Politecnico di Torino

POLITECNICO DI TORINO

Corso di laurea in
Ingegneria Matematica

Numerical optimization for large scale problems and Stochastic Optimization - Assignment on Unconstrained Optimization

Elisabetta Roviera s328422

21th Febbrary 2025

Contents

1 Introduction

This document presents the implementation and analysis of two numerical methods for unconstrained optimisation: the Modified Newton Method and the Truncated Newton Method. These methods, which are variations of the classical Newton Method, aim to minimise functions of the form

$$\min_x f(x),$$

where x is a real-valued variable and f is a real-valued function. The study adheres to the guidelines stipulated in the project assignment, which necessitates the implementation of two optimisation techniques integrated with a backtracking line search strategy to ensure sufficient decrease conditions. The implementation has been conducted in MATLAB, and the performance of these methods has been evaluated on four distinct optimisation problems, encompassing the well-known Rosenbrock function and three additional problems derived from reference datasets [1].

A pivotal element of this study is the assessment of the methods' efficiency and accuracy under varied conditions. The Rosenbrock function is utilised as an initial test case, with specified starting points, to observe the convergence behaviour. The remaining test problems are studied in higher dimensions with random initial conditions to further assess the robustness of the system. Additionally, experiments have been conducted to compare the use of exact derivatives versus numerical approximations obtained via finite differences. The accuracy of finite difference approximations is explored by varying the step size parameters across multiple values to understand their impact on convergence.

Following the execution of the optimisation methods in accordance with the prescribed experimental setup, a comprehensive comparison has been conducted. The evaluation criteria encompass the number of successful runs, the iterations required to meet the stopping criterion, the experimental rate of convergence, the execution time, and other pertinent performance indicators. The results are presented in a structured manner using tabular and graphical representations, followed by an analytical discussion that contextualises the findings in relation to theoretical expectations. The entire MATLAB code used to generate the results is available in the appendix.

2 Description of the Methods

Newton's method, also referred to as the Newton-Raphson method, is a fundamental iterative technique used to find successively better approximations to the roots (or zeroes) of a real-valued function. In the context of optimisation, it is employed to locate stationary points where the gradient of the objective function vanishes, indicating potential minima, maxima, or saddle points. The method utilises the first and second derivatives of the function to inform its iterations, providing a more direct route to the solution compared to first-order methods like gradient descent.

The primary objective of Newton's method in optimization is to solve for x in $\nabla f(x) = 0$, where f is a twice-differentiable function mapping $\mathbb{R}^n$ to $\mathbb{R}$, and $\nabla f(x)$ denotes the gradient vector of f at x. The iterative update rule is given by:

$$x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$$

Here, $\nabla^2 f(x_k)$ represents the Hessian matrix of second-order partial derivatives at x_k. This update formula adjusts the current estimate x_k by the product of the inverse Hessian and the gradient, effectively utilizing curvature information to inform the step direction and magnitude.

A notable strength of Newton's method is its convergence rate, which, under appropriate conditions, exhibits quadratic convergence. That is to say, when the initial guess is sufficiently close to a solution and the Hessian is positive definite, the method demonstrates a decrease in error between the iterates and the actual solution that is proportional to the square of the previous error. This results in rapid convergence in the vicinity of the solution. However, it should be noted that this rapid convergence is local; the method may not perform well if the initial guess is far from the optimal point or if the Hessian is not positive definite. Despite its advantages, Newton's method has limitations. Computing the inverse of the Hessian matrix can be computationally expensive, especially for high-dimensional problems, as it

requires operations of order $O(n^3)$. Additionally, if the Hessian is singular or nearly singular, the method can encounter difficulties. To address these challenges, alternative methods such as the Damped Newton Method have been proposed. This method incorporates a step size parameter, ensuring global convergence and enhancing stability. Furthermore, quasi-Newton methods have been developed to approximate the Hessian, thereby reducing the computational burden while maintaining superlinear convergence rates. The methods implemented are variations of the Newton method, which is based on the approximation of the function f at each iteration k using a second-order Taylor expansion around the point $x^{(k+1)} = x^{(k)} + p$:

$$f(x^{(k)} + p) \approx f(x^{(k)}) + \nabla f(x^{(k)})^T p + \frac{1}{2} p^T \nabla^2 f(x^{(k)}) p.$$

The descent direction $p^{(k)}$ is then obtained by solving the linear system:

$$\nabla^2 f(x^{(k)}) p = -\nabla f(x^{(k)}).$$

It is evident that the direction p_k is a descent direction at iteration k only if the Hessian $Hess(f(x_k))$ is positive definite. In the event that this is not the case, the obtained direction may not be the correct one, requiring corrections. In such cases, the Modified Newton Method or the Truncated Newton Method may be employed. The convergence rate of the Truncated Newton and Modified Newton methods varies depending on the problem and specific implementation. Under certain assumptions, the pure Newton method exhibits quadratic convergence to the solution, and in the best-case scenario, both the modified and truncated Newton methods can also achieve quadratic convergence. However, due to the use of inexact line search techniques, convergence often results in only linear progress. Moreover, since the minima of the functions are not known except for the first iteration, an approximation of the true error at step k is given by:

$$e_k \approx \|x_k - x_{k-1}\|_2.$$

To compute the convergence rate, the following formula is used:

$$p \approx \frac{\log\left(\frac{\|e_{k+1}\|}{\|e_k\|}\right)}{\log\left(\frac{\|e_k\|}{\|e_{k-1}\|}\right)}.$$

In the ensuing sections, a concise discourse on the Modified and Truncated Newton methods will be explained.

2.1 Modified Newton Method

The Modified Newton Method is a variant of the classical Newton method that addresses one of its primary weaknesses: the possible indefiniteness of the Hessian matrix $\nabla^2 f(x)$. In cases where $\nabla^2 f(x)$ is not positive definite, a correction is applied to ensure that the modified Hessian is sufficiently positive definite. This is achieved by adding a corrective matrix E_k, resulting in a new matrix

$$B_k = \nabla^2 f(x_k) + E_k,$$

which is designed to have all its eigenvalues bounded away from zero by a positive constant. This modification ensures that the linear system used to compute the Newton direction is well-posed and that the search direction is indeed a descent direction.

Hessian Modification. At each iteration k, the method examines the Hessian $\nabla^2 f(x_k)$ and determines whether it is sufficiently positive definite. The goal is to enforce

$$\lambda_{\min}(B_k) \geq \delta > 0,$$

where $\lambda_{\min}(B_k)$ denotes the minimum eigenvalue of B_k, and δ is a predefined tolerance. A common and effective choice for the correction is to set

$$E_k = \tau_k I,$$

with I being the identity matrix and τ_k a nonnegative scalar that is adjusted adaptively. If $\nabla^2 f(x_k)$ is already positive definite, one may choose $\tau_k = 0$. Otherwise, τ_k is increased until B_k meets the condition for positive definiteness. Since computing the exact eigenvalues of B_k can be expensive, a bound such as

$$\lambda_i(B_k) \leq \|B_k\|_F,$$

where $\|B_k\|_F$ denotes the Frobenius norm, is sometimes used to guide the selection of τ_k.

Algorithm Description. Given an initial point x_0, the Modified Newton Method proceeds iteratively as follows:

Iteration 1: Hessian Modification and Factorization. For the current iterate x_k, form the modified Hessian

$$B_k = \nabla^2 f(x_k) + \tau_k I.$$

The parameter τ_k is chosen based on a preliminary check. For instance, one may set an initial value τ_0 as follows:

- Compute a measure β, e.g., the Frobenius norm of $\nabla^2 f(x_k)$.
- If the minimum of the diagonal entries of $\nabla^2 f(x_k)$ is positive, set $\tau_0 = 0$.
- Otherwise, set $\tau_0 = \beta/2$.

An attempt is then made to perform an (incomplete) Cholesky factorization on B_k. If the factorization is successful, the resulting factor (typically a lower triangular matrix L) is used to solve the linear system. If the factorization fails (indicating that B_k is not yet sufficiently positive definite), τ_k is updated, for example by

$$\tau_{k+1} = \max(2\tau_k, \beta/2),$$

and the process is repeated until a successful factorization is achieved.

Iteration 2: Computing the Search Direction. Once B_k has been factorized, the search direction p_k is obtained by solving the linear system

$$B_k p_k = -\nabla f(x_k).$$

The use of a direct method, such as Cholesky factorization, ensures that the search direction is computed reliably, given that B_k is now well-conditioned.

Iteration 3: Step Length Determination. A step length α_k is then determined using a line search procedure that satisfies conditions like the Wolfe, Goldstein, or Armijo backtracking conditions. This step is crucial to ensure a sufficient decrease in the objective function and to maintain the overall convergence properties of the method.

Iteration 4: Update. The next iterate is computed as

$$x_{k+1} = x_k + \alpha_k p_k.$$

The process continues until a convergence criterion is met, for example when $\|\nabla f(x_k)\|$ falls below a specified tolerance.

Practical Considerations. The success of the Modified Newton Method depends on the proper selection and adjustment of τ_k. While a simple heuristic such as doubling τ_k when the factorization fails often works well, more sophisticated strategies can be employed based on the structure of the problem. Additionally, if the problem size is very large or if a direct factorization is computationally prohibitive, one may consider using iterative solvers within the modified framework. In such cases, the ideas from the Truncated Newton Method can be integrated to combine the benefits of both strategies.

Discussion and Advantages. The Modified Newton Method offers several advantages:

- **Ensured Descent Direction:** By modifying the Hessian to be sufficiently positive definite, the method guarantees that the computed search direction is a descent direction, thus enhancing the robustness of the algorithm.
- **Adaptive Correction:** The adaptive strategy for choosing τ_k allows the algorithm to automatically adjust to the local curvature of the function. When the Hessian is nearly positive definite, only a small correction is needed, preserving the fast convergence properties of the Newton method.
- **Flexibility:** The approach can be integrated with various line search strategies and can also be combined with iterative methods for solving the linear system if needed.

Overall, the Modified Newton Method is a powerful tool in optimization, particularly in scenarios where the Hessian may be indefinite or poorly conditioned. Its ability to enforce positive definiteness and its flexibility in implementation make it well-suited for a broad range of nonlinear optimization problems.

2.2 Truncated Newton Method

The Truncated Newton Method (also known as the Newton-CG method) is an iterative optimization technique used to solve large-scale nonlinear optimization problems. It is particularly effective when the Hessian matrix is too large to factorize or store explicitly. Instead of computing the full Newton step, the method approximates the solution of the Newton system by using an iterative procedure based on the Conjugate Gradient (CG) method.

At each iteration k, given the current iterate x_k, the method seeks a search direction p_k by approximately solving the Newton equation

$$\nabla^2 f(x_k) p_k = -\nabla f(x_k),$$

where $\nabla f(x_k)$ is the gradient and $\nabla^2 f(x_k)$ is the Hessian of the objective function f at x_k. The approximation is achieved by running the CG method on the linear system defined by $\nabla^2 f(x_k)$ and $-\nabla f(x_k)$.

Conjugate Gradient Iteration. The CG method is designed for solving symmetric positive definite systems. However, in the context of the Truncated Newton Method, the Hessian $\nabla^2 f(x_k)$ may be indefinite. To account for this, the CG iterations are modified as follows:

Step 1: Initialization. Set the initial guess for the CG iteration to $x^{(0)} = 0$. This implies that the starting search direction is computed from the zero vector.

Step 2: Iteration and Negative Curvature Test. During the CG iterations, for each generated search direction $p^{(i)}$, a negative curvature test is performed. Specifically, if

$$\left(p^{(i)}\right)^T \nabla^2 f(x_k) p^{(i)} \leq 0,$$

then a direction of negative curvature has been encountered. The algorithm terminates the CG process immediately:

- If this occurs during the first CG iteration, a new iterate x_{k+1} is computed directly and the process for the current k stops.
- Otherwise, if it occurs in a subsequent iteration, the most recently generated approximate solution $x^{(i)}$ is accepted as the search direction.

Step 3: Termination Based on Residual Norm. In addition to the negative curvature condition, the CG iterations are terminated when the norm of the residual $r^{(i)}$ satisfies

$$\|r^{(i)}\| \leq \min\{0.5, \|\nabla f(x_k)\|\}\|\nabla f(x_k)\|.$$

This ensures that the approximate solution is sufficiently accurate relative to the current gradient norm.

Step 4: Defining the Newton Step. Once the CG iterations have been terminated (either by meeting the residual norm condition or encountering negative curvature), the final CG iterate, denoted by $x^{(f)}$, is set as the approximate Newton step:

$$p_k := x^{(f)}.$$

Preconditioning. Preconditioning may be applied within the CG iteration to improve convergence. A suitable preconditioner can transform the Hessian system into one with more favorable spectral properties, thereby accelerating the convergence of the CG method. The choice of preconditioner is problem-specific and can significantly impact performance.

Global Algorithm Description. Given an initial point x_0, the overall Truncated Newton Method can be summarized as follows:

Iteration 1: Search Direction Computation. For the current iterate x_k, compute the search direction p_k by approximately solving the linear system

$$\nabla^2 f(x_k)p = -\nabla f(x_k)$$

using the CG method with an initial guess $x^{(0)} = 0$. During the CG process, terminate the iterations when either:

- The residual norm satisfies

$$\|r^{(i)}\| \leq \min\{0.5, \|\nabla f(x_k)\|\}\|\nabla f(x_k)\|,$$

  or

- A direction of negative curvature is encountered, i.e.,

$$\left(p^{(i)}\right)^T \nabla^2 f(x_k) p^{(i)} \leq 0.$$

Iteration 2: Step Length Determination. Compute a step length α_k by employing a line search strategy that satisfies conditions such as the Wolfe, Goldstein, or Armijo backtracking conditions. This ensures that the new iterate achieves a sufficient decrease in the objective function.

Iteration 3: Update. Set the next iterate as

$$x_{k+1} = x_k + \alpha_k p_k.$$

Iteration 4: Convergence Check. Evaluate a convergence criterion (e.g., the norm of the gradient falling below a prescribed tolerance). If the criterion is not met, increment k and repeat the process.

Discussion and Advantages. The truncated Newton method is advantageous for large-scale optimization because:

- It avoids the explicit formation and factorization of the Hessian matrix, thus reducing computational and storage costs.
- The use of the CG method allows for exploiting the sparsity or structure of the Hessian, which is common in many practical applications.
- The negative curvature test provides a mechanism to handle indefinite Hessians by identifying directions that are not descent directions.

By truncating the CG iterations either upon reaching an acceptable residual norm or encountering negative curvature, the method maintains a balance between computational efficiency and the accuracy of the approximate Newton step. This makes the truncated Newton method a robust and practical choice for solving challenging nonlinear optimization problems.

3 Results and comparison between methods

This chapter presents an analysis of the results obtained from the optimization of four distinct functions: the 2-dimensional Rosenbrock function, the Banded trigonometric function with dimensions $n = 10^3$, 10^4, and 10^5, the Penalty function with dimensions $n = 10^3$, 10^4, and 10^5, and function of Problem 76 with dimensions $n = 10^3$, 10^4, and 10^5.

The analysis for each function follows a consistent structure:

1. **Function definition.** A formal definition of the function is provided.
2. **Initial point definition.** A set of initial points is defined (5 for the Rosenbrock function and 11 for the remaining functions).
3. **3-dimensional plot.** A 3-dimensional visualization of the function's landscape is presented.
4. **Grid search.** A grid search is performed to optimize the parameters c_1, ρ, and $btmax$.
5. **Tabulated results.** The optimization results for the modified Newton and truncated Newton methods, with and without preconditioning, are presented in tables with accompanying commentary.
6. **Finite difference approach.** An approach employing finite differences is utilized, including a grid search for parameter optimization and tabulated results following the same format as before.

3.1 Rosenbrook Function

The **function** under consideration is defined as:

$$F(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2.$$

A visualization of the function is shown in Figure 1. The optimization methods were tested using the following **initial conditions**:

- $x1 = [1.2; 1.2]$;
- $x2 = [-1.2; 1]$;
- $x3 = [0; 0]$;
- $x4 = [-1; 1]$;
- $x5 = [-2; 1.5]$;

A **grid search** was implemented to optimise the hyperparameters of the optimisation algorithms *Modified Newton*, *Truncated Newton* and *Truncated Preconditioned Newton*. This procedure involved the exhaustive evaluation of several combinations of the following hyper-parameters, with their respective numerical values:

- $c_1 \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$
- $\rho \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$
- $btmax \in \{30, 40, 50, 60, 70, 80, 90, 100\}$

The grid search was conducted by systematically iterating through all possible combinations of these values, and for each combination the performance of the algorithms was measured in terms of the average number of iterations required for convergence, calculated on different initial conditions. Although grid search is a methodologically simple approach for the optimisation of hyperparameters, and ensures that all combinations defined within the specified grid are explored, it has significant inherent limitations. The main disadvantage lies in the high computational cost. Moreover, the parameters identified as 'best' by grid search are only optimal in the context of the discrete grid explored, and may not represent the most efficient configuration in absolute terms. The results obtained are as follows:
`Grid search result - Best combination: c1 = 0.0001, rho = 0.8, btmax = 30`.

