && ...                    % iteration
94          norm_grad_f_xk >= tollerance && ...       % stopping condition
95          stagnation < max_stagnation               % stagnation
96
97      % -- Computing descent direction --
98      % Compute the preconditioning matrix for pcg
99      A = hess_f(x_k);
100     precond_type = -1; % no preconditioning required
101     if do_pcg_precond
102         try
103             try
104                 % ichol is more stable and usually a better option
105                 precond = ichol(sparse(A));
106                 precond_type = 1; % ichol
107             catch ME
108                 % A is not SPD matrix, using ilu preconditioning
```

```
109                 precond = ilu(sparse(A));
110                 precond_type = 2; % ilu
111             end
112         catch ME
113             % Cannot apply preconditioning on the matrix A
114             % The algorithm continues, without apply preconditioning
115 
116             precond = [];
117             precond_type = 0; % cannot use preconditioning
118         end
119     end
120 
121     % Using -grad_f(xk) as starting point will guarantee that pcg will
122     % return a descent direction, even if matrix A is not SPD
123     [desc_dir, pcg_flag, ~, pcg_iter, ~] = ...
124         pcg(A, -grad_f_xk, pcg_tol, pcg_maxit, precond, precond', -grad_f_xk);
125 
126     % Check if pcg return a NaN vector
127     if ~any(desc_dir)
128         % use default value
129         desc_dir = -grad_f_xk;
130     end
131 
132     % -- Backtracking --
133     alpha = 1; % this ensure quadratic convergence in the long run
134 
135     x_new = x_k + alpha*desc_dir;
136     f_new = f(x_new);
137 
138     b = 0;
139     while b < max_backtrack && ...
140             f_new > f_xk + alpha*c1*grad_f_xk'*desc_dir % Armijo
141         % Reduce alpha
142         alpha = alpha * rho;
143 
144         % Re-compute
145         x_new = x_k + alpha*desc_dir;
146         f_new = f(x_new);
```

```
147         b = b + 1;
148     end
149
150     % Check for backtracking failure
151     if b >= max_backtrack && ...
152         f_new > f_xk + alpha*c1*grad_f_xk'*desc_dir
153         failure = true;
154         flag = 'Failure: Could not satisfy Armijo';
155
156         % No need to store the x_new, since it's not an improvment
157         break
158     end
159
160     % Check for stagnation
161     improvment = f_xk/f_new - 1;
162     % improvment < 0 means no improvment
163
164     if improvment < stagn_threshold
165         stagnation = stagnation + 1;
166     else
167         stagnation = 0;
168     end
169
170     % -- Update --
171     x_k = x_new;
172     f_xk = f_new;
173     grad_f_xk = grad_f(x_new);
174     norm_grad_f_xk = norm(grad_f_xk);
175
176     i = i + 1;
177
178     x_sequence(:, i) = x_k;
179     backtrack_sequence(i) = b;
180     pcg_sequence(:, i) = [pcg_iter; pcg_flag; precond_type];
181 end
182
183 % -- Save result --
184 % Final solution
```

```
185 x_found = x_k;
186 f_x = f_xk;
187 norm_grad_f_x = norm_grad_f_xk;
188 iteration = i;
189
190 % Resize sequence variables
191 x_sequence = [x_initial, x_sequence(:, 1:iteration)]; % add starting value
192 backtrack_sequence = backtrack_sequence(1:iteration);
193 pcg_sequence = pcg_sequence(:, 1:iteration);
194
195 % Flag output
196 if ~failure
197     if norm_grad_f_x < tollerance
198         flag = sprintf('Satysfied the tollerance in %d iteration', iteration);
199     else
200         if stagnation >= max_stagnation
201             flag = sprintf(['Failure: Stagnation, after %d iterations. ' ...
202                 'Norm of the gradient = %.3g'], iteration, norm_grad_f_x);
203         else
204             flag = sprintf(['Failure: tnm did %d iteration but did not converge.
                ↪  ' ...
205                 'Norm of the gradient = %.3g'], iteration, norm_grad_f_x);
206         end
207
208         failure = true;
209     end
210 end
211 end
```

B.3 Functions and starting points

B.3.1 Problem82

```
function F = problem_82(x)
    %PROBLEM_82 Problem 82 function evaluation
    % Input:
    %   x : n-dimensional vector
    % Output:
    %   F : scalar function value

    n = length(x);
    cos_x = cos(x);

    % First term
    F = x(1)^2;
    for i = 2:n
        F = F + (cos_x(i-1) + x(i) - 1)^2;
    end

    % Divide by two
    F = 0.5*F;
end
```

```
function gradF = problem_82_grad(x)
    %PROBLEM_82_GRAD Gradient of the Problem 82 function
    % Input:
    %   x : n-dimensional vector
    % Output:
    %   gradF : n-dimensional gradient vector

    n = length(x);
    cos_x = cos(x);
    sin_x = sin(x);

```

```
    gradF = zeros(n, 1);
    gradF(1) = x(1) - sin_x(1)*(cos_x(1) + x(2) - 1);

    for i = 2:n-1
        gradF(i) = cos_x(i-1) + x(i) - 1 - sin_x(i)*(cos_x(i) + x(i+1) - 1);
    end

    gradF(n) = cos_x(n-1) + x(n) - 1;
end
```

```
function HessF = problem_82_hess(x)
    %PROBLEM_82_HESS Hessian of the Problem 82 function
    % Input:
    %   x : n-dimensional vector
    % Output:
    %   HessF : (n x n) Hessian matrix (tri-diagonal)

    n = length(x);
    cos_x = cos(x);
    sin_x_2 = sin(x).^2;

    main_diag = ones(n, 1);
    off_diag = -sin(x(1:n-1));

    for i = 1:n-1
        main_diag(i) = 1 - cos_x(i)*(cos_x(i) + x(i+1) - 1) + sin_x_2(i);
    end