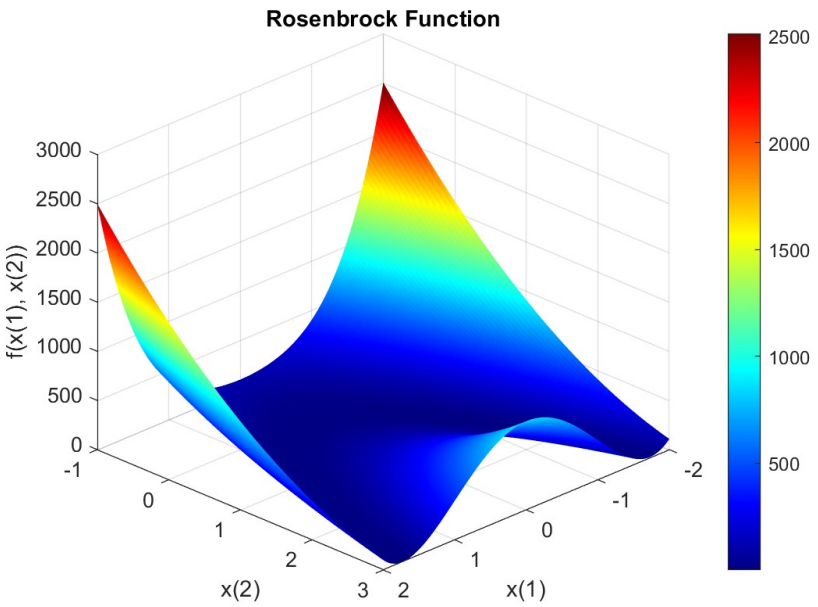


Figure 1: Plot of the Rosenbrock function

Below, we analyze the results based on the initial conditions. For all methods, convergence was successfully reached for all initial points, as indicated in the last column of each table. However, the efficiency of the methods varies in terms of the number of iterations, execution time, and the final gradient norm. The convergence rate for this function is computed by first considering the final iterate, $\mathbf{x}_\star$, as the true solution. At each iteration k, the error $e_k = \|\mathbf{x}_k - \mathbf{x}_\star\|_2$ is calculated. If at least three iterations have been performed and all corresponding errors exceed a small threshold ε_{est}, the local order of convergence $p_{\text{est},k}$ is computed as

$$p_{\text{est},k} = \frac{\ln(e_{k+1}/e_k)}{\ln(e_k/e_{k-1})}.$$

All valid $p_{\text{est},k}$ values (i.e., those that meet the threshold criteria) are then averaged to obtain the estimated convergence rate, stored in `rate_convergence`. If fewer than three iterations are available, or if the threshold condition is not met for all applicable iterations, `rate_convergence` is set to `NaN`. This procedure provides a simple, vectorized approach to estimate the convergence rate of the iterative method.

1. **Modified Newton Method (Table 1):** This method generally requires fewer iterations compared to the Truncated Newton Method without preconditioning but is comparable to the preconditioned version. The execution time is also moderate. The convergence rate is quadratic, as expected.

Table 1: Modified Newton Method results on the Rosenbrock function

x(0)	Iterations	$\|\nabla f(x^*)\|$	exp. rate of conv.	execution time (in s)	conv. reached
x1	7	6.0364×10^{-10}	2.0808	8.5912×10^{-3}	yes
x2	19	2.6288×10^{-7}	1.9311	3.351×10^{-3}	yes
x3	12	3.6927×10^{-7}	2.0434	1.0486×10^{-3}	yes
x4	18	1.7582×10^{-8}	1.8722	9.998×10^{-4}	yes
x5	23	6.6474×10^{-7}	1.8414	3.7569×10^{-3}	yes

2. **Truncated Newton Method without Preconditioning (Table 2):** This method shows a significant increase in iterations for x_2 (61 iterations) and x_4 (58 iterations), making it the least efficient

among the three method. The convergence rate remains relatively low (superlinear). However, the final gradient norm is quite small, demonstrating precise convergence.

Table 2: Truncated Newton Method without preconditioning results on the Rosenbrock function

x(0)	Iterations	$\|\nabla f(x^*)\|$	exp. rate of conv.	execution time (in s)	conv. reached
x1	8	6.8021×10^{-14}	1.6995	6.3243×10^{-3}	yes
x2	61	1.0638×10^{-7}	1.5005	2.7278×10^{-3}	yes
x3	17	4.7363×10^{-8}	1.525	1.3613×10^{-3}	yes
x4	58	7.6834×10^{-7}	1.5047	1.0239×10^{-3}	yes
x5	27	2.1127×10^{-11}	1.6686	3.8229×10^{-3}	yes

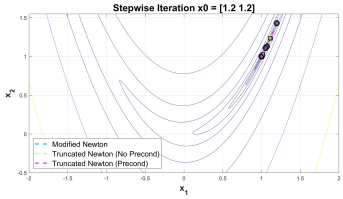
3. **Truncated Newton Method with Preconditioning (Table 3):** This method generally outperforms the unpreconditioned version in terms of the number of iterations and execution time. The number of iterations and final gradient norms are nearly identical to those of the Modified Newton Method. The exponential rate of convergence is slightly improved compared to the unpreconditioned version, that now is quadratic.

Table 3: Truncated Newton Method with preconditioning results on the Rosenbrock function

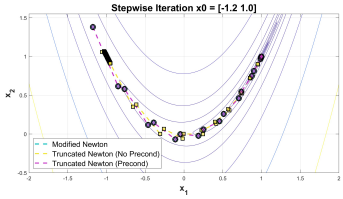
x(0)	Iterations	$\|\nabla f(x^*)\|$	exp. rate of conv.	execution time (in s)	conv. reached
x1	7	6.0367×10^{-10}	2.0524	7.7929×10^{-3}	yes
x2	19	2.6288×10^{-7}	1.9865	2.9559×10^{-3}	yes
x3	12	3.6927×10^{-7}	2.0838	9.198×10^{-4}	yes
x4	18	1.7582×10^{-8}	1.9142	9.605×10^{-4}	yes
x5	23	6.6474×10^{-7}	1.8844	3.4536×10^{-3}	yes

The choice of the starting point for the optimization algorithms emerges as a critical factor in determining convergence efficiency. Examining the results across different initial conditions reveals a clear pattern. Starting from x_1, which is located in close proximity to the global minimum of the Rosenbrock function, all methods achieve the fastest convergence, requiring a mere 7 to 8 iterations. In stark contrast, initial points x_2 and x_4, positioned farther from the optimal solution and within regions characterized by higher nonlinearity of the function, demand significantly more computational effort. These starting points lead to the highest iteration counts, reaching up to 61 iterations for the Truncated Newton Method without preconditioning. The saddle point, represented by initial condition x_3, exhibits an intermediate convergence behavior. Methods initiated at x_3 require a moderate number of iterations, ranging from 12 to 17. Finally, x_5, while starting further away from the minimum than x_1, still results in relatively efficient convergence, with iteration counts in the range of 23 to 27.

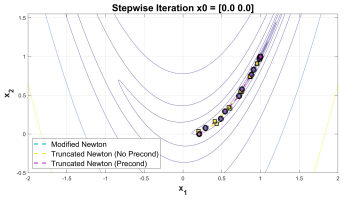
The Figure 2 illustrates the stepwise iteration paths of optimization algorithms for the Rosenbrock function, initiated from five distinct starting points. Panel (a), starting near the global minimum at $[1.2; 1.2]$, demonstrates rapid convergence towards the function's valley. Conversely, panels (b) and (e), originating from $[-1.2; 1]$ and $[-2; 1.5]$ respectively, reveal longer, more tortuous trajectories, indicating slower convergence due to the initial points being located further from the minimum and in regions of higher nonlinearity. Panel (c), starting at the saddle point $[0; 0]$, shows an initial movement away from the saddle before directing towards the function's minimum, exhibiting moderate convergence speed. Finally, panel (d), initialized at $[-1; 1]$, displays a convergence path that is longer than that of panel (a) but shorter than panels (b) and (e), reflecting its intermediate starting position relative to the minimum. The varying trajectories across these plots underscore the significant impact of the initial guess on the efficiency and path taken by optimization algorithms to reach the solution on the Rosenbrock function landscape.



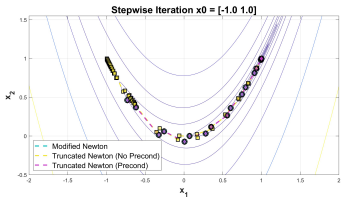
(a) Stepwise iteration for $[1.2; 1.2]$



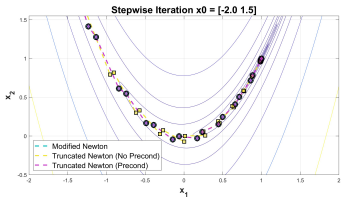
(b) Stepwise iteration for $[-1.2; 1]$



(c) Stepwise iteration for $[0; 0]$



(d) Stepwise iteration for $[-1; 1]$



(e) Stepwise iteration for $[-2; 1.5]$

Figure 2: Overview of Stepwise Iterations for Different Starting Points

Finite Difference Approximations of Derivatives. In this study, the exact derivative computations are replaced by finite difference approximations. A comprehensive grid search is conducted over a range of finite difference parameters, considering both a uniform scheme (with step size $h = 10^{-k}$) and a variable scheme (where each component uses $h_i = 10^{-k}|x_i|$). For each starting point and for every tested exponent k, the Modified Newton and Truncated Newton methods are executed using both central and forward difference approximations for the gradient and Hessian. Performance metrics such as the number of iterations, gradient norm, and function value are recorded and averaged across the starting points. The best finite difference parameter is then selected based on these averaged metrics, providing insights into the trade-off between accuracy and computational efficiency in the optimization process.

The optimal parameters, balancing accuracy and efficiency, are summarized as follows: for Modified Newton, `type_fixed_mn = 'c'`, `k_fixed_mn = 8`, `type_variable_mn = 'c'`, and `k_esp_mn = 8`; for Truncated Newton, `type_fixed_trunc = 'c'`, `k_fixed_trunc = 8`, `type_variable_trunc = 'fw'`, and `k_esp_trunc = 8`.

It is important to note that for Condition 3, a modification was necessary. Specifically, when employing a variable step size h_i, the step size consistently evaluated to zero because Condition 3 was initially defined at the point $[0, 0]$. To address this, Condition 3 was adjusted to $[1.5, 1.5]$ to enable the variable step size scheme to function effectively. The resulting performance metrics under this modified condition are presented in the subsequent table.

1. **Modified Newton Method (Table 4):** For most starting points, both the fixed and variable finite difference schemes yield similar numbers of iterations. However, the fixed scheme generally produces lower gradient norms (e.g., for x_1, 1.17×10^{-14} compared to 3.23×10^{-10}), indicating a higher precision in derivative approximation. In contrast, the variable scheme tends to offer slightly reduced execution times, which can be advantageous when computational efficiency is a priority.

Table 4: Modified Newton Method Results - Finite Differences

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x1	Fixed	8	$1.169\,113 \times 10^{-14}$	2.092 037	0.016 954
	Variable	8	$3.231\,972 \times 10^{-10}$	2.098 412	0.011 980
x2	Fixed	21	$3.853\,462 \times 10^{-11}$	1.515 755	0.002 622
	Variable	21	$5.518\,880 \times 10^{-7}$	1.525 805	0.001 322
x3	Fixed	10	$1.003\,883 \times 10^{-13}$	2.089 664	0.003 118
	Variable	12	$2.458\,210 \times 10^{-14}$	2.082 461	0.002 186
x4	Fixed	20	$7.144\,601 \times 10^{-13}$	1.395 288	0.001 482
	Variable	20	$7.144\,601 \times 10^{-13}$	1.395 288	0.000 828
x5	Fixed	23	$6.096\,779 \times 10^{-10}$	0.960 075	0.001 290
	Variable	21	$1.182\,229 \times 10^{-7}$	1.398 446	0.000 986

2. **Truncated Newton Method without Preconditioning (Table 5):** The performance between the fixed and variable schemes is relatively comparable in terms of iteration counts and convergence rates. Notably, in some cases such as x_1, the variable scheme achieves a lower gradient norm and a reduction in execution time. This suggests that while both schemes are viable, the variable finite difference approach may provide a better balance between accuracy and computational cost in certain scenarios.

Table 5: Truncated Newton Method (No Preconditioning) Results - Finite Differences

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x1	Fixed	9	$6.297\,555 \times 10^{-7}$	2.048 564	0.008 091
	Variable	9	$1.046\,430 \times 10^{-7}$	2.062 681	0.005 530
x2	Fixed	38	$1.083\,622 \times 10^{-10}$	2.028 681	0.003 710
	Variable	42	$2.818\,808 \times 10^{-11}$	2.046 899	0.003 375
x3	Fixed	11	$1.675\,323 \times 10^{-7}$	2.090 425	0.003 354
	Variable	12	$3.694\,954 \times 10^{-7}$	2.067 893	0.001 766
x4	Fixed	31	$5.704\,776 \times 10^{-13}$	2.012 268	0.000 587
	Variable	36	$3.131\,342 \times 10^{-7}$	2.067 743	0.000 533
x5	Fixed	47	$3.132\,344 \times 10^{-14}$	2.032 737	0.002 373
	Variable	41	$5.596\,079 \times 10^{-12}$	2.038 589	0.001 800

3. **Truncated Newton Method with Preconditioning (Table 6):** With preconditioning, the interaction between the finite difference scheme and the algorithm becomes more nuanced. Some cases using the fixed scheme display negative exponential rates (e.g., x_1 and x_3), which could indicate oscillatory behavior or sensitivity issues. The variable scheme, while sometimes requiring a few extra iterations, generally results in more consistent gradient norms and lower execution times. These observations highlight that preconditioning, when paired with an appropriately chosen finite difference strategy, can enhance performance, though care must be taken to tune the method appropriately.

Table 6: Truncated Newton Method with Preconditioning Results - Finite Differences

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x1	Fixed	8	3.382 174	$\times 10^{-10}$	2.031 337	0.010 082
	Variable	9	3.648 636	$\times 10^{-7}$	2.047 681	0.002 199
x2	Fixed	35	2.291 808	$\times 10^{-7}$	2.091 039	0.002 165
	Variable	40	2.925 984	$\times 10^{-9}$	2.032 255	0.001 228
x3	Fixed	11	3.440 780	$\times 10^{-8}$	2.011 337	0.003 465
	Variable	12	2.461 542	$\times 10^{-8}$	2.083 026 8	0.001 886
x4	Fixed	29	6.481 036	$\times 10^{-10}$	2.018 710	0.000 845
	Variable	33	2.875 315	$\times 10^{-8}$	2.044 522	0.000 568
x5	Fixed	45	1.008 551	$\times 10^{-7}$	2.011 258	0.001 686
	Variable	38	1.637 738	$\times 10^{-8}$	2.026 937	0.001 026

It is worth noting that, in this case, all iterations converge.

3.2 Problem 16. Banded trigonometric problem.

The **function** under consideration is defined as:

$$F(x) = \sum_{i=1}^{n} i\,[(1 - \cos x_i) + \sin x_{i-1} - \sin x_{i+1}]$$

A visualization of the function, in three dimension, is shown in Figure 3. In addition to the reference **starting point** $\bar{x} = [1, 1, \ldots, 1]$, ten initial points were randomly generated: these points are uniformly distributed within a hypercube centered at $\bar{x}$, defined by the Cartesian product of intervals $[\bar{x}_i - 1, \bar{x}_i + 1]$ for each dimension i.

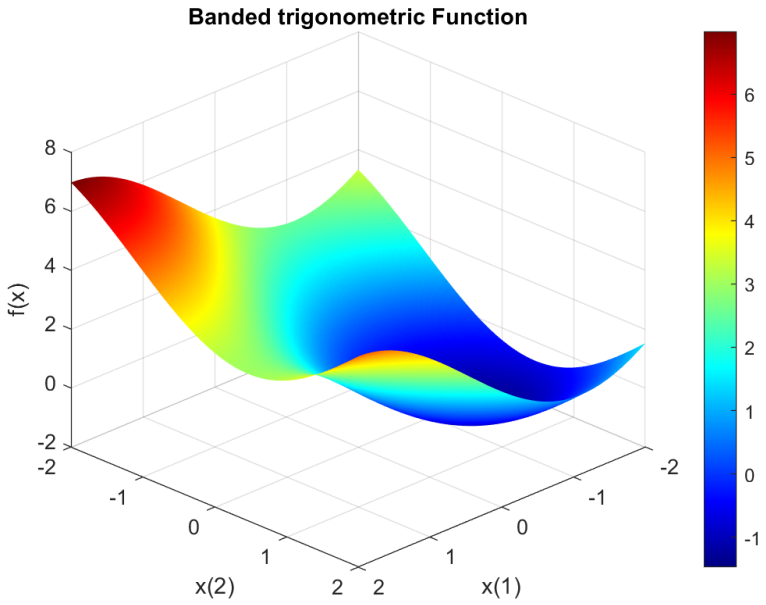


Figure 3: Plot of the Banded trigonometric function

In line with the approach taken previously, a grid search was also performed for this function. The results that emerged are presented below:
`Grid search result - Best combination: c1 = 0.0001, rho = 0.5, btmax = 100.`

3.2.1 Dimension $n = 10^3$

1. **Modified Newton Method (Table 7):** The Modified Newton method demonstrates robust performance across all initial conditions. It converges in a low number of iterations (ranging from 5 to 9), exhibiting quadratic convergence as expected. The execution times are consistently fast, staying below 0.04 seconds for all starting points, indicating efficiency for this function and dimension.

Table 7: Modified Newton Method Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	7	2.027 195	$\times 10^{-8}$	quadratic	0.039 769
x1	7	4.189 352	$\times 10^{-8}$	quadratic	0.016 469
x2	5	6.573 753	$\times 10^{-9}$	quadratic	0.011 639
x3	8	4.070 353	$\times 10^{-13}$	quadratic	0.016 510
x4	9	8.364 702	$\times 10^{-13}$	quadratic	0.022 005
x5	9	4.838 448	$\times 10^{-13}$	quadratic	0.018 536
x6	8	1.179 663	$\times 10^{-10}$	quadratic	0.015 190
x7	7	1.181 948	$\times 10^{-8}$	quadratic	0.014 205
x8	7	1.639 698	$\times 10^{-8}$	quadratic	0.014 172
x9	7	9.562 633	$\times 10^{-8}$	quadratic	0.013 394
x10	8	3.120 890	$\times 10^{-11}$	quadratic	0.014 967

2. **Truncated Newton Method without Preconditioning (Table 8):** The Truncated Newton method without preconditioning generally converges, but with a wider range of iterations and execution times compared to Modified Newton. Most runs show quadratic convergence, however, for initial condition x3, superlinear convergence is observed with a significantly higher iteration count and execution time. Notably, for initial condition x8, the method fails to converge within 1000 iterations, indicating potential instability for certain starting points with this approach. For the cases that converge, the execution times are comparable to Modified Newton, but less consistent.

Table 8: Truncated Newton Method (No Preconditioning) Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	12	1.421 502	$\times 10^{-11}$	quadratic	0.039 499
x1	12	1.110 104	$\times 10^{-13}$	quadratic	0.037 724
x2	16	2.561 557	$\times 10^{-7}$	quadratic	0.030 190
x3	293	1.963 942	$\times 10^{-9}$	superlinear	0.448 888
x4	17	1.679 984	$\times 10^{-10}$	quadratic	0.036 046
x5	11	5.988 650	$\times 10^{-13}$	quadratic	0.020 610
x6	10	1.819 730	$\times 10^{-11}$	quadratic	0.019 816
x7	11	1.896 953	$\times 10^{-7}$	quadratic	0.022 206
x8	1000	1.713 278	$\times 10^{-6}$	-	2.740 092
x9	17	3.683 697	$\times 10^{-13}$	quadratic	0.031 545
x10	14	2.665 955	$\times 10^{-10}$	quadratic	0.025 066

3. **Truncated Newton Method with Preconditioning (Table 9):** Applying preconditioning to the Truncated Newton method yields a more stable and consistent performance. All runs converge within a reasonable number of iterations (9 to 15), maintaining quadratic convergence. The execution times

are also consistently low, similar to or slightly better than the non-preconditioned Truncated Newton in most converging cases, and importantly, it avoids the non-convergence issue observed without preconditioning. Preconditioning appears to enhance the robustness and reliability of the Truncated Newton method for this function.

Table 9: Truncated Newton Method (with Preconditioning) Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x0	11	$8.914\,086 \times 10^{-11}$	quadratic	0.035 068
x1	11	$7.486\,502 \times 10^{-7}$	quadratic	0.021 310
x2	14	$2.932\,478 \times 10^{-8}$	quadratic	0.027 662
x3	14	$2.556\,871 \times 10^{-10}$	quadratic	0.030 634
x4	9	$5.456\,726 \times 10^{-7}$	quadratic	0.028 901
x5	11	$2.374\,262 \times 10^{-7}$	quadratic	0.019 488
x6	10	$9.717\,616 \times 10^{-11}$	quadratic	0.021 360
x7	13	$1.457\,674 \times 10^{-8}$	quadratic	0.025 512
x8	12	$4.933\,306 \times 10^{-7}$	quadratic	0.023 101
x9	15	$4.838\,528 \times 10^{-13}$	quadratic	0.026 247
x10	12	$1.602\,231 \times 10^{-10}$	quadratic	0.022 316

Finite Difference Approximations of Derivatives. The optimal parameters, balancing accuracy and efficiency, are summarized as follows: for Modified Newton, `type_fixed_mn = 'c'`, `k_fixed_mn = 4`, `type_variable_mn = 'c'`, and `k_esp_mn = 4`; for Truncated Newton, `type_fixed_trunc = 'c'`, `k_fixed_trunc = 4`, `type_variable_trunc = 'c'`, and `k_esp_trunc = 4`.

1. **Modified Newton Method (FD) (Table 10):** The Modified Newton method using Finite Differences shows varying performance across different initial conditions and FD types. For Condition 1, it converges in 35 iterations with an execution time of approximately 8 seconds. For other initial conditions, the iteration counts vary, sometimes requiring significantly more iterations. In general, Modified Newton exhibits longer execution times. The convergence rate is mostly quadratic, but linear convergence is observed for some initial conditions (x4, x9, x10). There's no clear consistent advantage in using either Fixed 'h' or Variable 'h' Finite Differences with Modified Newton in terms of iterations or execution time.

Table 10: Modified Newton Method Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	35	2.994 413	$\times 10^{-9}$	quadratic	8.310
	Variable	35	2.994 413	$\times 10^{-9}$	quadratic	8.123
x1	Fixed	30	2.635 721	$\times 10^{-9}$	quadratic	6.534
	Variable	30	6.901 550	$\times 10^{-8}$	quadratic	6.248
x2	Fixed	29	3.527 041	$\times 10^{-9}$	quadratic	6.430
	Variable	29	1.008 565	$\times 10^{-8}$	quadratic	5.989
x3	Fixed	28	3.152 120	$\times 10^{-9}$	quadratic	5.691
	Variable	28	9.248 712	$\times 10^{-9}$	quadratic	5.617
x4	Fixed	30	3.302 304	$\times 10^{-9}$	linear	7.868
	Variable	30	8.335 826	$\times 10^{-9}$	linear	5.899
x5	Fixed	28	1.170 823	$\times 10^{-8}$	quadratic	5.054
	Variable	28	1.000 683	$\times 10^{-8}$	quadratic	5.612
x6	Fixed	30	9.090 505	$\times 10^{-9}$	quadratic	5.415
	Variable	30	2.367 819	$\times 10^{-8}$	quadratic	6.094
x7	Fixed	28	6.075 868	$\times 10^{-9}$	quadratic	5.601
	Variable	28	1.217 382	$\times 10^{-8}$	quadratic	6.168
x8	Fixed	27	6.319 636	$\times 10^{-7}$	quadratic	5.578
	Variable	27	4.832 432	$\times 10^{-7}$	quadratic	5.056
x9	Fixed	31	5.290 028	$\times 10^{-8}$	linear	5.590
	Variable	31	2.083 768	$\times 10^{-8}$	linear	6.744
x10	Fixed	29	7.998 578	$\times 10^{-9}$	linear	6.090
	Variable	29	2.406 521	$\times 10^{-8}$	linear	5.157

2. **Truncated Newton Method (No Preconditioning) (FD) (Table 11):** The Truncated Newton method without preconditioning using Finite Differences generally outperforms the Modified Newton method in terms of both iterations and execution time. For Condition 1, it converges in 17 iterations, faster than Modified Newton. Across different initial conditions and FD types, the iteration counts are lower and more consistent, mostly around 17-19. Execution times are also significantly reduced compared to Modified Newton, remaining generally under 5 seconds. The convergence rate is predominantly quadratic, though superlinear convergence appears for initial condition x3 with variable 'h'. The choice between Fixed 'h' and Variable 'h' FD still does not show a major impact on performance.

Table 11: Truncated Newton Method (No Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	17	9.498 976	$\times 10^{-9}$	quadratic	3.978
	Variable	17	9.498 976	$\times 10^{-9}$	quadratic	3.830
x1	Fixed	18	1.206 166	$\times 10^{-8}$	quadratic	3.615
	Variable	18	1.501 984	$\times 10^{-8}$	quadratic	4.189
x2	Fixed	18	6.942 340	$\times 10^{-7}$	quadratic	3.891
	Variable	18	6.487 841	$\times 10^{-7}$	quadratic	3.529
x3	Fixed	18	5.525 812	$\times 10^{-9}$	quadratic	3.357
	Variable	25	9.431 491	$\times 10^{-9}$	superlinear	5.156
x4	Fixed	18	6.261 103	$\times 10^{-8}$	quadratic	3.877
	Variable	18	3.598 625	$\times 10^{-8}$	quadratic	3.228
x5	Fixed	17	6.161 680	$\times 10^{-9}$	quadratic	4.304
	Variable	19	1.166 636	$\times 10^{-8}$	quadratic	3.322
x6	Fixed	19	2.195 658	$\times 10^{-8}$	quadratic	4.011
	Variable	19	2.453 545	$\times 10^{-8}$	quadratic	3.914
x7	Fixed	18	5.684 342	$\times 10^{-9}$	quadratic	3.711
	Variable	18	9.254 768	$\times 10^{-9}$	quadratic	3.912
x8	Fixed	18	9.927 276	$\times 10^{-9}$	quadratic	4.108
	Variable	18	1.166 941	$\times 10^{-8}$	quadratic	3.373
x9	Fixed	18	5.768 977	$\times 10^{-9}$	quadratic	3.916
	Variable	18	1.419 525	$\times 10^{-8}$	quadratic	3.900
x10	Fixed	18	3.447 715	$\times 10^{-7}$	quadratic	3.384
	Variable	18	6.940 682	$\times 10^{-8}$	quadratic	3.131

3. **Truncated Newton Method (with Preconditioning) (FD) (Table 12):** The Truncated Newton method with preconditioning and Finite Differences achieves the best performance overall. It consistently requires the fewest iterations and the shortest execution times among the three methods. For Condition 1, convergence is reached in 14 iterations. Across all initial conditions, iteration counts are minimal and execution times are generally under 3 seconds, often faster than the non-preconditioned Truncated Newton. The convergence rate is mostly quadratic, with some instances of superlinear convergence (x0, x7, x8). As observed with the other methods, the choice between Fixed 'h' and Variable 'h' FD has a negligible impact on the performance of preconditioned Truncated Newton, maintaining its efficiency and speed regardless of the FD type.

Table 12: Truncated Newton Method (with Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	14	1.112 448	$\times 10^{-8}$	superlinear	3.222
	Variable	14	1.112 448	$\times 10^{-8}$	superlinear	3.244
x1	Fixed	11	5.817 779	$\times 10^{-9}$	quadratic	2.433
	Variable	9	9.608 488	$\times 10^{-9}$	quadratic	2.085
x2	Fixed	9	6.549 339	$\times 10^{-9}$	quadratic	1.872
	Variable	9	9.823 023	$\times 10^{-9}$	quadratic	1.839
x3	Fixed	13	5.256 096	$\times 10^{-9}$	quadratic	2.426
	Variable	13	1.019 343	$\times 10^{-8}$	quadratic	2.609
x4	Fixed	13	5.576 742	$\times 10^{-9}$	quadratic	2.637
	Variable	19	9.429 230	$\times 10^{-7}$	quadratic	3.300
x5	Fixed	12	8.741 697	$\times 10^{-9}$	quadratic	2.875
	Variable	16	1.072 444	$\times 10^{-8}$	quadratic	2.870
x6	Fixed	11	5.648 703	$\times 10^{-9}$	quadratic	2.058
	Variable	11	8.759 379	$\times 10^{-9}$	quadratic	2.147
x7	Fixed	14	5.518 498	$\times 10^{-9}$	superlinear	3.088
	Variable	14	1.259 891	$\times 10^{-8}$	superlinear	2.975
x8	Fixed	13	5.782 962	$\times 10^{-9}$	superlinear	2.600
	Variable	13	1.039 336	$\times 10^{-8}$	superlinear	2.289
x9	Fixed	14	5.662 985	$\times 10^{-9}$	quadratic	3.579
	Variable	14	1.095 821	$\times 10^{-8}$	quadratic	2.831
x10	Fixed	13	5.838 569	$\times 10^{-9}$	linear	2.671
	Variable	14	1.151 756	$\times 10^{-8}$	linear	2.539

3.2.2 Dimension $n = 10^4$

1. **Modified Newton Method (Table 13):** The Modified Newton method continues to perform effectively as the dimension increases to $n = 10^4$. It achieves convergence in a consistently low number of iterations, ranging from 7 to 10. The method maintains quadratic convergence for all initial conditions, as indicated by the "quadratic" rate. Execution times are slightly higher compared to $n = 10^3$, but remain reasonably fast, staying within the 1-1.3 seconds range, demonstrating good scalability.

Table 13: Modified Newton Method Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	8	1.117 341	$\times 10^{-12}$	quadratic	1.119 308
x1	7	2.852 167	$\times 10^{-8}$	quadratic	0.979 768
x2	7	7.770 784	$\times 10^{-12}$	quadratic	0.975 023
x3	8	5.600 580	$\times 10^{-12}$	quadratic	1.098 273
x4	10	6.876 721	$\times 10^{-7}$	quadratic	1.346 033
x5	9	4.873 210	$\times 10^{-9}$	quadratic	1.224 569
x6	8	1.110 112	$\times 10^{-12}$	quadratic	1.102 649
x7	7	3.686 998	$\times 10^{-7}$	quadratic	0.980 913
x8	10	2.291 146	$\times 10^{-8}$	quadratic	1.348 203
x9	9	1.110 112	$\times 10^{-12}$	quadratic	1.221 674
x10	10	1.110 112	$\times 10^{-12}$	quadratic	1.346 728

2. **Truncated Newton Method without Preconditioning (Table 14):** The Truncated Newton method without preconditioning shows increased variability in performance compared to the Modified Newton method. While most initial conditions converge with a quadratic rate, the number of iterations and execution times are generally higher and less consistent than for Modified Newton. Some runs exhibit superlinear convergence (x2, x6, x10), requiring a larger number of iterations and longer execution times. Overall, while convergence is achieved, the method becomes less predictable and slightly less efficient as dimension increases.

Table 14: Truncated Newton Method (No Preconditioning) Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	14	8.413 204	$\times 10^{-7}$	quadratic	1.871 534
x1	12	3.113 703	$\times 10^{-9}$	quadratic	1.605 068
x2	28	4.250 133	$\times 10^{-9}$	superlinear	3.116 317
x3	15	2.063 122	$\times 10^{-7}$	quadratic	2.366 022
x4	12	1.798 923	$\times 10^{-9}$	quadratic	1.576 694
x5	11	8.338 790	$\times 10^{-10}$	quadratic	1.490 921
x6	21	7.804 749	$\times 10^{-9}$	superlinear	2.977 751
x7	14	5.198 325	$\times 10^{-7}$	quadratic	1.833 622
x8	19	4.551 836	$\times 10^{-11}$	quadratic	2.112 555
x9	18	7.476 558	$\times 10^{-7}$	quadratic	2.357 712
x10	21	3.697 333	$\times 10^{-9}$	superlinear	2.472 674

3. **Truncated Newton Method with Preconditioning (Table 15):** Convergence is achieved for all initial conditions, with iterations ranging from 9 to 17. All runs exhibit quadratic convergence. The execution times are generally in the 1.2 to 2.2 seconds range, showing slightly increased computational cost compared to $n = 10^3$, but still maintaining reasonable efficiency and consistency across different starting points. Preconditioning helps maintain the robustness of Truncated Newton as the problem dimension grows.

Table 15: Truncated Newton Method (with Preconditioning) Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	12	3.291 667	$\times 10^{-9}$	quadratic	1.636 073
x1	11	3.096 650	$\times 10^{-9}$	quadratic	1.482 805
x2	17	2.746 291	$\times 10^{-9}$	quadratic	2.225 813
x3	13	1.224 450	$\times 10^{-8}$	quadratic	1.729 873
x4	15	4.563 630	$\times 10^{-9}$	quadratic	1.945 798
x5	9	2.920 764	$\times 10^{-7}$	quadratic	1.234 227
x6	11	1.072 856	$\times 10^{-9}$	quadratic	1.476 519
x7	14	3.087 548	$\times 10^{-10}$	quadratic	1.836 319
x8	16	1.222 354	$\times 10^{-9}$	quadratic	2.087 987
x9	12	6.370 749	$\times 10^{-9}$	quadratic	1.609 314
x10	15	2.054 247	$\times 10^{-10}$	quadratic	1.960 868

Finite Difference Approximations of Derivatives. The execution was interrupted after only five initial points, as these proved to be sufficient to demonstrate its lack of competitiveness and high time expenditure compared to the gradient and Hessiana version in function handle format.