    % Build sparse Hessian
    Bin = [[off_diag; 0], main_diag, [0; off_diag]];
    HessF = spdiags(Bin, [-1 0 1], n, n);
end
```

```
function Hess = problem_82_hess_approx(x_bar, h, specific, gradF, gradF_xbar)
    %PROBLEM_82_HESS_APPROX Approximation of the Jacobian of the Problem 82
    %function
    %   For specific tri-diagonal jacobian
    %
    %   Input:
    %       x_bar - point in where to compute the jacobian
    %           row vector
    %       h - level of approximation
    %           scalar
    %       specific - implement a specific approximation based on x_bar
    %           logical value
    %       gradF - gradient of the Problem 82 function
    %           function handle
    %       gradF_xbar - gradient at x_bar
    %           row vector
    %
    %   Output
    %       Hess - approximation of the hessian
    %           sparse matrix (tri-diagonal)

    % Problem dimension
    n = length(x_bar);

    % Gradient at x_bar
    if isempty(gradF_xbar)
        gradF_xbar = gradF(x_bar);
    end

    % Approximation step for each component of x
    if specific
        h_vec = h*abs(x_bar);
    else
        h_vec = h*ones(n, 1);
    end

    % Perturbation vectors
    e1 = zeros(n, 1);
```

```
    e2 = zeros(n, 1);
    e3 = zeros(n, 1);

    e1(1:3:n) = h_vec(1:3:n);
    e2(2:3:n) = h_vec(2:3:n);
    e3(3:3:n) = h_vec(3:3:n);

    % Compute hessian component
    approx = @(e) (gradF(x_bar + e) - gradF_xbar)./h_vec;
    eval = [approx(e1), approx(e2), approx(e3)];

    % Hessian indices for sparsity
    row_idx = (2:n)';
    col_idx = mod(row_idx - 1, 3) + 1;

    % Build hessian
    main_diag = [eval(1, 1); eval(sub2ind(size(eval), row_idx, col_idx))];
    off_diag = eval(sub2ind(size(eval), row_idx - 1, col_idx));

    % Build a sparse matrix
    Bin = [[off_diag; 0], main_diag, [0; off_diag]];
    Hess = spdiags(Bin, [-1 0 1], n, n);
end
```

B.3.2 Extended Rosenbrock

```
function F = extended_rosenbrock(x)
    % EXTENDED_ROSENBROCK Extended Rosenbrock function evaluation
    % Input:
    %   x : n-dimensional vector
    % Output:
    %   F : scalar function value

    % Dimension of input vector
    n = length(x);

    % Check that input dimension is even
    if mod(n, 2) ~= 0
        error('Input dimension n must be even for the Extended Rosenbrock
        ↪  function.');
    end

    % Initialize function value
    F = 0;

    % Compute function value
    for i = 1:2:n-1
        % Compute the two terms of the Rosenbrock function
        k = i; % (odd)
        f_odd = 10 * (x(k)^2 - x(k+1));

        k = i+1; % (even)
        f_even = (x(k - 1) - 1);

        % Accumulate the result
        F = F + f_odd^2 + f_even^2;
    end

    % Divide by two
    F = 0.5 * F;
end
```

```
function gradF = extended_rosenbrock_grad(x)
    % EXTENDED_ROSENBROCK_GRAD Gradient of the Extended Rosenbrock function
    % Input:
    %   x : n-dimensional vector
    % Output:
    %   gradF : n-dimensional gradient vector

    % Dimension of input vector
    n = length(x);

    % Check that input dimension is even
    if mod(n, 2) ~= 0
        error('Input dimension n must be even for the Gradient of the Extended
        ↪ Rosenbrock function.');
    end

    % Preallocate gradient vector
    gradF = zeros(n, 1);

    % Compute the gradient in pairs
    for i = 1:2:n-1

        % Gradient w.r.t. x_i (odd)
        k = i; % always odd
        gradF(k) = 200*x(k)^3 - 200*x(k)*x(k+1) + x(k) - 1;

        % Gradient w.r.t. x_{i+1} (even)
        k = i+1; % always even
        gradF(k) = 100*(x(k) - x(k-1)^2);
    end
end
```

```
function HessF = extended_rosenbrock_hess(x)
    % EXTENDED_ROSENBROCK_HESS Hessian of the Extended Rosenbrock function
    % Input:
    %   x : n-dimensional vector
    % Output:
    %   HessF : (n x n) Hessian matrix (symm. tri-diagonal)

    % Dimension of input vector
    n = length(x);