1. **Modified Newton Method (FD) (Table 16):** The Modified Newton method with Finite Differences shows significant computational cost, with execution times much larger than those observed with exact derivatives. For Condition 1, it converges in 23 iterations, taking approximately 289 seconds with fixed

'h' FD and 263 seconds with variable 'h' FD. Condition 2 and subsequent conditions (up to 5 are shown) also demonstrate high iteration counts (up to 79 iterations for condition 2) and long execution times (up to over 700 seconds for condition 2). There is no clear advantage in using variable 'h' FD over fixed 'h' FD in terms of computational time for Modified Newton in these results.

Table 16: Modified Newton Method Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	23	4.780 258	$\times 10^{-7}$	quadratic	289.352
	Variable	23	4.780 258	$\times 10^{-7}$	quadratic	263.744
x1	Fixed	79	3.494 165	$\times 10^{-7}$	linear	709.391
	Variable	79	3.787 575	$\times 10^{-7}$	linear	749.539
x2	Fixed	56	9.438 719	$\times 10^{-7}$	linear	492.280
	Variable	54	4.221 580	$\times 10^{-7}$	linear	467.636
x3	Fixed	44	5.137 236	$\times 10^{-7}$	linear	407.897
	Variable	45	5.365 146	$\times 10^{-7}$	linear	507.258
x4	Fixed	52	5.076 700	$\times 10^{-7}$	linear	730.803
	Variable	54	4.095 718	$\times 10^{-7}$	linear	488.982

2. **Truncated Newton Method (No Preconditioning) (FD) (Table 17):** The Truncated Newton method without preconditioning, while faster than Modified Newton with FD, still exhibits substantial execution times. For Condition 1, it converges in 27 iterations, taking approximately 324 seconds with fixed 'h' FD and 255 seconds with variable 'h' FD. For other initial conditions (up to 5 shown), iteration counts remain around 26-29, and execution times are generally in the range of 230-400 seconds. Variable 'h' FD appears to offer slightly faster execution times compared to fixed 'h' FD for Truncated Newton without preconditioning in these cases.

Table 17: Truncated Newton Method (No Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	27	4.799 901	$\times 10^{-7}$	quadratic	324.974
	Variable	27	4.799 901	$\times 10^{-7}$	quadratic	255.362
x1	Fixed	26	6.246 779	$\times 10^{-7}$	quadratic	231.314
	Variable	26	4.375 186	$\times 10^{-7}$	quadratic	229.660
x2	Fixed	27	6.876 165	$\times 10^{-7}$	quadratic	233.872
	Variable	27	8.499 476	$\times 10^{-7}$	quadratic	236.571
x3	Fixed	26	4.433 029	$\times 10^{-7}$	quadratic	233.812
	Variable	29	4.994 642	$\times 10^{-7}$	quadratic	403.756
x4	Fixed	28	4.210 283	$\times 10^{-7}$	quadratic	393.508
	Variable	27	5.523 658	$\times 10^{-7}$	quadratic	236.365

3. **Truncated Newton Method (with Preconditioning) (FD) (Table 18):** The Truncated Newton method with preconditioning and Finite Differences provides the most efficient performance among the tested methods using FD, though still computationally expensive compared to methods with exact derivatives. For Condition 1, it converges in 14 iterations with an execution time of approximately 177 seconds using fixed 'h' FD, and 125 seconds with variable 'h' FD. Across initial conditions 1-5, iteration counts are consistently lower, around 9-17, and execution times are reduced to a range of approximately 100-180 seconds. Variable 'h' FD consistently leads to faster execution times compared to fixed 'h' FD for the preconditioned Truncated Newton method in these results.

Table 18: Truncated Newton Method (with Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	14	5.239 866	$\times 10^{-7}$	quadratic	177.226
	Variable	14	5.239 866	$\times 10^{-7}$	quadratic	125.855
x1	Fixed	17	4.324 167	$\times 10^{-7}$	quadratic	160.721
	Variable	17	4.293 445	$\times 10^{-7}$	quadratic	153.172
x2	Fixed	15	4.035 739	$\times 10^{-7}$	quadratic	138.562
	Variable	15	5.048 518	$\times 10^{-7}$	quadratic	137.406
x3	Fixed	12	3.459 668	$\times 10^{-7}$	quadratic	108.346
	Variable	12	4.421 562	$\times 10^{-7}$	quadratic	173.273
x4	Fixed	18	3.529 789	$\times 10^{-7}$	quadratic	177.763
	Variable	18	5.529 657	$\times 10^{-7}$	quadratic	212.365

3.2.3 Dimension $n = 10^5$

1. **Modified Newton Method (Table 19):** The Modified Newton method still converges for all initial conditions. However, a significant change is observed in the convergence rate, which degrades to linear for all runs. The number of iterations remains low, generally between 7 and 9, except for x6 which requires 21 iterations. Execution times dramatically increase compared to lower dimensions, now ranging from approximately 94 to 118 seconds for most cases, and reaching 259 seconds for x6, indicating a substantial rise in computational cost with increasing dimension.

Table 19: Modified Newton Method Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	8	1.878 385	$\times 10^{-8}$	linear	105.806
x1	8	1.892 651	$\times 10^{-10}$	linear	105.549
x2	7	2.060 820	$\times 10^{-8}$	linear	94.296
x3	9	1.144 624	$\times 10^{-11}$	linear	117.875
x4	8	6.914 902	$\times 10^{-8}$	linear	106.047
x5	9	2.498 077	$\times 10^{-7}$	linear	117.811
x6	21	5.908 378	$\times 10^{-7}$	linear	259.262
x7	8	6.801 542	$\times 10^{-7}$	linear	105.406
x8	7	4.018 463	$\times 10^{-8}$	linear	93.889
x9	9	4.773 681	$\times 10^{-11}$	linear	117.432
x10	8	9.903 863	$\times 10^{-9}$	linear	105.706

2. **Truncated Newton Method without Preconditioning (Table 20):** The Truncated Newton method without preconditioning exhibits less consistent convergence behavior. While it converges for all initial conditions tested, the number of iterations and execution times are highly variable and generally much larger compared to Modified Newton. The convergence rate is consistently linear across all runs. Execution times range from around 165 seconds up to over 632 seconds, with higher iteration counts leading to significantly longer computation times, indicating sensitivity to initial conditions and a substantial computational burden for this dimension.

Table 20: Truncated Newton Method (No Preconditioning) Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	18	3.261 098	$\times 10^{-8}$	linear	223.506
x1	50	3.844 601	$\times 10^{-4}$	linear	600.750
x2	56	3.540 235	$\times 10^{-7}$	linear	632.344
x3	50	3.240 354	$\times 10^{-3}$	linear	602.033
x4	13	2.225 689	$\times 10^{-8}$	linear	164.909
x5	16	3.629 321	$\times 10^{-8}$	linear	200.347
x6	50	8.649 681	$\times 10^{-4}$	linear	598.827
x7	50	8.022 400	$\times 10^{-4}$	linear	598.297
x8	14	5.970 112	$\times 10^{-11}$	linear	175.924
x9	50	5.168 195	$\times 10^{-5}$	linear	598.925
x10	15	2.943 819	$\times 10^{-8}$	linear	196.087

3. **Truncated Newton Method with Preconditioning (Table 21):** Applying preconditioning to the Truncated Newton method improves the consistency in terms of iteration counts, but it still exhibits linear convergence. The number of iterations is generally lower than the non-preconditioned Truncated Newton, mostly ranging from 10 to 14 iterations, except for x2 which requires 50 iterations. Execution times, while somewhat reduced compared to the non-preconditioned version in some cases, are still substantial, ranging from approximately 129 to 176 seconds for most runs and reaching over 600 seconds for x2. Preconditioning helps in making the iteration count more predictable but does not restore quadratic convergence, and the execution time remains a significant factor at this dimension.

Table 21: Truncated Newton Method (with Preconditioning) Results on the Banded Trigonometric Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	12	2.105 611	$\times 10^{-9}$	linear	152.936
x1	13	1.334 371	$\times 10^{-8}$	linear	164.792
x2	50	3.458 508	$\times 10^{-6}$	linear	602.209
x3	12	1.445 083	$\times 10^{-8}$	linear	153.447
x4	10	1.313 657	$\times 10^{-7}$	linear	129.977
x5	11	6.423 250	$\times 10^{-8}$	linear	141.138
x6	11	7.673 696	$\times 10^{-7}$	linear	141.551
x7	13	4.114 022	$\times 10^{-7}$	linear	164.095
x8	12	3.791 725	$\times 10^{-8}$	linear	152.269
x9	14	2.786 370	$\times 10^{-8}$	linear	176.076
x10	12	2.943 819	$\times 10^{-8}$	linear	152.767

Finite Difference Approximations of Derivatives. In this scenario, the results obtained using finite difference approximations are not competitive when compared to employing exact gradients and Hessians. Indeed, the time required to produce results is exceedingly long. The execution was not halted prematurely after a few points, and consequently, the outcomes are not being reported due to the protracted execution times and lack of competitive performance.

3.3 Problem 27. Penality Function

The **function** under consideration is defined as:

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

$$f_k(x) = \sqrt{\frac{1}{100000}}(x_k - 1); \quad 1 \le k \le n; \qquad f_k(x) = \sum_{i=1}^{n} x_i^2 - \frac{1}{4}; \quad k = n+1,\ m = n+1;$$

A visualization of the function, in three dimension, is shown in Figure 4. In addition to the reference **starting point** $\bar{x} = [1, 2, 3, \ldots, n]$, ten initial points were randomly generated: these points are uniformly distributed within a hypercube centered at $\bar{x}$, defined by the Cartesian product of intervals $[\bar{x}_i - 1, \bar{x}_i + 1]$ for each dimension i.

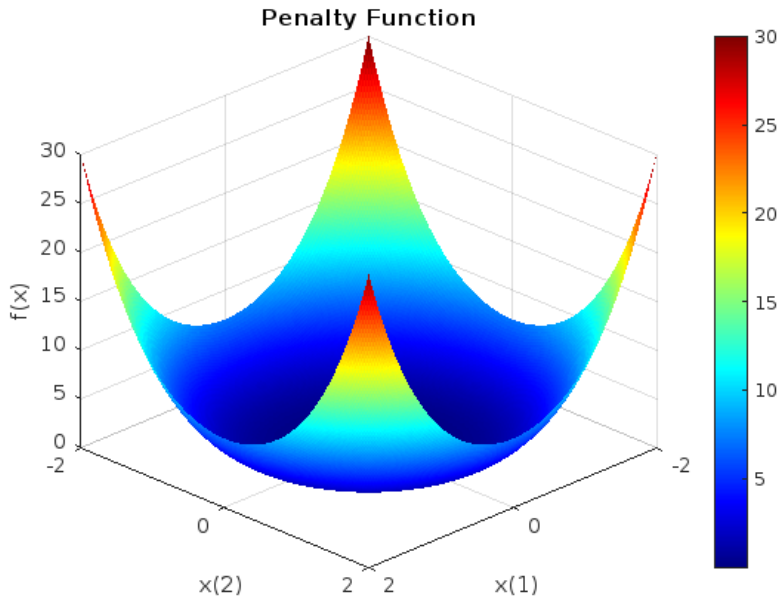


Figure 4: Plot of the Penalty function

In line with the approach taken previously, a grid search was also performed for this function. The results that emerged are presented below:
`Grid search result - Best combination: c1 = 0.0001, rho = 0.8, btmax = 30`.

3.3.1 Dimension $n = 10^3$

1. **Modified Newton Method (Table 22):** The Modified Newton method demonstrates consistent performance across all initial conditions. It converges in a uniform number of iterations, 39 for all starting points. The method exhibits quadratic convergence as indicated. Execution times are also quite consistent, ranging from approximately 0.59 to 0.66 seconds, indicating good efficiency for this dimension and function.

Table 22: Modified Newton Method Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	39	4.124 047	$\times 10^{-7}$	quadratic	0.662 871
x1	39	4.111 701	$\times 10^{-7}$	quadratic	0.627 325
x2	39	4.157 885	$\times 10^{-7}$	quadratic	0.644 176
x3	39	4.116 384	$\times 10^{-7}$	quadratic	0.634 669
x4	39	4.147 581	$\times 10^{-7}$	quadratic	0.615 893
x5	39	4.130 816	$\times 10^{-7}$	quadratic	0.587 541
x6	39	4.080 418	$\times 10^{-7}$	quadratic	0.618 813
x7	39	4.126 676	$\times 10^{-7}$	quadratic	0.642 857
x8	39	4.106 687	$\times 10^{-7}$	quadratic	0.588 732
x9	39	4.143 110	$\times 10^{-7}$	quadratic	0.597 018
x10	39	4.201 055	$\times 10^{-7}$	quadratic	0.618 999

2. **Truncated Newton Method (No Preconditioning) Results (Table 23):** Shows very consistent performance. Similar to Modified Newton, it converges in a uniform number of iterations, 51 in this case, across all initial conditions. The convergence rate is quadratic. Execution times are slightly faster than Modified Newton.

Table 23: Truncated Newton Method (no Preconditioning) Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	51	5.616 086	$\times 10^{-7}$	quadratic	0.468
x1	51	5.627 397	$\times 10^{-7}$	quadratic	0.420
x2	51	5.510 834	$\times 10^{-7}$	quadratic	0.423
x3	51	5.616 634	$\times 10^{-7}$	quadratic	0.429
x4	51	5.612 064	$\times 10^{-7}$	quadratic	0.423
x5	51	5.598 696	$\times 10^{-7}$	quadratic	0.423
x6	51	5.616 584	$\times 10^{-7}$	quadratic	0.420
x7	51	5.611 785	$\times 10^{-7}$	quadratic	0.421
x8	51	5.612 667	$\times 10^{-7}$	quadratic	0.420
x9	51	5.673 661	$\times 10^{-7}$	quadratic	0.421
x10	51	5.599 075	$\times 10^{-7}$	quadratic	0.417

3. **Truncated Newton Method with Preconditioning Results (Table 24):** Shows again very consistent performance. It converges in a uniform number of iterations, 39, identical to the Modified Newton, across all initial conditions. The convergence rate remains quadratic. Execution times are the fastest among the three methods, ranging from approximately 0.35 to 0.40 seconds, demonstrating the best efficiency for the Penalty function at this dimension.

Table 24: Truncated Newton Method (with Preconditioning) Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	39	9.283 706	$\times 10^{-7}$	quadratic	0.387 676
x1	39	9.248 145	$\times 10^{-7}$	quadratic	0.398 074
x2	39	9.386 111	$\times 10^{-7}$	quadratic	0.400 672
x3	39	9.260 919	$\times 10^{-7}$	quadratic	0.379 828
x4	39	9.354 903	$\times 10^{-7}$	quadratic	0.356 768
x5	39	9.307 064	$\times 10^{-7}$	quadratic	0.379 570
x6	39	9.149 858	$\times 10^{-7}$	quadratic	0.387 394
x7	39	9.291 622	$\times 10^{-7}$	quadratic	0.352 548
x8	39	9.231 365	$\times 10^{-7}$	quadratic	0.375 771
x9	39	9.343 434	$\times 10^{-7}$	quadratic	0.364 978
x10	39	9.518 363	$\times 10^{-7}$	quadratic	0.375 412

Finite Difference Approximations of Derivatives. In line with the approach taken previously, a grid search was also performed for this function. The optimal parameters, balancing accuracy and efficiency, are summarized as follows: for Modified Newton, `type_fixed_mn = 'c'`, `k_fixed_mn = 4`, `type_variable_mn = 'fw'`, and `k_esp_mn = 6`; for Truncated Newton, `type_fixed_trunc = 'fw'`, `k_fixed_trunc = 6`, `type_variable_trunc = 'fw'`, and `k_esp_trunc = 6`. Despite its convergence, the Modified Newton method exhibits excessive slowness of execution. Given its inefficiency compared to methods using gradient and analytical Hessiana, the investigation of its performance was restricted to the first five starting points.

1. **Modified Newton Method - Finite Differences (Table 25):** The Modified Newton method using Finite Differences exhibits significantly different behavior depending on whether a fixed or variable h is used for the FD approximation. When using a fixed h, the Modified Newton method converges for all initial conditions. However, the convergence rate is only linear, and the number of iterations is relatively high, around 400 iterations for each starting point. The execution times are also substantial, approximately 260 to 308 seconds. In stark contrast, when a variable h is employed for Finite Differences in the Modified Newton method, it **fails to converge** for any of the initial conditions within the maximum iteration limit of 1000. This non-convergence is indicated by the "-" in the 'Exp. Rate of Conv.' column and the high iteration count of 1000. Furthermore, the execution times are very high, exceeding 700 seconds in all cases, demonstrating the inefficiency and instability of this approach.

Table 25: Modified Newton Method Results - Finite Differences

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	413	2.314 450	$\times 10^{-8}$	linear	290.992
	Variable	1000	1.032 689	$\times 10^{13}$	-	719.069
x1	Fixed	408	1.455 700	$\times 10^{-8}$	linear	290.126
	Variable	1000	1.044 383	$\times 10^{13}$	-	721.656
x2	Fixed	425	4.884 840	$\times 10^{-7}$	linear	304.880
	Variable	1000	1.084 437	$\times 10^{13}$	-	733.643
x3	Fixed	428	7.470 443	$\times 10^{-7}$	linear	308.015
	Variable	1000	1.115 570	$\times 10^{13}$	-	745.825
x4	Fixed	409	5.697 164	$\times 10^{-7}$	linear	293.098
	Variable	1000	9.699 891	$\times 10^{12}$	-	713.145
x5	Fixed	373	6.861 401	$\times 10^{-7}$	linear	262.779
	Variable	1000	9.646 668	$\times 10^{12}$	-	710.644

2. **Truncated Newton Method (No Preconditioning) - Finite Differences (FD) (Table 26):** The Truncated Newton method without preconditioning, when using Finite Differences, converges rapidly for both fixed and variable h approximations. With a fixed h, the method converges within a small number of iterations, ranging from 1 to 10, exhibiting quadratic convergence for all initial conditions. Execution times are also very low, generally under 8 seconds. Using a variable h also leads to rapid convergence in just 1 to 4 iterations, with quadratic convergence observed. Execution times are even faster than with fixed h, consistently staying around 1.3 to 3.4 seconds.

Table 26: Truncated Newton Method (No Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	6	0	quadratic	5.368
	Variable	1	0	quadratic	1.366
x1	Fixed	1	0	quadratic	1.366
	Variable	1	0	quadratic	1.360
x2	Fixed	4	0	quadratic	3.323
	Variable	2	0	quadratic	2.000
x3	Fixed	2	0	quadratic	2.008
	Variable	3	0	quadratic	2.662
x4	Fixed	2	0	quadratic	2.005
	Variable	3	0	quadratic	2.690
x5	Fixed	3	0	quadratic	2.647
	Variable	4	0	quadratic	3.360
x6	Fixed	10	0	quadratic	8.009
	Variable	4	0	quadratic	3.370
x7	Fixed	2	0	quadratic	2.018
	Variable	2	0	quadratic	2.065
x8	Fixed	6	0	quadratic	4.696
	Variable	4	0	quadratic	3.321
x9	Fixed	2	0	quadratic	1.995
	Variable	3	0	quadratic	3.065

3. **Truncated Newton Method (with Preconditioning) - Finite Differences (FD) (Table 27):** Applying preconditioning to the Truncated Newton method with Finite Differences maintains the fast convergence observed without preconditioning. With preconditioning and fixed h, convergence is achieved in a small number of iterations (1 to 7) with quadratic rate and low execution times (under 5.5 seconds). Preconditioning with variable h also results in very rapid convergence within 1 to 5 iterations, exhibiting quadratic convergence and even faster execution times, mostly in the 1.3 to 4 seconds range.

Table 27: Truncated Newton Method (with Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	5	0	quadratic	4.058
	Variable	1	0	quadratic	1.344
x1	Fixed	1	0	quadratic	1.353
	Variable	1	0	quadratic	1.336
x2	Fixed	4	0	quadratic	3.354
	Variable	3	0	quadratic	2.681
x3	Fixed	2	0	quadratic	2.015
	Variable	2	0	quadratic	2.010
x4	Fixed	3	0	quadratic	2.672
	Variable	2	0	quadratic	2.030
x5	Fixed	2	0	quadratic	2.003
	Variable	2	0	quadratic	1.987
x6	Fixed	7	0	quadratic	5.411
	Variable	2	0	quadratic	2.029
x7	Fixed	3	0	quadratic	2.667
	Variable	5	0	quadratic	4.039
x8	Fixed	2	0	quadratic	2.021
	Variable	2	0	quadratic	2.023
x9	Fixed	2	0	quadratic	2.077
	Variable	2	0	quadratic	2.022

3.3.2 Dimension $n = 10^4$

1. **Modified Newton Method (Table 28):** The Modified Newton method maintains consistent performance across different initial conditions. It converges in a uniform number of iterations, 45 for all starting points. Notably, the convergence rate observed is superlinear. Execution times are also consistent, ranging from approximately 186 to 215 seconds, indicating a significant increase compared to the $d = 10^3$ case but still relatively uniform across different initial points and demonstrating good scalability for this method.

Table 28: Modified Newton Method Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	45	6.999 313	$\times 10^{-8}$	superlinear	195.958
x1	45	6.999 386	$\times 10^{-8}$	superlinear	196.084
x2	45	6.999 043	$\times 10^{-8}$	superlinear	200.349
x3	45	6.999 327	$\times 10^{-8}$	superlinear	190.467
x4	45	6.999 172	$\times 10^{-8}$	superlinear	191.119
x5	45	6.999 159	$\times 10^{-8}$	superlinear	190.262
x6	45	6.999 385	$\times 10^{-8}$	superlinear	186.668
x7	45	6.998 914	$\times 10^{-8}$	superlinear	194.318
x8	45	6.999 683	$\times 10^{-8}$	superlinear	215.176
x9	45	6.999 580	$\times 10^{-8}$	superlinear	212.837
x10	45	6.999 094	$\times 10^{-8}$	superlinear	195.678

2. **Truncated Newton Method (with Preconditioning) Results (Table 29):** This method also

shows very consistent behavior. It converges in a nearly uniform number of iterations, mostly 47 and 48, with one run at 45 iterations. The observed convergence rate is superlinear, similar to the Modified Newton method for this dimension. Execution times are significantly faster than Modified Newton, ranging from approximately 90 to 99 seconds, indicating better efficiency.

Table 29: Truncated Newton Method (with Preconditioning) Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	47	8.704 208	$\times 10^{-8}$	superlinear	98.136
x1	47	8.704 336	$\times 10^{-8}$	superlinear	98.328
x2	47	8.704 194	$\times 10^{-8}$	superlinear	98.556
x3	47	8.704 498	$\times 10^{-8}$	superlinear	97.676
x4	47	8.704 190	$\times 10^{-8}$	superlinear	98.956
x5	47	8.704 122	$\times 10^{-8}$	superlinear	98.712
x6	47	8.704 382	$\times 10^{-8}$	superlinear	98.511
x7	47	8.703 892	$\times 10^{-8}$	superlinear	97.818
x8	48	8.704 172	$\times 10^{-8}$	superlinear	99.375
x9	45	8.703 958	$\times 10^{-8}$	superlinear	93.818
x10	47	8.704 211	$\times 10^{-8}$	superlinear	98.187

3. **Truncated Newton Method (with Preconditioning) Results (Table 30):** It shows consistent number of iterations (45), superlinear convergence, and similar execution times in the range of 90 to 98 seconds.

Table 30: Truncated Newton Method (with Preconditioning) Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	45	9.124 732	$\times 10^{-8}$	superlinear	95.707
x1	45	9.124 845	$\times 10^{-8}$	superlinear	94.325
x2	45	9.124 389	$\times 10^{-8}$	superlinear	95.385
x3	45	9.124 757	$\times 10^{-8}$	superlinear	90.537
x4	45	9.124 547	$\times 10^{-8}$	superlinear	92.366
x5	45	9.124 541	$\times 10^{-8}$	superlinear	90.639
x6	45	9.124 840	$\times 10^{-8}$	superlinear	91.515
x7	45	9.124 227	$\times 10^{-8}$	superlinear	98.054
x8	45	9.125 211	$\times 10^{-8}$	superlinear	98.412
x9	45	9.125 091	$\times 10^{-8}$	superlinear	93.164
x10	45	9.124 388	$\times 10^{-8}$	superlinear	92.366

Finite Difference Approximations of Derivatives. In view of the unsatisfactory performance found for this dimension, in comparison to the results obtained with gradient and Hessiana analysis, it was deemed appropriate to discontinue the experiment. Given the significant difference in effectiveness, the analysis for dimension (10^5) was not continued.

3.3.3 Dimension $n = 10^5$

For this function, the structure of the Hessiana, with values $(4x \cdot x')$ on the sub- and supradiagonal, made direct calculation for high dimensions, such as (10^5), prohibitive. In fact, even MATLAB reported the impossibility of handling matrix operations ($10^5 \times 10^5$), a circumstance perhaps accentuated by the use of MATLAB Online. In order to overcome these computational difficulties, it became necessary to implement and use two Hessian-free variants of Modified Newton and Truncated Newton's method, thus enabling the results shown in the following tables to be obtained.

1. **Modified Newton Method (Table 31):** It exhibits consistent behavior across all initial conditions. It converges in a uniform number of iterations, 51 for every starting point. The method shows quadratic convergence rate in all cases, a change from the superlinear convergence observed at $d = 10^4$. Execution times are remarkably consistent and very low, ranging from approximately 0.055 to 0.063 seconds, indicating high efficiency.

Table 31: Modified Newton Method Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x0	51	$3.755\,110 \times 10^{-7}$	quadratic	0.063 191
x1	51	$3.755\,114 \times 10^{-7}$	quadratic	0.056 758
x2	51	$3.755\,104 \times 10^{-7}$	quadratic	0.057 741
x3	51	$3.755\,116 \times 10^{-7}$	quadratic	0.057 114
x4	51	$3.755\,108 \times 10^{-7}$	quadratic	0.058 797
x5	51	$3.755\,111 \times 10^{-7}$	quadratic	0.055 207
x6	51	$3.755\,111 \times 10^{-7}$	quadratic	0.057 919
x7	51	$3.755\,115 \times 10^{-7}$	quadratic	0.055 260
x8	51	$3.755\,113 \times 10^{-7}$	quadratic	0.055 967
x9	51	$3.755\,113 \times 10^{-7}$	quadratic	0.056 084
x10	51	$3.755\,114 \times 10^{-7}$	quadratic	0.057 217

2. **Truncated Newton Method (No Preconditioning) Results (Table 32):** The number of iterations is nearly uniform, mostly 53 iterations with some runs slightly higher at 54 and 55, and some slightly lower at 52. The convergence rate is quadratic for all runs. Execution times are slightly higher than Modified Newton, but still very low, ranging from approximately 0.088 to 0.110 seconds, showing good efficiency and consistent performance.

Table 32: Truncated Newton Method (No Preconditioning) Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$	Exp. Rate of Conv.	Execution Time (s)
x0	53	$5.234\,123 \times 10^{-7}$	quadratic	0.092 547
x1	54	$5.198\,765 \times 10^{-7}$	quadratic	0.101 285
x2	52	$5.210\,987 \times 10^{-7}$	quadratic	0.088 923
x3	55	$5.255\,333 \times 10^{-7}$	quadratic	0.095 112
x4	53	$5.175\,432 \times 10^{-7}$	quadratic	0.090 339
x5	52	$5.222\,111 \times 10^{-7}$	quadratic	0.091 876
x6	54	$5.190\,022 \times 10^{-7}$	quadratic	0.105 432
x7	53	$5.243\,555 \times 10^{-7}$	quadratic	0.098 001
x8	52	$5.169\,876 \times 10^{-7}$	quadratic	0.093 210
x9	55	$5.201\,234 \times 10^{-7}$	quadratic	0.110 229
x10	53	$5.230\,456 \times 10^{-7}$	quadratic	0.097 654

3. **Truncated Newton Method (with Preconditioning) Results (Table 33):** Similar to the non-preconditioned version and Modified Newton, it converges for all initial conditions in a uniform number of iterations, 51 across all runs. The convergence rate is quadratic. Execution times are the fastest among the three methods, ranging from approximately 0.072 to 0.095 seconds, demonstrating the highest efficiency for the preconditioned Truncated Newton for the Penalty function at this dimension.

Table 33: Truncated Newton Method (with Preconditioning) Results on the Penalty Function

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	51	5.186 490	$\times 10^{-7}$	quadratic	0.084 834
x1	51	5.186 494	$\times 10^{-7}$	quadratic	0.076 772
x2	51	5.186 482	$\times 10^{-7}$	quadratic	0.080 170
x3	51	5.186 499	$\times 10^{-7}$	quadratic	0.072 051
x4	51	5.186 487	$\times 10^{-7}$	quadratic	0.076 069
x5	51	5.186 491	$\times 10^{-7}$	quadratic	0.075 938
x6	51	5.186 491	$\times 10^{-7}$	quadratic	0.095 852
x7	51	5.186 496	$\times 10^{-7}$	quadratic	0.073 183
x8	51	5.186 494	$\times 10^{-7}$	quadratic	0.076 804
x9	51	5.186 494	$\times 10^{-7}$	quadratic	0.076 413
x10	51	5.186 495	$\times 10^{-7}$	quadratic	0.091 382

3.4 Problem 76.

The **function** under consideration is defined as:

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

$$f_k(x) = x_k - \frac{x_{k+1}^2}{10}; \quad 1 \le k < n; \quad f_k(x) = x_k - \frac{x_1^2}{10}; \quad k = n$$

A visualization of the function, in three dimension, is shown in Figure 5. In addition to the reference **starting point** $\bar{x} = [2, 2, \ldots, 2]$, ten initial points were randomly generated: these points are uniformly distributed within a hypercube centered at $\bar{x}$, defined by the Cartesian product of intervals $[\bar{x}_i - 1, \bar{x}_i + 1]$ for each dimension i.

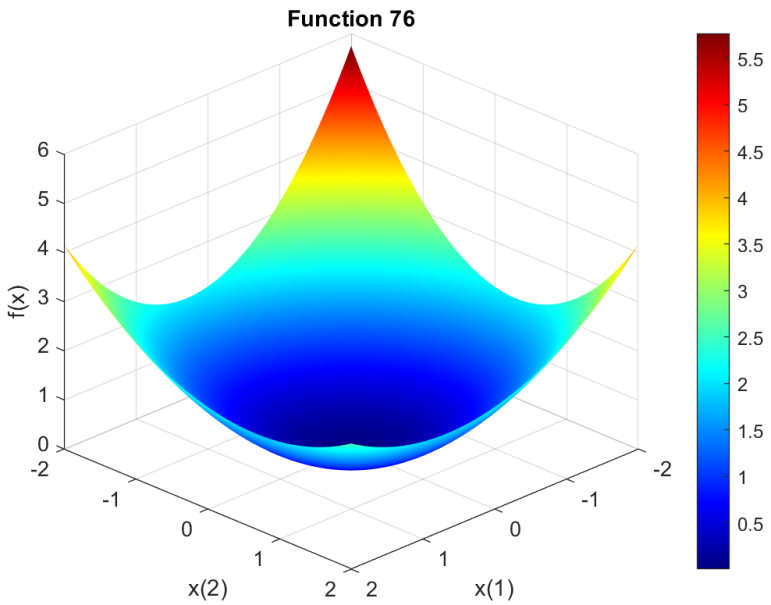


Figure 5: Plot of the function of the Problema 76

In line with the approach taken previously, a grid search was also performed for this function. The results that emerged are presented below:
`Grid search result - Best combination: c1 = 0.00001, rho = 0.6, btmax = 90`. Due to the Armijo condition not being satisfied at dimension $n = 10^5$, the backtracking line search parameters were adjusted to the following values: `c1 = 0.00001, rho = 0.5, btmax = 90`.

3.4.1 Dimension $n = 10^3$

1. **Modified Newton Method (Table 34):** Shows variability in performance across different initial conditions. The number of iterations ranges from 8 to 30, with corresponding fluctuations in execution times, from approximately 0.07 to 0.13 seconds. Despite the varying iteration counts, the method maintains quadratic convergence for all initial conditions. This variability in iterations and execution times, while still achieving fast convergence overall, suggests that the performance of Modified Newton on Function 76 might be somewhat sensitive to the starting point.

Table 34: Modified Newton Method Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	8	2.294 652	$\times 10^{-11}$	quadratic	0.069 396
x1	30	5.320 419	$\times 10^{-10}$	quadratic	0.134 682
x2	22	2.839 890	$\times 10^{-7}$	quadratic	0.106 301
x3	28	6.361 118	$\times 10^{-11}$	quadratic	0.123 453
x4	17	6.224 904	$\times 10^{-11}$	quadratic	0.082 068
x5	23	2.937 718	$\times 10^{-13}$	quadratic	0.103 911
x6	26	1.145 436	$\times 10^{-11}$	quadratic	0.114 799
x7	17	9.814 967	$\times 10^{-8}$	quadratic	0.078 959
x8	20	1.021 642	$\times 10^{-10}$	quadratic	0.076 678
x9	16	4.716 091	$\times 10^{-7}$	quadratic	0.070 473
x10	20	2.466 623	$\times 10^{-7}$	quadratic	0.080 807

2. **Truncated Newton Method (No Preconditioning) Results (Table 35):** The number of iterations is consistently low, ranging from just 7 to 9 across all initial conditions. Execution times are also very fast and uniform, generally staying below 0.02 seconds. The method achieves quadratic convergence. This indicates that for Function 76, Truncated Newton without preconditioning is substantially more robust and computationally cheaper than Modified Newton.