    % Check that input dimension is even
    if mod(n, 2) ~= 0
        error('Input dimension n must be even for the Hessian of the Extended
        ↪  Rosenbrock function.');
    end

    % Initialize the diagonals vectors
    main_diag = 100*ones(n, 1); % Note: Half of main diagonal entries them are 100
    ↪  by defintion (see later)
    off_diag = zeros(n - 1, 1); % Note: Half of off-diagonal entries are 0 by
    ↪  definition (see later)

    for i = 1:2:n-1

        k = i; % (odd)
        main_diag(k) = 600*x(k)^2 - 200*x(k+1) + 1;
        off_diag(k) = -200*x(k);

        % Values for k=i+1 are already set before loop starts
        % k = i+1; % (even)
        % main_diag(k) = 100;
        % off_diag(k)  = 0;
    end

    % Build a sparse matrix
    Bin = [[off_diag; 0], main_diag, [0; off_diag]];
    HessF = spdiags(Bin, [-1 0 1], n, n);
```

```
end
```

```
function Hess = extended_rosenbrock_hess_approx(x_bar, h, specific, gradF,
↪ gradF_xbar)
    %EXTENDED_ROSENBROCK_HESS_APPROX Approximation of the Jacobian of the Extended
    ↪ Rosenbrock function, for specific tri-diagonal jacobian
    %
    % Input:
    %     x_bar - point in where to compute the jacobian
    %         row vector
    %     h - level of approximation
    %         scalar
    %     specific - implement a specific approximation based on x_bar
    %         logical value
    %     gradF - gradient of the Rosenbrock function
    %         function handle
    %     gradF_xbar - gradient at x_bar
    %         row vector
    % Output
    %     Hess - approximation of the hessian
    %         sparse matrix (tri-diagonal)

    % Problem dimension
    n = length(x_bar);

    % Gradient at x_bar
    if isempty(gradF_xbar)
        gradF_xbar = gradF(x_bar);
    end

    % Approximation step for each component of x
    if specific
        h_vec = h*abs(x_bar);
    else
        h_vec = h*ones(n, 1);
```

```
    end

    % Perturbation vectors
    e1 = zeros(n, 1);
    e2 = zeros(n, 1);

    e1(1:2:n) = h_vec(1:2:n);
    e2(2:2:n) = h_vec(2:2:n);

    % Compute hessian component
    approx = @(e) (gradF(x_bar + e) - gradF_xbar)./h_vec;
    eval = [approx(e1), approx(e2)];

    % Compute Hessian indices
    row_idx = (2:n)';
    col_idx = mod(row_idx - 1, 2) + 1;

    % Build hessian
    main_diag = [eval(1, 1); eval(sub2ind(size(eval), row_idx, col_idx))];
    off_diag = eval(sub2ind(size(eval), row_idx - 1, col_idx));

    % Set to 0 all even elements of off_diag
    off_diag(2:2:n-1) = 0;

    % Build a sparse matrix
    Bin = [[off_diag; 0], main_diag, [0; off_diag]];
    Hess = spdiags(Bin, [-1 0 1], n, n);
end
```

B.3.3 Extended Powell badly scaled

```
function F = extended_powell(x)
    % EXTENDED_POWELL Extended Powell function evaluation
    % Input:
    %   x     : n-dimensional vector (n must be even)
    % Output:
    %   F : scalar function value

    % Badly scaling parameter
    alpha   = 1e4;
    beta    = 1;
    gamma   = 1 + 1e-4;

    % Dimension of input vector
    n = length(x);

    % Ensure input dimension is even
    if mod(n, 2) ~= 0
        error('Input dimension n must be even for the Extended Powel function.');
    end

    % Initialize function value
    F = 0;

    % Compute the function value
    for i = 1:2:n-1 % Compute entries at k=i and k=i+1 at the same time

        % k = i (odd)
        k = i;
        f_odd = alpha * x(k) * x(k+1) - beta;

        % k = i + 1 (even)
        k = i+1;
        f_even = exp(-x(k - 1)) + exp(-x(k)) - gamma;

        % Accumulate the result
        F = F + f_odd^2 + f_even^2;
```

```
    end

    % Divide by two
    F = 0.5 * F;
end
```

```
function gradF = extended_powell_grad(x)
    % EXTENDED_POWELL_GRAD Gradient of the Extended Powell function
    % Input:
    %   x     : n-dimensional vector (n must be even)
    % Output:
    %   gradF : n-dimensional gradient vector

    % Badly scaling parameter
    alpha   = 1e4;
    beta    = 1;
    gamma   = 1 + 1e-4;

    % Dimension of input vector
    n = length(x);

    % Ensure input dimension is even
    if mod(n, 2) ~= 0
        error('Input dimension n must be even for the Extended Powel function.');
    end

    % Initialize the gradient vector
    gradF = zeros(n, 1);

    % Compute the gradient
    for i = 1:2:n-1

        % Odd indices
        k = i;
        gradF(k) = x(k)*x(k+1)^2*alpha^2 - alpha*beta*x(k+1) - exp(-2*x(k)) -
        ↪  exp(-x(k)-x(k+1)) + gamma*exp(-x(k));
```