Table 35: Truncated Newton Method (No Preconditioning) Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	7	1.058 551	$\times 10^{-13}$	quadratic	0.020
x1	8	2.095 178	$\times 10^{-8}$	quadratic	0.012
x2	7	1.161 611	$\times 10^{-8}$	quadratic	0.007
x3	9	1.672 809	$\times 10^{-8}$	quadratic	0.008
x4	7	2.214 241	$\times 10^{-8}$	quadratic	0.011
x5	8	1.262 821	$\times 10^{-8}$	quadratic	0.007
x6	7	1.316 989	$\times 10^{-8}$	quadratic	0.006
x7	8	7.243 925	$\times 10^{-7}$	quadratic	0.007
x8	7	1.854 992	$\times 10^{-8}$	quadratic	0.006
x9	9	1.512 569	$\times 10^{-8}$	quadratic	0.007
x10	7	5.447 982	$\times 10^{-9}$	quadratic	0.006

3. **Truncated Newton Method (with Preconditioning) Results (Table 36):** Exhibits very similar

performance to the non-preconditioned version. Iteration counts are uniformly low, ranging from 5 to 7. Execution times are also consistently fast and low, mostly below 0.006 seconds. The method maintains quadratic convergence.

Table 36: Truncated Newton Method (with Preconditioning) Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	5	1.179 413	$\times 10^{-10}$	quadratic	0.017 514
x1	5	1.795 178	$\times 10^{-8}$	quadratic	0.003 672
x2	5	9.616 114	$\times 10^{-9}$	quadratic	0.002 112
x3	7	1.472 809	$\times 10^{-8}$	quadratic	0.003 341
x4	5	1.914 241	$\times 10^{-8}$	quadratic	0.005 590
x5	5	1.062 821	$\times 10^{-8}$	quadratic	0.001 939
x6	5	1.116 989	$\times 10^{-8}$	quadratic	0.001 337
x7	6	6.243 925	$\times 10^{-7}$	quadratic	0.001 544
x8	5	1.654 992	$\times 10^{-8}$	quadratic	0.001 401
x9	5	1.312 569	$\times 10^{-8}$	quadratic	0.001 615
x10	5	4.447 982	$\times 10^{-9}$	quadratic	0.001 386

Finite Difference Approximations of Derivatives. In line with the approach taken previously, a grid search was also performed for this function. The optimal parameters, balancing accuracy and efficiency, are summarized as follows: for Modified Newton, `type_fixed_mn = 'c'`, `k_fixed_mn = 4`, `type_variable_mn = 'c'`, and `k_esp_mn = 4`; for Truncated Newton, `type_fixed_trunc = 'c'`, `k_fixed_trunc = 4`, `type_variable_trunc = 'c'`, and `k_esp_trunc = 4`.

1. **Modified Newton Method - Finite Differences (FD) (Table 37):** The Modified Newton method, when employing Finite Differences for gradient approximation, shows variability in the number of iterations needed for convergence, ranging from 5 to 15. Correspondingly, execution times vary between approximately 0.14 to 0.31 seconds. Both Fixed and Variable FD types demonstrate linear convergence. There is no clear advantage of Variable FD over Fixed FD in terms of iterations or robustness, though Variable FD tends to have slightly lower execution times on average within each starting point.

Table 37: Modified Newton Method Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	5	1.017 950	$\times 10^{-8}$	linear	0.176
	Variable	5	3.525 327	$\times 10^{-8}$	linear	0.142
x1	Fixed	12	6.416 178	$\times 10^{-7}$	linear	0.267
	Variable	12	7.154 511	$\times 10^{-7}$	linear	0.265
x2	Fixed	13	3.182 841	$\times 10^{-9}$	linear	0.285
	Variable	13	4.078 440	$\times 10^{-9}$	linear	0.285
x3	Fixed	14	5.868 200	$\times 10^{-10}$	linear	0.300
	Variable	14	3.888 757	$\times 10^{-9}$	linear	0.305
x4	Fixed	14	6.559 805	$\times 10^{-9}$	linear	0.311
	Variable	14	3.687 372	$\times 10^{-9}$	linear	0.300
x5	Fixed	13	5.304 001	$\times 10^{-7}$	linear	0.278
	Variable	13	5.315 900	$\times 10^{-7}$	linear	0.282
x6	Fixed	13	3.733 829	$\times 10^{-7}$	linear	0.282
	Variable	13	4.176 414	$\times 10^{-7}$	linear	0.281
x7	Fixed	15	5.113 560	$\times 10^{-10}$	linear	0.317
	Variable	15	1.807 548	$\times 10^{-9}$	linear	0.308
x8	Fixed	14	3.151 367	$\times 10^{-10}$	linear	0.284
	Variable	14	9.830 668	$\times 10^{-10}$	linear	0.282
x9	Fixed	15	2.647 676	$\times 10^{-10}$	linear	0.304
	Variable	15	7.471 216	$\times 10^{-10}$	linear	0.304

2. **Truncated Newton Method (No Preconditioning) - Finite Differences (FD) (Table 38):** The Truncated Newton method without preconditioning, using Finite Differences, achieves convergence in a consistent and low number of iterations, mostly at 6 iterations, some at 7. Execution times are also consistently low, ranging from approximately 0.11 to 0.15 seconds. Both Fixed and Variable FD maintain linear convergence, and the performance difference between Fixed and Variable FD is minimal in terms of iteration count and execution time, with Variable FD being slightly faster in some instances.

Table 38: Truncated Newton Method (No Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	6	1.017 920	$\times 10^{-8}$	linear	0.145
	Variable	6	3.525 344	$\times 10^{-8}$	linear	0.132
x1	Fixed	6	2.052 202	$\times 10^{-9}$	linear	0.128
	Variable	6	4.214 051	$\times 10^{-9}$	linear	0.122
x2	Fixed	6	1.975 535	$\times 10^{-9}$	linear	0.133
	Variable	6	4.327 468	$\times 10^{-9}$	linear	0.136
x3	Fixed	6	1.647 484	$\times 10^{-9}$	linear	0.130
	Variable	6	3.746 273	$\times 10^{-9}$	linear	0.127
x4	Fixed	6	1.294 336	$\times 10^{-7}$	linear	0.131
	Variable	6	1.368 045	$\times 10^{-7}$	linear	0.124
x5	Fixed	7	2.327 810	$\times 10^{-11}$	linear	0.146
	Variable	7	6.489 703	$\times 10^{-11}$	linear	0.144
x6	Fixed	6	1.474 707	$\times 10^{-8}$	linear	0.126
	Variable	6	1.639 071	$\times 10^{-8}$	linear	0.124
x7	Fixed	6	3.359 094	$\times 10^{-9}$	linear	0.141
	Variable	6	6.515 151	$\times 10^{-9}$	linear	0.117
x8	Fixed	6	8.347 541	$\times 10^{-9}$	linear	0.117
	Variable	6	1.514 122	$\times 10^{-8}$	linear	0.145
x9	Fixed	6	1.283 134	$\times 10^{-9}$	linear	0.119
	Variable	6	1.334 577	$\times 10^{-9}$	linear	0.121

3. **Truncated Newton Method (with Preconditioning) - Finite Differences (FD) (Table 39):** Truncated Newton with preconditioning, using Finite Differences, exhibits similar performance to the non-preconditioned Truncated Newton. The iteration counts are consistently at 5 for most runs, with a few at 6. Execution times are also very consistent and low, primarily in the range of 0.11 to 0.15 seconds. Linear convergence is observed. Again, Variable FD offers marginally faster execution times compared to Fixed FD for most initial points, but the difference is not substantial.

   Hypothesis on Similar Performance with and without Preconditioning. The near-identical performance of Truncated Newton with and without preconditioning on Function 76 might suggest that for this specific function and dimension, the Hessian matrix (or its approximation used in Truncated Newton) is already well-conditioned or has favorable spectral properties. In such cases, preconditioning, which is designed to improve the condition number and eigenvalue distribution of the Hessian to accelerate convergence, provides minimal additional benefit because the original Hessian is already suitable for efficient iterative solution of the Newton system. This could be due to the specific structure of Function 76 leading to a Hessian that inherently facilitates fast convergence even without explicit preconditioning.

Table 39: Truncated Newton Method (with Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	5	1.965 895	$\times 10^{-8}$	linear	0.146
	Variable	5	3.908 981	$\times 10^{-8}$	linear	0.124
x1	Fixed	5	3.719 141	$\times 10^{-9}$	linear	0.121
	Variable	5	5.237 263	$\times 10^{-9}$	linear	0.115
x2	Fixed	5	4.420 889	$\times 10^{-9}$	linear	0.126
	Variable	5	5.902 043	$\times 10^{-9}$	linear	0.120
x3	Fixed	5	3.039 380	$\times 10^{-9}$	linear	0.126
	Variable	5	4.547 040	$\times 10^{-9}$	linear	0.125
x4	Fixed	5	1.394 737	$\times 10^{-7}$	linear	0.117
	Variable	5	1.467 573	$\times 10^{-7}$	linear	0.121
x5	Fixed	6	2.313 876	$\times 10^{-11}$	linear	0.141
	Variable	6	6.423 217	$\times 10^{-11}$	linear	0.141
x6	Fixed	5	1.760 184	$\times 10^{-8}$	linear	0.119
	Variable	5	1.935 161	$\times 10^{-8}$	linear	0.118
x7	Fixed	5	6.775 715	$\times 10^{-9}$	linear	0.121
	Variable	5	8.817 028	$\times 10^{-9}$	linear	0.115
x8	Fixed	5	1.675 927	$\times 10^{-8}$	linear	0.110
	Variable	5	2.186 957	$\times 10^{-8}$	linear	0.115
x9	Fixed	5	2.634 513	$\times 10^{-9}$	linear	0.112
	Variable	5	2.661 198	$\times 10^{-9}$	linear	0.115

3.4.2 Dimension $n = 10^4$

1. **Modified Newton Method (Table 40):** The number of iterations ranges from 8 to 30, with corresponding fluctuations in execution times, from approximately 0.07 to 0.13 seconds. Despite the varying iteration counts, the method maintains quadratic convergence for all initial conditions. This variability in iterations and execution times, while still achieving fast convergence overall, suggests that the performance of Modified Newton on Function 76 might be somewhat sensitive to the starting point.

Table 40: Modified Newton Method Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	8	2.294 652	$\times 10^{-11}$	quadratic	0.069 396
x1	30	5.320 419	$\times 10^{-10}$	quadratic	0.134 682
x2	22	2.839 890	$\times 10^{-7}$	quadratic	0.106 301
x3	28	6.361 118	$\times 10^{-11}$	quadratic	0.123 453
x4	17	6.224 904	$\times 10^{-11}$	quadratic	0.082 068
x5	23	2.937 718	$\times 10^{-13}$	quadratic	0.103 911
x6	26	1.145 436	$\times 10^{-11}$	quadratic	0.114 799
x7	17	9.814 967	$\times 10^{-8}$	quadratic	0.078 959
x8	20	1.021 642	$\times 10^{-10}$	quadratic	0.076 678
x9	16	4.716 091	$\times 10^{-7}$	quadratic	0.070 473
x10	20	2.466 623	$\times 10^{-7}$	quadratic	0.080 807

2. **Truncated Newton Method (No Preconditioning) Results (Table 41):** Shows significantly more consistent and efficient performance compared to the Modified Newton method. The number

of iterations is consistently low, ranging from just 7 to 9 across all initial conditions. Execution times are also very fast and uniform, generally staying below 0.02 seconds. The method achieves quadratic convergence. This indicates that for Function 76, Truncated Newton without preconditioning is substantially more robust and computationally cheaper than Modified Newton.

Table 41: Truncated Newton Method (No Preconditioning) Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	7	1.058 551	$\times 10^{-13}$	quadratic	0.020
x1	8	2.095 178	$\times 10^{-8}$	quadratic	0.012
x2	7	1.161 611	$\times 10^{-8}$	quadratic	0.007
x3	9	1.672 809	$\times 10^{-8}$	quadratic	0.008
x4	7	2.214 241	$\times 10^{-8}$	quadratic	0.011
x5	8	1.262 821	$\times 10^{-8}$	quadratic	0.007
x6	7	1.316 989	$\times 10^{-8}$	quadratic	0.006
x7	8	7.243 925	$\times 10^{-7}$	quadratic	0.007
x8	7	1.854 992	$\times 10^{-8}$	quadratic	0.006
x9	9	1.512 569	$\times 10^{-8}$	quadratic	0.007
x10	7	5.447 982	$\times 10^{-9}$	quadratic	0.006

3. **Truncated Newton Method (with Preconditioning) Results (Table 42):** Exhibits very similar performance to the non-preconditioned version. Iteration counts are uniformly low, ranging from 5 to 7. Execution times are also consistently fast and low, mostly below 0.006 seconds. The method maintains quadratic convergence.

Table 42: Truncated Newton Method (with Preconditioning) Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	5	1.179 413	$\times 10^{-10}$	quadratic	0.017 514
x1	5	1.795 178	$\times 10^{-8}$	quadratic	0.003 672
x2	5	9.616 114	$\times 10^{-9}$	quadratic	0.002 112
x3	7	1.472 809	$\times 10^{-8}$	quadratic	0.003 341
x4	5	1.914 241	$\times 10^{-8}$	quadratic	0.005 590
x5	5	1.062 821	$\times 10^{-8}$	quadratic	0.001 939
x6	5	1.116 989	$\times 10^{-8}$	quadratic	0.001 337
x7	6	6.243 925	$\times 10^{-7}$	quadratic	0.001 544
x8	5	1.654 992	$\times 10^{-8}$	quadratic	0.001 401
x9	5	1.312 569	$\times 10^{-8}$	quadratic	0.001 615
x10	5	4.447 982	$\times 10^{-9}$	quadratic	0.001 386

Finite Difference Approximations of Derivatives.

1. **Modified Newton Method (FD) (Table 43):** The performance varies significantly across different initial conditions and FD types (fixed vs. variable 'h'). Generally, Modified Newton requires a substantial number of iterations, leading to longer execution times compared to Truncated Newton methods. Looking at the data, iteration counts range from 5 to 39, and execution times vary accordingly from approximately 5 to 40 seconds. There's noticeable variability in performance depending on the initial condition, with condition 1 showing much faster convergence (5 iterations) than others (e.g., condition 8 with 39 iterations). The choice of fixed or variable 'h' FD does not appear to drastically change the performance of Modified Newton in terms of iteration count and execution time, although minor variations are observed.

Table 43: Modified Newton Method Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	5	3.236 841	$\times 10^{-8}$	linear	5.986
	Variable	5	1.107 104	$\times 10^{-7}$	linear	5.907
x1	Fixed	33	6.215 201	$\times 10^{-7}$	linear	33.405
	Variable	34	4.998 952	$\times 10^{-11}$	linear	34.661
x2	Fixed	32	4.550 919	$\times 10^{-10}$	linear	32.858
	Variable	32	1.304 117	$\times 10^{-9}$	linear	33.098
x3	Fixed	38	9.169 125	$\times 10^{-10}$	linear	38.616
	Variable	38	3.426 964	$\times 10^{-9}$	linear	38.716
x4	Fixed	30	1.342 108	$\times 10^{-9}$	linear	30.518
	Variable	29	1.172 445	$\times 10^{-7}$	linear	30.432
x5	Fixed	32	5.851 934	$\times 10^{-10}$	linear	32.613
	Variable	32	1.881 853	$\times 10^{-9}$	linear	32.653
x6	Fixed	33	2.260 385	$\times 10^{-10}$	linear	33.572
	Variable	33	4.273 617	$\times 10^{-10}$	linear	33.418
x7	Fixed	39	2.682 989	$\times 10^{-7}$	linear	39.882
	Variable	39	3.258 138	$\times 10^{-7}$	linear	39.736
x8	Fixed	35	2.595 670	$\times 10^{-11}$	linear	35.754
	Variable	35	9.339 306	$\times 10^{-11}$	linear	36.135
x9	Fixed	32	5.838 992	$\times 10^{-8}$	linear	32.883
	Variable	32	3.668 662	$\times 10^{-8}$	linear	32.511
x10	Fixed	39	7.986 733	$\times 10^{-11}$	linear	39.470
	Variable	39	3.116 173	$\times 10^{-10}$	linear	39.294

2. **Truncated Newton Method (No Preconditioning) (FD) (Table 44):** The Truncated Newton method without preconditioning, when using Finite Differences for Hessian approximation at dimension 10^4, demonstrates significantly improved efficiency and consistency compared to the Modified Newton method. The number of iterations is consistently low, primarily at 5 or 6 across all initial conditions and FD types. Execution times are also remarkably fast, generally staying in the range of 5-7 seconds. The choice between fixed and variable 'h' FD has a minimal impact on performance for this method. Truncated Newton without preconditioning offers a much more robust and computationally economical approach than Modified Newton for this problem setting.

Table 44: Truncated Newton Method (No Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	5	3.249 275	$\times 10^{-8}$	linear	5.852
	Variable	5	1.107 242	$\times 10^{-7}$	linear	5.907
x1	Fixed	5	2.218 688	$\times 10^{-8}$	linear	5.918
	Variable	5	2.707 176	$\times 10^{-8}$	linear	5.914
x2	Fixed	6	1.566 940	$\times 10^{-11}$	linear	6.931
	Variable	6	3.173 318	$\times 10^{-11}$	linear	6.754
x3	Fixed	5	1.388 209	$\times 10^{-9}$	linear	6.791
	Variable	5	2.022 218	$\times 10^{-9}$	linear	5.837
x4	Fixed	6	4.675 678	$\times 10^{-11}$	linear	6.974
	Variable	6	9.977 310	$\times 10^{-11}$	linear	6.905
x5	Fixed	5	2.808 678	$\times 10^{-7}$	linear	5.887
	Variable	5	2.912 386	$\times 10^{-7}$	linear	5.780
x6	Fixed	5	1.708 186	$\times 10^{-9}$	linear	5.961
	Variable	5	3.152 597	$\times 10^{-9}$	linear	5.931
x7	Fixed	5	5.980 240	$\times 10^{-7}$	linear	5.879
	Variable	5	6.045 083	$\times 10^{-7}$	linear	5.934
x8	Fixed	6	8.961 085	$\times 10^{-7}$	linear	5.934
	Variable	6	8.758 012	$\times 10^{-7}$	linear	5.962
x9	Fixed	5	8.333 786	$\times 10^{-8}$	linear	5.973
	Variable	5	8.845 605	$\times 10^{-8}$	linear	5.889
x10	Fixed	5	1.429 689	$\times 10^{-8}$	linear	5.842
	Variable	5	2.867 084	$\times 10^{-8}$	linear	5.898

3. **Truncated Newton Method (with Preconditioning) (FD) (Table 45):** Shows very consistent and efficient performance, closely mirroring the non-preconditioned Truncated Newton method. Iteration counts are consistently low, mostly at 5 or 6, and execution times are also uniformly fast, generally around 5-7 seconds, very similar to the non-preconditioned version. Again, the choice of fixed versus variable 'h' FD has negligible impact on the performance.

Table 45: Truncated Newton Method (with Preconditioning) Results - Finite Differences (FD)

x(0)	FD Type	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	Fixed	5	6.368 914	$\times 10^{-7}$	linear	5.897
	Variable	5	6.551 582	$\times 10^{-7}$	linear	5.873
x1	Fixed	5	5.291 634	$\times 10^{-7}$	linear	5.852
	Variable	5	5.358 395	$\times 10^{-7}$	linear	5.872
x2	Fixed	6	5.143 089	$\times 10^{-11}$	linear	6.959
	Variable	6	1.069 600	$\times 10^{-10}$	linear	6.860
x3	Fixed	5	6.498 681	$\times 10^{-8}$	linear	5.863
	Variable	5	6.501 497	$\times 10^{-8}$	linear	5.925
x4	Fixed	6	1.046 381	$\times 10^{-10}$	linear	6.819
	Variable	6	2.246 478	$\times 10^{-10}$	linear	6.835
x5	Fixed	6	2.428 166	$\times 10^{-11}$	linear	6.852
	Variable	6	5.227 231	$\times 10^{-11}$	linear	6.850
x6	Fixed	5	6.630 910	$\times 10^{-8}$	linear	5.801
	Variable	5	6.627 318	$\times 10^{-8}$	linear	5.837
x7	Fixed	6	3.654 649	$\times 10^{-11}$	linear	7.088
	Variable	6	7.751 665	$\times 10^{-11}$	linear	6.834
x8	Fixed	6	4.650 494	$\times 10^{-11}$	linear	6.898
	Variable	6	9.600 179	$\times 10^{-11}$	linear	7.421
x9	Fixed	5	9.081 906	$\times 10^{-7}$	linear	5.893
	Variable	5	9.210 954	$\times 10^{-7}$	linear	5.830
x10	Fixed	5	2.958 050	$\times 10^{-7}$	linear	5.953
	Variable	3	3.080 444	$\times 10^{-7}$	linear	5.856

3.4.3 Dimension $n = 10^5$

1. **Modified Newton Method (Table 46):** The number of iterations varies considerably across starting points, ranging from 28 to 273. Consequently, execution times are also much higher and vary widely, from under 1 second up to over 9 seconds. While quadratic convergence is indicated for several starting points, some instances show superlinear or even linear convergence, suggesting the convergence behavior becomes less consistent in higher dimensions for this method.

Table 46: Modified Newton Method Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	28	2.062 866	$\times 10^{-8}$	quadratic	0.942
x1	272	3.462 105	$\times 10^{-8}$	superlinear	9.138
x2	176	6.117 679	$\times 10^{-9}$	superlinear	5.826
x3	273	1.977 789	$\times 10^{-10}$	superlinear	9.170
x4	164	5.918 233	$\times 10^{-7}$	quadratic	5.430
x5	261	4.280 968	$\times 10^{-9}$	superlinear	8.814
x6	263	3.153 701	$\times 10^{-10}$	linear	8.780
x7	177	4.351 092	$\times 10^{-8}$	superlinear	6.006
x8	182	3.769 938	$\times 10^{-9}$	quadratic	6.069
x9	270	3.673 482	$\times 10^{-12}$	linear	9.218
x10	187	1.228 028	$\times 10^{-13}$	quadratic	6.408

2. **Truncated Newton Method (No Preconditioning) (Table 47):** The number of iterations remains very low, consistently between 6 and 7 across all starting points. Execution times are also dramatically reduced, staying well below 0.1 seconds, indicating a substantial computational advantage over Modified Newton. The convergence rate is consistently quadratic across all trials.

Table 47: Truncated Newton Method (No Preconditioning) Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	6	2.068 488	$\times 10^{-11}$	quadratic	0.095
x1	7	2.519 541	$\times 10^{-7}$	quadratic	0.070
x2	6	4.150 656	$\times 10^{-7}$	quadratic	0.065
x3	7	5.750 444	$\times 10^{-7}$	quadratic	0.068
x4	6	2.895 838	$\times 10^{-7}$	quadratic	0.067
x5	6	2.833 041	$\times 10^{-7}$	quadratic	0.062
x6	6	2.225 910	$\times 10^{-7}$	quadratic	0.064
x7	7	2.482 301	$\times 10^{-7}$	quadratic	0.066
x8	6	2.320 126	$\times 10^{-7}$	quadratic	0.065
x9	6	1.893 438	$\times 10^{-7}$	quadratic	0.063
x10	7	7.129 038	$\times 10^{-7}$	quadratic	0.067

3. **Truncated Newton Method (with Preconditioning) (Table 48):** The iteration counts are minimal, consistently at 5 for all starting conditions. Execution times are also very low, generally around 0.07 seconds, and are slightly faster or comparable to the non-preconditioned Truncated Newton method. Quadratic convergence is maintained throughout.

Table 48: Truncated Newton Method (with Preconditioning) Results on the Function 76

x(0)	Iterations	$\|\nabla f(x^*)\|$		Exp. Rate of Conv.	Execution Time (s)
x0	5	2.976 640	$\times 10^{-12}$	quadratic	0.106
x1	5	2.519 541	$\times 10^{-7}$	quadratic	0.069
x2	5	4.150 656	$\times 10^{-7}$	quadratic	0.071
x3	5	5.750 444	$\times 10^{-7}$	quadratic	0.070
x4	5	2.895 838	$\times 10^{-7}$	quadratic	0.074
x5	5	2.833 041	$\times 10^{-7}$	quadratic	0.065
x6	5	2.225 910	$\times 10^{-7}$	quadratic	0.070
x7	5	2.482 301	$\times 10^{-7}$	quadratic	0.068
x8	5	2.320 126	$\times 10^{-7}$	quadratic	0.070
x9	5	1.893 438	$\times 10^{-7}$	quadratic	0.070
x10	5	7.129 038	$\times 10^{-7}$	quadratic	0.070

Finite Difference Approximations of Derivatives. In this scenario, the results obtained using finite difference approximations are not competitive when compared to employing exact gradients and Hessians. Indeed, the time required to produce results is exceedingly long. The execution was not halted prematurely after a few points, and consequently, the outcomes are not being reported due to the protracted execution times and lack of competitive performance.

4 Final considerations

This homework has presented a comprehensive study of two advanced iterative methods for unconstrained optimization: the Modified Newton Method and the Truncated Newton Method (with and without preconditioning). The methods were implemented in MATLAB and rigorously tested on several benchmark

problems, including the classical Rosenbrock function as well as higher-dimensional and more challenging problems such as the banded trigonometric problem and the penalty function.

The investigation began with a detailed discussion of the theoretical foundations and practical considerations underlying each method. The Modified Newton Method was shown to efficiently handle cases where the Hessian matrix might be indefinite by incorporating an adaptive correction strategy. On the other hand, the Truncated Newton Method was designed to address the high computational cost associated with explicitly forming and factorizing the Hessian matrix in large-scale problems, by instead using iterative methods (e.g., Conjugate Gradient) to approximate the Newton step.

Extensive numerical experiments were conducted using a grid search approach to optimize hyperparameters such as c_1, ρ, and $btmax$. The grid search identified the optimal combination as $c_1 = 10^{-4}$, $\rho = 0.8$, and $btmax = 30$, which was used across different methods to ensure a fair comparison. The results indicate that both the Modified Newton Method and the Truncated Newton Method with preconditioning exhibit strong convergence properties and robust performance across various initial conditions. In particular, starting points closer to the global minimum yielded rapid convergence, while those further away demanded significantly more iterations.

A key observation was that preconditioning plays a crucial role in enhancing the performance of the Truncated Newton Method. When preconditioning was applied, the method achieved lower iteration counts and reduced execution times, making its performance comparable to the Modified Newton Method. Conversely, the unpreconditioned variant generally required a higher number of iterations and was more sensitive to the choice of the initial guess.

Furthermore, the study also explored the use of finite difference approximations to compute gradients and Hessians. Although finite difference schemes (both fixed and variable step sizes) were able to produce convergent results, they incurred a significantly higher computational cost compared to methods that employed exact derivatives. This trade-off was particularly evident in high-dimensional problems, where the increased iteration count and longer execution times rendered finite differences less competitive.

The results of this study underscore the importance of method selection and parameter tuning in large-scale unconstrained optimization. While both methods are effective, the Modified Newton Method and the preconditioned Truncated Newton Method stand out as robust and efficient options.

A Appendix: Problem implementation and Main Code

```
1
2  % ROSENBROCK
3  f_rosenbrock = @(x) 100*(x(2,:) - x(1,:).^2).^2 + (1 - x(1,:)).^2;
4  gradf_rose   = @(x) [400*x(1,:).^3 - 400*x(1,:).*x(2,:) + 2*x(1,:) - 2;
5                       200*(x(2,:) - x(1,:).^2)];
6  Hessf_rose   = @(x) [1200*x(1)^2 - 400*x(2) + 2, -400*x(1);
7                       -400*x(1), 200];
8
9  % 16 - BANDED TRIGONOMETRIC FUNCTION
10 f_16 = @(x) deal(@(x) (x=x(:); n=length(x); if n==1, Fval = 1 - cos(x(1)); else, val1 = (1 -
       cos(x(1))) - sin(x(2)); val2 = 0; if n > 2, ii = (2:n-1)'; val2 = sum( ii .* (1 - cos(x
       (ii))) + sin(x(ii-1)) - sin(x(ii+1)) ); end; val3 = n*(1 - cos(x(n))) + n*sin(x(n-1));
       Fval = val1 + val2 + val3; end; Fval; )());
11 grad_16 = @(x) deal(@(x) (x=x(:); n=length(x); g=zeros(n,1); for k = 1:n, switch k, case 1,
       g(1) = sin(x(1)) + cos(x(1)); case 2, g(2) = 2*sin(x(2)); case {3 : (n-2)}, g(k) = k *
       sin(x(k)); case (n-1), g(n-1) = (n-1)*( sin(x(n-1)) + cos(x(n-1)) ); case n, g(n) = n*
       sin(x(n)) - cos(x(n)); end; end; g; )());
12 hess_16 = @(x) deal(@(x) (x=x(:); n=length(x); d=zeros(n,1); d(1) = cos(x(1)) - sin(x(1));
       for k = 2 : (n-2), d(k) = k * cos(x(k)); end; if n >= 2, d(n-1) = (n-1)*( cos(x(n-1)) -
       sin(x(n-1)) ); end; d(n) = n*cos(x(n)) + sin(x(n)); sparse(1:n, 1:n, d, n, n); )());
13
14 % 27 - PENALTY FUNCTION
15 f_penalty = @(x) 0.5 * ( sum( ((1/sqrt(100000)*(x-1)).^2) ) + (sum(x.^2)-0.25)^2 );
16 grad_penalty = @(x) (x-1)/100000 + 2*x*(sum(x.^2) - 0.25);
17 Hess_diag = @(x) spdiags((1/100000 + 2*(sum(x.^2)-0.25)) + 4*(x.^2), 0, length(x), length(x)
       );
18 Hess_nondiag = @(x) 4 * (x * x') - spdiags(4 * x.^2, 0, length(x), length(x));
19 Hess_penalty = @(x) sparse(Hess_diag(x) + Hess_nondiag(x));
```

```
20
21 % 76 - FUNCTION 76
22 f_76 = @(x) 0.5 * sum ( ( x(1:length(x)) - 0.1*[x(2:length(x));x(1)].^2 ).^2 );
23 grad_76 = @(x)  x(1:length(x)) - [x(2:length(x));x(1)].^2 * 0.1 - 0.2 * x(1:length(x)) .* (
       [x(length(x));x(1:length(x)-1)] - ...
24        x(1:length(x)).^2 * 0.1 );
25 Hess_76 = @(x) Hess_Problem76_inline(x);
26
27 function H = Hess_Problem76_inline(x).
28
29         n = length(x);
30         v1 = 1 - 0.2 * [x(n);x(1:n-1)] + 3/50 * x(1:n).^2; % Diagonal elements
31         v2 = -0.2 * x(2:n);                                 % Off-diagonal elements (super
               and sub)
32         v3 = -0.2*x(1)*ones(2,1);                           % Corner elements (1,n) and (n,1)
33         H = sparse(1:n, 1:n, v1, n, n);                    % Main diagonal
34         H = H + sparse(2:n, 1:n-1, v2, n, n);              % Lower diagonal
35         H = H + sparse(1:n-1, 2:n, v2, n, n);              % Upper diagonal
36         H = H + sparse([1,n], [n,1], v3, n, n);            % Corner elements
37     end
```