```
30
31          % Even indices
32          k = i+1;
33          gradF(k) = x(k)*x(k-1)^2*alpha^2 - alpha*beta*x(k-1) - exp(-2*x(k)) -
            ↪  exp(-x(k)-x(k-1)) + gamma*exp(-x(k));
34      end
35  end
```

```
1   function HessF = extended_powell_hess(x)
2       % EXTENDED_POWELL_HESS Hessian of the Extended Powell function
3       % Input:
4       %   x     : n-dimensional vector (n must be even)
5       % Output:
6       %   HessF : (n x n) Hessian matrix (symmetric tri-diagonal)
7
8       % Badly scaling parameter
9       alpha   = 1e4;
10      beta    = 1;
11      gamma   = 1 + 1e-4;
12
13      % Dimension of input vector
14      n = length(x);
15
16      % Ensure input dimension is even (useless, but provided for coherence)
17      if mod(n, 2) ~= 0
18          error('Input dimension n must be even for the Extended Powel function.');
19      end
20
21      % Initialize the gradient vector
22      main_diag = zeros(n, 1);
23      off_diag = zeros(n - 1, 1);
24
25      % Compute the gradient
26      for i = 1:2:n-1
27
```

```
        k = i;
        main_diag(k) = (alpha*x(k+1))^2 + 2*exp(-2*x(k)) +
        ↪  exp(-x(k))*(exp(-x(k+1)) - gamma);
        off_diag(k) = 2*x(k)*x(k+1)*alpha^2 - alpha*beta + exp(-x(k) - x(k+1));

        k = i+1;
        main_diag(k) = (alpha*x(k - 1))^2 + 2*exp(-2*x(k)) +
        ↪  exp(-x(k))*(exp(-x(k-1)) - gamma);
        % off_diag(k) = 0 % Skipped since sparsity storage will automaticcally
        ↪  fill
    end

    % Implement Hessian as sparse
    Bin = [[off_diag; 0], main_diag, [0; off_diag]];
    HessF = spdiags(Bin, [-1 0 1], n, n);
end
```

```
function Hess = extended_powell_hess_approx(x_bar, h, specific, gradF, gradF_xbar)
    %EXTENDED_POWELL_HESS_APPROX Approximation of the Jacobian of the Extended
    %Powell function
    %   For specific tri-diagonal jacobian
    %
    %   Input:
    %       x_bar - point in where to compute the jacobian
    %           row vector
    %       h - level of approximation
    %           scalar
    %       specific - implement a specific approximation based on x_bar
    %           logical value
    %       gradF - gradient of the Powell function
    %           function handle
    %       gradF_xbar - gradient at x_bar
    %           row vector
    %
    %   Output
```

```
%       Hess - approximation of the Hessian
%            sparse matrix (tri-diagonal)

% Problem dimension
n = length(x_bar);

% Gradient at x_bar
if isempty(gradF_xbar)
    gradF_xbar = gradF(x_bar);
end

% Approximation step for each component of x
if specific
    h_vec = h*abs(x_bar);
else
    h_vec = h*ones(n, 1);
end

% Perturbation vectors
e1 = zeros(n, 1);
e2 = zeros(n, 1);

e1(1:2:n) = h_vec(1:2:n);
e2(2:2:n) = h_vec(2:2:n);

% Compute Hessian component
approx = @(e) (gradF(x_bar + e) - gradF_xbar)./h_vec;
eval = [approx(e1), approx(e2)];

% Compute Hessian indices
row_idx = (2:n)';
col_idx = mod(row_idx - 1, 2) + 1;

% Build Hessian
main_diag = [eval(1, 1); eval(sub2ind(size(eval), row_idx, col_idx))];
off_diag = eval(sub2ind(size(eval), row_idx - 1, col_idx));

% Set to 0 all even elements of off_diag
```

```
    off_diag(2:2:n-1) = 0;

    % Build a sparse matrix
    Bin = [[off_diag; 0], main_diag, [0; off_diag]];
    Hess = spdiags(Bin, [-1 0 1], n, n);
end
```

B.3.4 Random Starting Points generation

```
clear;
clc;

% Random seed, from the student id
seed = min([318684, 337728, 338137]);
problem_dim = [1e3 1e4 1e5];

%% Problem 82
file_name = 'Problem_82.mat';

f = @problem_82;
grad_f = @problem_82_grad;
hess_f = @problem_82_hess;
hess_approx = @problem_82_hess_approx;

% Problem dimension -> 1000
x_0 = .5*ones(1000, 1);                    % starting point
min_1000 = zeros(1000, 1);                 % actual minima
x_1000 = [x_0, create_points(seed, x_0)];

% Problem dimension -> 10000
x_0 = .5*ones(10000, 1);                   % starting point
min_10000 = zeros(10000, 1);               % actual minima
x_10000 = [x_0, create_points(seed, x_0)];

% Problem dimension -> 100000
x_0 = .5*ones(100000, 1);                  % starting point
min_100000 = zeros(100000, 1);             % actual minima
x_100000 = [x_0, create_points(seed, x_0)];

save(file_name, "x_1000", "x_10000", "x_100000", "f", "grad_f", "hess_f",...
    "hess_approx", "min_1000", "min_10000", "min_100000");

%% Extended Rosenbrock
file_name = 'Ext_Rosenbrock.mat';

```

```
f = @extended_rosenbrock;
grad_f = @extended_rosenbrock_grad;
hess_f = @extended_rosenbrock_hess;
hess_approx = @extended_rosenbrock_hess_approx;

x_0 = [-1.2; 1];

% Problem dimension -> 1000
x_0 = repmat(x_0, 500, 1);                  % starting point
min_1000 = ones(1000, 1);                   % actual minima
x_1000 = [x_0, create_points(seed, x_0)];

% Problem dimension -> 10000
x_0 = repmat(x_0, 10, 1);                   % starting point
min_10000 = ones(10000, 1);                 % actual minima
x_10000 = [x_0, create_points(seed, x_0)];

% Problem dimension -> 100000
x_0 = repmat(x_0, 10, 1);                   % starting point
min_100000 = ones(100000, 1);               % actual minima
x_100000 = [x_0, create_points(seed, x_0)];

save(file_name, "x_1000", "x_10000", "x_100000", "f", "grad_f", "hess_f",...
    "hess_approx", "min_1000", "min_10000", "min_100000");

%% Extended Powell
file_name = 'Ext_Powell.mat';

f = @extended_powell;
grad_f = @extended_powell_grad;
hess_f = @extended_powell_hess;
hess_approx = @extended_powell_hess_approx;

x_0 = [0; 1];
minimum = [1.09815933e-5; 9.106146738];
% f(min) = 2.3e-21; norm of grad = 6.2e-6

% Problem dimension -> 1000
```

```
x_0 = repmat(x_0, 500, 1);                  % starting point
min_1000 = repmat(minimum, 500, 1);         % actual minima
x_1000 = [x_0, create_points(seed, x_0)];

% Problem dimension -> 10000
x_0 = repmat(x_0, 10, 1);                   % starting point
min_10000 = repmat(minimum, 5000, 1);       % actual minima
x_10000 = [x_0, create_points(seed, x_0)];

% Problem dimension -> 100000
x_0 = repmat(x_0, 10, 1);                   % starting point
min_100000 = repmat(minimum, 50000, 1);     % actual minima
x_100000 = [x_0, create_points(seed, x_0)];

save(file_name, "x_1000", "x_10000", "x_100000", "f", "grad_f", "hess_f",...
    "hess_approx", "min_1000", "min_10000", "min_100000");

function all_x = create_points(seed, x_0)
    %CREATE_POINTS
    % Creates 10 new starting points randomly generated with uniform
    % distribution in a hyper-cube centered at x_0

    rng(seed);

    n_points = 10;
    n = length(x_0);

    all_x = 2*rand(n, n_points) - 1; % Interval [-1, 1]
    all_x = all_x + x_0; % Interval [x_0 - 1,  x_0 + 1]
end
```

B.3.5 Testing functions

Modified Newton Method Testing

```
close all; clear; clc;

%% Add folders

addpath('test_problems_for_unconstrained_optimization\');
addpath("starting_points\");

%% Variables Initialization and tuning

% *** Function + starting points ***
% Choose among:
%   'Problem_82.mat'
%   'Ext_Rosenbrock.mat'
%   'Ext_Powell.mat'
% % %
load('Problem_82.mat');

% Outer loop
max_iterations = 5000;
tollerance = 1e-4;

% Backtracking
max_back_iterations = 50;
c1 = 1e-4;
rho = .5;

% PCG preconditioning
do_precondintioning = true;

% Hessian approximation
h_approximation = 1e-12;
specific_approx = true;

do_hess_approx = false;
```

```
35 if do_hess_approx
36     hess_f = [];
37 end
38
39 %  *** Correction tuning ***
40 % Problem 82:
41 % - 'minima': 1e-2<, 1e-1, 1 with success runs 0, 3, 33 (converges to 0).
42 % - 'diag':   1e-8
43 %
44 % Extended Rosenbrock:
45 % - 'minima': 1e-4<, 350, 400 with success runs 3, 8, 23 (converges to 0).
46 % - 'diag':   1e-8
47 % - NOTES:
48 %   - At 350, 3 standard + 5 points from n=1e3 converge.
49 %   - At 375, everything works for n=1e3, but only for some n=1e4. Unknown for
   ↪  n=1e5 (too long).
50 %   - Long runtime for some points (x_6, x_7, x_9, x_10, x_11) at 375.
51 %
52 % Extended Powell:
53 % - 'minima': 1e-8< with success runs 3 (converges to 0).
54 % - 'diag':   1e-8
55
56 correction_method = 'minima'; % Use one among 'minima', 'diag'
57 correction_parameters = 400;
58
59 fprintf("*** USING CORRECTION METHOD: %s WITH TOLLERANCE %.1e *** \n\n",
   ↪  correction_method, correction_parameters);
60
61 %% Choose points to analyze and plots to display
62
63 % Set which point you want to analyze
64 %
65 %       1 --> Default starting point (improved convergence)
66 % [2, 11] --> correspondent to the 10 points randomly generated
67 %
68 % Multiple points can be run, just by setting variable
69 % 'point' as a list that ranges between integers
70 %   e.g. 1:4    will run the default point + first 3 randomly generated
```

```
%       2:4    will run first 3 randomly generated
%       1:11   will run all point (defualt + the 10 randomly generated)
point = 1:11;

% Stats
tot_success = 3*length(point);

% Plots to display
plot_rate_convergence       = true;
plot_matrix_corrections     = true;
plot_function_convergence   = true;
plot_gradient_convergence   = true;

%% Dimension 1000 (1e3)

for i = point
    x_0 = x_1000(:, i);

    fprintf("PROBLEM DIMENSION: %.1e\n", length(x_0));
    fprintf("x_%d = ", i-1); print_summary(x_0);

    tic;
    [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
        x_sequence, backtrack_sequence, corr_sequence, fseq, gradnormseq] = ...
    modifiedNM(f, grad_f, hess_f, x_0, max_iterations, ...
        tollerance, c1, rho, max_back_iterations, do_precondintioning, ...
        h_approximation, specific_approx, hess_approx, correction_method,
        ↪  correction_parameters);
    execution_time = toc;

    % Output
    print_output(flag, x_found, f_x, norm_grad_f_x, iteration, ...
        max_iterations, execution_time);

    % Plot order of convergence, corrections, function value, and gradient
    ↪  convergence
    if iteration > 1
        if plot_rate_convergence == true
```

```
107             rate_convergence(iteration, min_1000, x_sequence, i);
108         end
109         if plot_matrix_corrections == true
110             matrix_corrections(corr_sequence, length(x_0));
111         end
112         if plot_function_convergence == true
113             function_convergence(fseq, length(x_0));
114         end
115         if plot_gradient_convergence == true
116             gradient_convergence(gradnormseq, length(x_0));
117         end
118     end
119
120     % Stats
121     tot_success = tot_success - failure;
122
123     fprintf("\n\n");
124 end
125
126 %% Dimension 10000 (1e4)
127
128 for i = point
129     x_0 = x_10000(:, i);
130
131     fprintf("PROBLEM DIMENSION: %.1e\n", length(x_0));
132     fprintf("x_%d = ", i-1); print_summary(x_0);
133
134     tic;
135     [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
136         x_sequence, backtrack_sequence, corr_sequence, fseq, gradnormseq] = ...
137     modifiedNM(f, grad_f, hess_f, x_0, max_iterations, ...
138         tollerance, c1, rho, max_back_iterations, do_precondintioning, ...
139         h_approximation, specific_approx, hess_approx, correction_method,
        ↪  correction_parameters);
140     execution_time = toc;
141
142     % Output
143     print_output(flag, x_found, f_x, norm_grad_f_x, iteration, ...
```

```
144         max_iterations, execution_time);
145
146     % Plot order of convergence, corrections, function value, and gradient
        ↪ convergence
147     if iteration > 1
148         if plot_rate_convergence == true
149             rate_convergence(iteration, min_10000, x_sequence, i);
150         end
151         if plot_matrix_corrections == true
152             matrix_corrections(corr_sequence, length(x_0));
153         end
154         if plot_function_convergence == true
155             function_convergence(fseq, length(x_0));
156         end
157         if plot_gradient_convergence == true
158             gradient_convergence(gradnormseq, length(x_0));
159         end
160     end
161
162     % Stats
163     tot_success = tot_success - failure;
164
165     fprintf("\n\n");
166 end
167
168 %% Dimension 100000 (1e5)
169
170 for i = point
171     x_0 = x_100000(:, i);
172
173     fprintf("PROBLEM DIMENSION: %.1e\n", length(x_0));
174     fprintf("x_%d = ", i-1); print_summary(x_0);
175
176     tic;
177     [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
178         x_sequence, backtrack_sequence, corr_sequence, fseq, gradnormseq] = ...
179     modifiedNM(f, grad_f, hess_f, x_0, max_iterations, ...
180         tollerance, c1, rho, max_back_iterations, do_precondintioning, ...
```

```
181         h_approximation, specific_approx, hess_approx, correction_method,
            ↪ correction_parameters);
182     execution_time = toc;
183
184     % Output
185     print_output(flag, x_found, f_x, norm_grad_f_x, iteration, ...
186         max_iterations, execution_time);
187
188     % Plot order of convergence, corrections, function value, and gradient
        ↪ convergence
189     if iteration > 1
190         if plot_rate_convergence == true
191             rate_convergence(iteration, min_100000, x_sequence, i);
192         end
193         if plot_matrix_corrections == true
194             matrix_corrections(corr_sequence, length(x_0));
195         end
196         if plot_function_convergence == true
197             function_convergence(fseq, length(x_0));
198         end
199         if plot_gradient_convergence == true
200             gradient_convergence(gradnormseq, length(x_0));
201         end
202     end
203
204     % Stats
205     tot_success = tot_success - failure;
206
207     fprintf("\n\n");
208 end
209
210 %% Print count of successful runs
211
212 fprintf("Successful run: %d/%d\n", tot_success, 3*length(point));
213
214 %% Plotting and Analysis Functions
215
216 function print_output(flag, sol, f_x, norm_grad, iters, max_iters, time)
```

```
217     fprintf("FLAG: %s\n", flag);
218     fprintf("SOLUTION FOUND = "); print_summary(sol);
219     fprintf("FUNCTION VALUE = %.3g\n" ,f_x);
220     fprintf("NORM OF THE GRADIENT = %.3g\n", norm_grad);
221     fprintf("ITERATIONS PERFORMED: %d/%d\n", iters, max_iters);
222     fprintf("EXECUTION TIME: %f s\n", time);
223 end
224 
225 function print_summary(vector)
226     fprintf("[");
227     fprintf(" %.3f ", vector(1:4));
228     fprintf("...");
229     fprintf(" %.3f ", vector(end-3:end));
230     fprintf("]\n");
231 end
232 
233 function rate_convergence(iterations, actual_minimum, sequence, i)
234     % Compute errors
235     error = vecnorm(sequence - actual_minimum);
236 
237     % Initialize convergence metrics
238     sup_linear = zeros(1, iterations);
239     quadratic = zeros(1, iterations);
240 
241     % Calculate rates
242     for err = 1:iterations
243         sup_linear(err) = error(err + 1)/error(err);
244         quadratic(err) = error(err + 1)/(error(err)^2);
245     end
246 
247     % Create tiled layout and plots
248     figure('Name', ['x_', num2str(i), ' (Dim = ', ...
249         num2str(length(actual_minimum)), '): Order of convergence'], ...
250         'NumberTitle', 'off');
251     t = tiledlayout(1, 2); % Define tiled layout
252 
253     % Plot Linear/Superlinear Convergence
254     ax1 = nexttile;
```

```
255     plot(1:iterations, sup_linear, '-r');
256     title(ax1, 'Linear/Superlinear Convergence'); % Assign title to the correct
        ↪ axis
257     xlabel(ax1, 'Iteration');
258     ylabel(ax1, 'Rate');
259
260     % Plot Quadratic Convergence
261     ax2 = nexttile;
262     plot(1:iterations, quadratic, '-r');
263     title(ax2, 'Quadratic Convergence'); % Assign title to the correct axis
264     xlabel(ax2, 'Iteration');
265     ylabel(ax2, 'Rate');
266
267     % Add a global title for the tiled layout
268     sgtitle(t, ['Convergence Analysis: x_', num2str(i)]);
269 end
270
271
272 function matrix_corrections(correction_sequence, prob_size)
273     figure('Name', sprintf('Matrix Corrections Applied, n = %1.e', prob_size),
        ↪ 'NumberTitle', 'off');
274     if length(find(correction_sequence ~= 0)) >= 100
275         scatter(find(correction_sequence ~= 0),
            ↪ correction_sequence(correction_sequence ~= 0), 'filled', 'LineWidth',
            ↪ 1.5);
276     else
277         stem(find(correction_sequence ~= 0),
            ↪ correction_sequence(correction_sequence ~= 0), 'filled', 'LineWidth',
            ↪ 1.5);
278     end
279     xlabel('Iteration');
280     ylabel('Correction Value');
281     title(sprintf('Matrix Corrections Applied, n = %1.e', prob_size));
282     grid on;
283 end
284
285 function function_convergence(function_values, prob_size)
286     figure('Name', sprintf('Function Value Convergence, n = %1.e', prob_size),
        ↪ 'NumberTitle', 'off');
```

```
287     semilogy(1:length(function_values), function_values, 'LineWidth', 2);
288     xlabel('Iteration');
289     ylabel('Function Value');
290     title(sprintf('Function Value Convergence, n = %1.e', prob_size));
291     grid on;
292 end
293 
294 function gradient_convergence(gradient_norms, prob_size)
295     figure('Name', sprintf('Gradient Norm Convergence, n = %1.e', prob_size),
        ↪  'NumberTitle', 'off');
296     semilogy(1:length(gradient_norms), gradient_norms, 'LineWidth', 2);
297     xlabel('Iteration');
298     ylabel('Gradient Norm (log scale)');
299     title(sprintf('Gradient Norm Convergence, n = %1.e', prob_size));
300     grid on;
301 end
```

Truncated Newton Method Testing

```
close all; clear; clc;

%% Add folders

addpath('test_problems_for_unconstrained_optimization\');
addpath("starting_points\");

%% Variables Initialization and tuning

% *** Function + starting points ***
% Choose among:
%   'Problem_82.mat'
%   'Ext_Rosenbrock.mat'
%   'Ext_Powell.mat'
% % %
load('Problem_82.mat');

% Outer loop
max_iterations = 5000;
tollerance = 1e-6;

% Backtracking
max_back_iterations = 50;
c1 = 1e-4;
rho = .5;

% PCG preconditioning
do_precondintioning = false;

% Hessian approximation
h_approximation = 1e-12;
specific_approx = false;

do_hess_approx = false;
if do_hess_approx
    hess_f = [];
```

```
37  end
38
39  %% Choose points to analyze and plots to display
40
41  % Set which point you want to analyze
42  %
43  %       1 --> Default starting point (improved convergence)
44  % [2, 11] --> correspondent to the 10 points randomly generated
45  %
46  % Multiple points can be run, just by setting variable
47  %  'point' as a list that ranges between integers
48  %   e.g. 1:4 will run the default point + first 3 randomly generated
49  %       2:4 will run first 3 randomly generated
50  %       1:11 will run all point (defualt + the 10 randomly generated)
51  point = 1:4;
52
53  % Stats
54  tot_success = 3*length(point);
55
56  % Plots to display
57  plot_rate_convergence = false;
58
59  %% Dimension 1000
60  for i = point
61      x_0 = x_1000(:, i);
62
63      fprintf("PROBLEM DIMENSION: %d\n", length(x_0));
64      fprintf("x_%d = ", i-1); print_summary(x_0);
65
66      tic;
67      [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
68          x_sequence, backtrack_sequence, pcg_sequence] = ...
69      truncatedNM(f, grad_f, hess_f, x_0, max_iterations, ...
70          tollerance, c1, rho, max_back_iterations, do_precondintioning, ...
71          h_approximation, specific_approx, hess_approx);
72      execution_time = toc;
73
74      % Output
```

```
75      print_output(flag, x_found, f_x, norm_grad_f_x, iteration, ...
76          max_iterations, execution_time);
77
78      % Order of convergence
79      if iteration > 1
80          if rate_convergence
81              rate_convergence(iteration, min_1000, x_sequence, i);
82          end
83      end
84
85      % Stats
86      tot_success = tot_success - failure;
87
88      fprintf("\n\n");
89  end
90
91  %% Dimension 10000
92  for i = point
93      x_0 = x_10000(:, i);
94
95      fprintf("PROBLEM DIMENSION: %d\n", length(x_0));
96      fprintf("x_%d = ", i-1); print_summary(x_0);
97
98      tic;
99      [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
100         x_sequence, backtrack_sequence, pcg_sequence] = ...
101     truncatedNM(f, grad_f, hess_f, x_0, max_iterations, ...
102         tollerance, c1, rho, max_back_iterations, do_precondintioning, ...
103         h_approximation, specific_approx, hess_approx);
104     execution_time = toc;
105
106     % Output
107     print_output(flag, x_found, f_x, norm_grad_f_x, iteration, ...
108         max_iterations, execution_time);
109
110     % Order of convergence
111     if iteration > 1
112         if rate_convergence
```

```
113             rate_convergence(iteration, min_10000, x_sequence, i);
114         end
115     end
116
117     % Stats
118     tot_success = tot_success - failure;
119
120     fprintf("\n\n");
121 end
122
123 %% Dimension 100000
124 for i = point
125     x_0 = x_100000(:, i);
126
127     fprintf("PROBLEM DIMENSION: %d\n", length(x_0));
128     fprintf("x_%d = ", i-1); print_summary(x_0);
129
130     tic;
131     [x_found, f_x, norm_grad_f_x, iteration, failure, flag, ...
132         x_sequence, backtrack_sequence, pcg_sequence] = ...
133     truncatedNM(f, grad_f, hess_f, x_0, max_iterations, ...
134         tollerance, c1, rho, max_back_iterations, do_precondintioning, ...
135         h_approximation, specific_approx, hess_approx);
136     execution_time = toc;
137
138     % Output
139     print_output(flag, x_found, f_x, norm_grad_f_x, iteration, ...
140         max_iterations, execution_time);
141
142     % Order of convergence
143     if iteration > 1
144         if rate_convergence
145             rate_convergence(iteration, min_100000, x_sequence, i);
146         end
147     end
148
149     % Stats
150     tot_success = tot_success - failure;
```

```
151
152         fprintf("\n\n");
153     end
154
155     fprintf("Successful run: %d/%d\n", tot_success, 3*length(point));
156
157     function print_output(flag, sol, f_x, norm_grad, iters, max_iters, time)
158         fprintf("FLAG: %s\n", flag);
159         fprintf("SOLUTION FOUND = "); print_summary(sol);
160         fprintf("FUNCTION VALUE = %.3g\n" ,f_x);
161         fprintf("NORM OF THE GRADIENT = %.3g\n", norm_grad);
162         fprintf("ITERATIONS PERFORMED: %d/%d\n", iters, max_iters);
163         fprintf("EXECUTION TIME: %f s\n", time);
164     end
165
166     function print_summary(vector)
167         fprintf("[");
168         fprintf(" %.3f ", vector(1:4));
169         fprintf("...");
170         fprintf(" %.3f ", vector(end-3:end));
171         fprintf("]\n");
172     end
173
174     function rate_convergence(iterations, actual_minimum, sequence, i)
175         error = vecnorm(sequence - actual_minimum);
176
177         sup_linear = zeros(1, iterations);
178         quadratic = zeros(1, iterations);
179
180         for err = 1:iterations
181             sup_linear(err) = error(err + 1)/error(err);
182             quadratic(err) = error(err + 1)/(error(err)^2);
183         end
184
185         % Ratio Plots
186         figure('Name', ['x_', num2str(i), ' (Dim = ', ...
187             num2str(length(actual_minimum)), '): Order of convergence'], ...
188             'NumberTitle','off');
```

```
    tiledlayout(1, 2);

    nexttile
    title('Linear/Superlinear convergence');
    plot(1:iterations, sup_linear, '-r', ...
        1:iterations, zeros(1, iterations), '.-b');

    nexttile
    title('Quadratic convergence');
    plot(1:iterations, quadratic, '-r');
end
```

References

[1] Ladislav Lukšan and Jan Vlcek. Test problems for unconstrained optimization. *Academy of Sciences of the Czech Republic, Institute of Computer Science, Technical Report*, (897), 2003.

[2] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.