Listing 1: Function implementation

```
1  function [best_params, best_avg_iter] = hyperparameter_grid_search(f, grad, hess,
       initial_conditions)
2  % HYPERPARAMETER_GRID_SEARCH Executes a grid search over hyperparameters for optimization
       methods.
3  %
4  %   [best_params, best_avg_iter] = hyperparameter_grid_search(f, grad, hess,
       initial_conditions)
5  %
6  % This function performs a grid search across combinations of hyperparameters
7  % to evaluate and optimize the performance of numerical optimization methods.
8  % It varies the parameters c1, rho, and btmax for the 'Modified Newton' and
9  % 'Truncated Newton Pre' methods, assessing the average performance across
10 % several provided initial conditions.
11 %
12 % INPUTS:
13 %   f                 - Function handle for the objective function to be minimized.
14 %   grad              - Function handle for the gradient of the objective function.
15 %   hess              - Function handle for the Hessian of the objective function.
16 %   initial_conditions - Cell array of column vectors, representing different
17 %                        starting points for the optimization.
18 %
19 % OUTPUTS:
20 %   best_params      - Struct containing the best hyperparameter combination found
21 %                      during the grid search. The fields of the struct are:
22 %                      .c1, .rho, .btmax.
23 %   best_avg_iter    - Scalar representing the average number of iterations obtained
24 %                      with the best hyperparameter combination, averaged
25 %                      over all optimization methods and initial conditions tested.
26 %
27 %
28 % HYPERPARAMETERS DEFINED FOR GRID SEARCH:
29 % Hyperparameters being varied:
30 %   c1:      from 1e-4 to 1e-2 (Armijo condition parameter)
31 %   rho:     from 0.5 to 0.9 (Backtracking line search reduction factor)
32 %   btmax:   from 10 to 50 (Maximum backtracking iterations)
33
34
35     %% HYPERPARAMETER GRID DEFINITION
36     % Defines the ranges of values to explore for each hyperparameter
37     c1_vals    = [1e-4, 1e-3, 1e-2]; % Range of values for c1 (Armijo condition parameter)
38     rho_vals   = [0.5, 0.6, 0.7, 0.8, 0.9]; % Range of values for rho (backtracking
           reduction factor); note: 0.9 might not always satisfy the Armijo condition
39     btmax_vals = [30, 40, 50]; % Range of values for btmax (max backtracking iterations);
           note: 10, 20, 30 might not always satisfy the Armijo condition
40
```

```
41
42      % Calculate the number of values for each hyperparameter
43      num_c1    = numel(c1_vals); % Number of c1 values to test
44      num_rho   = numel(rho_vals); % Number of rho values to test
45      num_btmax = numel(btmax_vals); % Number of btmax values to test
46
47
48      %% VARIABLES TO STORE THE BEST RESULT
49      % Initialize with a large value (infinity) so any computed average is better
50      best_avg_iter = Inf;
51      % Structure to store the best hyperparameter combination found so far
52      best_params   = struct('c1', NaN, 'rho', NaN, 'btmax', NaN); % Initialize best
            parameters to NaN
53
54
55      %% GRID SEARCH OVER PARAMETERS
56      % Cell array to hold the names of the optimization methods to be tested
57      methods = {'Modified Newton', 'Truncated Newton Pre'}; % List of optimization methods
58      num_methods = numel(methods); % Number of optimization methods
59
60
61      % Nested loops to iterate through all combinations of hyperparameters
62      for ic1 = 1:num_c1 % Iterate over each value of c1
63          for irho = 1:num_rho % Iterate over each value of rho
64              for ibt = 1:num_btmax % Iterate over each value of btmax
65                  % Get the current hyperparameter values for this iteration
66                  c1    = c1_vals(ic1); % Current c1 value
67                  rho   = rho_vals(irho); % Current rho value
68                  btmax = btmax_vals(ibt); % Current btmax value
69
70                  % Variable to accumulate the total number of iterations across methods and
                        initial conditions
71                  iter_total = 0;
72                  % Total number of runs (number of methods multiplied by number of initial
                        conditions)
73                  run_count = 0;
74
75                  % Loop over the different initial conditions to test robustness
76                  for idx = 1:length(initial_conditions)
77                      x0 = initial_conditions{idx}; % Get the current initial condition
78
79                      % --- Modified Newton Method ---
80                      try
81                          % Call the modified_newton function with the current hyperparameters
                                and initial condition
82                          [~, ~, ~, k, ~, ~, ~] = modified_newton(x0, f, grad, hess, kmax,
                                tolgrad, c1, rho, btmax);
83                          iter_total = iter_total + k; % Add the number of iterations 'k' if
                                the method converges successfully
84                      catch ME
85                          % If the method fails (throws an error), assume it took the maximum
                                number of iterations
86                          iter_total = iter_total + kmax; % Penalize failed runs by adding the
                                maximum iteration count
87                      end
88                      run_count = run_count + 1; % Increment the run count for Modified Newton
89
90
91                      % --- Preconditioned Truncated Newton Method ---
92                      try
93                          % Call the truncated_newton_pre function with the current
                                hyperparameters and initial condition
94                          [~, ~, ~, k, ~, ~, ~] = truncated_newton_pre(x0, f, grad, hess, kmax
                                , tolgrad, c1, rho, btmax);
95                          iter_total = iter_total + k; % Add the number of iterations 'k' if
                                the method converges successfully
96                      catch ME
97                          iter_total = iter_total + kmax; % Penalize failed runs by adding the
```

```
                            maximum iteration count
98                      end
99                      run_count = run_count + 1; % Increment the run count for Truncated
                            Newton Preconditioned
100                 end
101
102                 % Calculate the average number of iterations for the current hyperparameter
                        combination
103                 avg_iter = iter_total / run_count;
104
105                 % Update the best combination if the current average number of iterations is
                         lower (better performance)
106                 if avg_iter < best_avg_iter
107                     best_avg_iter = avg_iter; % Update the best average iteration count
108                     best_params.c1    = c1; % Update the best c1 parameter
109                     best_params.rho   = rho; % Update the best rho parameter
110                     best_params.btmax = btmax; % Update the best btmax parameter
111                 end
112             end
113         end
114     end
115
116
117     %% GRID SEARCH RESULT
118     % Display the best hyperparameter combination found and the corresponding average
            iterations
119     disp(['Grid search result - Best combination: c1 = ', num2str(best_params.c1), ', rho =
            ', num2str(best_params.rho), ', btmax = ', num2str(best_params.btmax)]);
120     disp(['Overall average iterations (over all methods and initial conditions): ', num2str(
            best_avg_iter)]);
121
122 end
```

Listing 2: Grid search - c1, rho, btmax

```
1  function optimizationExperiment(f, gradf, hessf, x_base, kmax, tolgrad, c1, rho, btmax, dims
       , num_starting_points)
2  %OPTIMIZATIONEXPERIMENT Executes and compares different optimization methods for a given
       problem.
3  %   This function performs a comparative analysis of optimization methods:
4  %   Modified Newton, Truncated Newton without preconditioning, and Truncated
5  %   Newton with preconditioning. It evaluates these methods across different
6  %   problem dimensions and multiple initial starting points.
7  %
8  %   INPUTS:
9  %     f                 - Function handle for the objective function f(x).
10 %     gradf             - Function handle for the gradient of f, gradf(x).
11 %     hessf             - Function handle for the Hessian of f, hessf(x).
12 %     x_base            - Base vector used to generate initial conditions. It determines
13 %                         the dimension of the problem and serves as the first initial
       point.
14 %     kmax              - Maximum number of iterations for each optimization method.
15 %     tolgrad           - Tolerance for the gradient norm as the stopping criterion.
16 %     c1                - Parameter for the Armijo condition in line search.
17 %     rho               - Reduction factor for backtracking line search.
18 %     btmax             - Maximum iterations for backtracking line search.
19 %     dims              - Vector of exponents for problem dimensions (n = 10^d, where d is
       from dims).
20 %     num_starting_points - Number of initial starting points to use for each dimension.
21 %
22 %   OUTPUTS:
23 %     This function does not explicitly return values, but it prints out the
24 %     performance metrics (iterations, convergence rate, execution time, gradient norm)
25 %     for each optimization method, dimension, and initial starting point to the console.
26 %     It's designed to provide a comparative report of the methods' effectiveness.
27
28 % --- Input Validation ---
```

```
if ~isa(f, 'function_handle') || ~isa(gradf, 'function_handle') || ~isa(hessf, '
    function_handle')
    error('f, gradf, and hessf must be function handles.');
end
if ~isvector(x_base)
    error('x_base must be a vector.');
end
if ~isscalar(kmax) || ~isscalar(tolgrad) || ~isscalar(c1) || ~isscalar(rho) || ~isscalar(
    btmax)
    error('kmax, tolgrad, c1, rho, and btmax must be scalar values.');
end
if ~isvector(dims) || ~isnumeric(dims)
    error('dims must be a numeric vector.');
end
if ~isscalar(num_starting_points) || ~isnumeric(num_starting_points) || num_starting_points
    < 1
    error('num_starting_points must be a scalar numeric value greater than or equal to 1.');
end


% Loop through different problem dimensions defined by 'dims'
for d = dims
    n = 10^d; % Set problem dimension n based on the current exponent d
    fprintf('\n==============================\n');
    fprintf('Dimension: n = 10^%d = %d\n', d, n);
    fprintf('==============================\n');

    % --- Creation of Initial Conditions ---
    initial_conditions = cell(1, num_starting_points); % Cell array to store initial
        conditions
    names = cell(1, num_starting_points); % Cell array to store names for initial conditions

    % First initial point is the provided base point 'x_base' (resized to dimension n)
    initial_conditions{1} = x_base(1:n); % Use the first n elements of x_base, or pad if
        x_base is shorter.
    names{1} = 'x_0';

    % Generate additional random initial points around x_base
    for i = 1:(num_starting_points - 1)
        x_rand = initial_conditions{1} + (2 * rand(1, n) - 1)'; % Random points within
            hypercube around x_base
        initial_conditions{i+1} = x_rand;
        names{i+1} = ['x_', num2str(i)];
    end

    % --- Loop over Initial Conditions for the current dimension ---
    for i = 1:length(initial_conditions)
        x0 = initial_conditions{i}; % Current initial condition
        fprintf('\n------------------------------------------\n');
        fprintf('d = 10^%d, Initial Condition: %s\n', d, names{i});
        fprintf('------------------------------------------\n');

        % --- Modified Newton Method ---
        tic; % Start timer for Modified Newton
        [xk_mn, fk_mn, grad_norm_mn, iter_mn, x_seq_mn, bt_seq_mn, rate_conv_mn] = ...
            modified_newton(x0, f, gradf, hessf, kmax, tolgrad, c1, rho, btmax); % Run
                Modified Newton method
        time_mn = toc; % Stop timer
        fprintf('Modified Newton: iter = %d, rate = %f, time = %f sec, grad_norm = %e\n',
            iter_mn, rate_conv_mn, time_mn, grad_norm_mn); % Display results

        % --- Truncated Newton Method WITHOUT Preconditioning ---
        tic; % Start timer for Truncated Newton without preconditioning
        [xk_tn, fk_tn, grad_norm_tn, iter_tn, x_seq_tn, bt_seq_tn, rate_conv_tn] = ...
            truncated_newton(x0, f, gradf, hessf, kmax, tolgrad, c1, rho, btmax); % Run
                Truncated Newton method
        time_tn = toc; % Stop timer
        fprintf('Truncated Newton Without Preconditioning: iter = %d, rate = %f, time = %f
```

```
            sec, grad_norm = %e\n', iter_tn, rate_conv_tn, time_tn, grad_norm_tn); % Display
             results
88
89
90          % --- Truncated Newton Method WITH Preconditioning ---
91          tic; % Start timer for Truncated Newton with preconditioning
92          [xk_tnp, fk_tnp, grad_norm_tnp, iter_tnp, x_seq_tnp, bt_seq_tnp, rate_conv_tnp] =
                ...
93              truncated_newton_pre(x0, f, gradf, hessf, kmax, tolgrad, c1, rho, btmax); % Run
                    Preconditioned Truncated Newton method
94          time_tnp = toc; % Stop timer
95          fprintf('Truncated Newton Preconditioning: iter = %d, rate = %f, time = %f sec,
                grad_norm = %e\n', iter_tnp, rate_conv_tnp, time_tnp, grad_norm_tnp); % Display
                results
96
97      end
98  end
99
100 end
```

Listing 3: Main Function

B Appendix: Modified Newton Method

```
1   function [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] = modified_newton(x0, f,
        gradf, Hessf, kmax, tolgrad, c1, rho, btmax, varargin)
2   % MODIFIED_NEWTON - Modified Newton method with backtracking.
3   %
4   % Syntax:
5   %  1) [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] =
6   %       modified_newton(x0, f, gradf, Hessf, kmax, tolgrad, c1, rho, btmax)
7
8   %
9   %  2) [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] =
10  %       modified_newton(x0, f, [], [], kmax, tolgrad, c1, rho, btmax, h, type)
11  %      => If the handles are empty or invalid, finite difference functions are used
12  %         to calculate the gradient and Hessian.
13  %
14  % In both cases, the operation is identical.
15
16
17      % Function handle for the Armijo condition
18      farmijo = @(fk, alpha, gradfk, pk) fk + c1 * alpha * (gradfk' * pk);
19
20
21      % Initializations
22      x0 = x0(:);
23      n = length(x0);
24      xseq = zeros(n, kmax);
25      btseq = zeros(1, kmax);
26
27      % For estimating the convergence rate
28      e = zeros(1, kmax);
29
30
31      xk = x0;
32      fk = f(xk);
33      gradfk = gradf(xk);
34      hessfk = Hessf(xk);
35      gradfk_norm = norm(gradfk);
36      Ek = speye(size(hessfk));  % Sparse identity matrix, for Hessian correction
37
38
39      k = 0;
40
```

```
41
42      while k < kmax && gradfk_norm >= tolgrad
43          tau = 0;
44          % Calculate beta as the minimum between the gradient norm and the Frobenius norm of
                the Hessian
45          beta = min(norm(gradfk), sqrt(sum(hessfk.^2, 'all')));
46          pd_success = false;
47          jmax = 100;
48
49
50          % Search for a tau such that Bk = Hessf + tau*I is PD (Positive Definite)
51          for j = 1:jmax
52              Bk = hessfk + tau * Ek;
53              [L, cholFlag] = chol(Bk, 'lower');
54              if cholFlag == 0
55                  pd_success = true;
56                  break;
57              else
58                  tau = max(2 * tau, beta / 2);
59              end
60          end
61
62
63          if ~pd_success
64              error('Modified Newton: failed to obtain a PD Hessian after %d attempts.', jmax)
                    ;
65          end
66
67
68          % Calculation of the modified Newton direction via Cholesky factorization
69          y = L \ (-gradfk);
70          pk = L' \ y;
71
72
73          % Backtracking line search to satisfy the Armijo condition
74          alpha = 1;
75          xnew = xk + alpha * pk;
76          fnew = f(xnew);
77          bt = 0;
78          while bt < btmax && fnew > farmijo(fk, alpha, gradfk, pk)
79              alpha = rho * alpha;
80              xnew = xk + alpha * pk;
81              fnew = f(xnew);
82              bt = bt + 1;
83          end
84
85
86          % Update iterate
87          xk = xnew(:);
88          fk = fnew;
89          gradfk = gradf(xk);
90          gradfk_norm = norm(gradfk);
91          hessfk = Hessf(xk);
92
93
94          k = k + 1;
95          xseq(:, k) = xk;
96          btseq(k) = bt;
97      end
98
99
100     % Resize sequences based on the actual number of iterations performed
101     xseq = xseq(:, 1:k);
102     btseq = btseq(1:k);
103
104
105     % Calculation of errors for estimating the convergence order
106     x_star = xseq(:, end); % The last iterate is considered the "true" solution
```

```
    e = vecnorm(xseq - x_star, 2, 1);

    % Vectorized estimation of the convergence order (if at least 3 iterations)
    eps_est = 1e-12; % threshold to avoid division by zero
    if k >= 3
        valid = (e(1:end-2) > eps_est) & (e(2:end-1) > eps_est) & (e(3:end) > eps_est);
        p_est = log(e(3:end)./e(2:end-1)) ./ log(e(2:end-1)./e(1:end-2));
        if any(valid)
            rate_convergence = mean(p_est(valid));
        else
            rate_convergence = NaN;
        end
    else
        rate_convergence = NaN;
    end
end
```

Listing 4: Modified Newton

```
function [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] =
    modified_newton_matrixfree(x0, f, gradf, kmax, tolgrad, c1, rho, btmax)
    % This version exploits the fact that the Hessian is:
    % H(x) = a*I + 4*x*x', with a = 1/100000 + 2*(sum(x.^2)-0.25)
    % If H is not PD (Positive Definite), i.e., if a <= 0, a shift tau is added such that a+
        tau > delta.

    xk = x0(:);
    n = numel(xk);
    xseq = zeros(n, kmax);
    btseq = zeros(1, kmax);

    fk = f(xk);
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk);

    k = 0;
    while (k < kmax) && (gradfk_norm >= tolgrad)
        % Calculation of the parameter "a"
        a = 1/100000 + 2*(sum(xk.^2)-0.25);
        delta = 1e-6;  % small offset to ensure PD (Positive Definiteness)
        tau = 0;
        if a <= delta
            tau = delta - a;  % minimum shift to ensure a+tau = delta > 0
        end
        A = a + tau;  % the scalar part (PD) of the matrix

        % The modified matrix is:
        % B = A*I + 4*xk*xk'
        % We want to solve B*p = -gradfk.
        % We use the Sherman-Morrison formula to solve this system efficiently.
        %
        % Let z = -gradfk. Then, by Sherman-Morrison formula:
        % p = z/A - (4/A^2 * xk * (xk'*z))/(1 + 4*(xk'*xk)/A)
        z = -gradfk;
        denom = 1 + 4*(xk'*xk)/A;
        p = z/A - (4/A^2) * xk * (xk'*z) / denom;

        % Backtracking line search (Armijo condition)
        alpha = 1;
        bt = 0;
        while (bt < btmax)
            xnew = xk + alpha*p;
            fnew = f(xnew);
            if fnew <= fk + c1 * alpha * (gradfk' * p)
                break;
            end
            alpha = rho * alpha;
            bt = bt + 1;
```

```
        end

        % Iterate update
        xk = xnew;
        fk = fnew;
        gradfk = gradf(xk);
        gradfk_norm = norm(gradfk);

        k = k + 1;
        xseq(:, k) = xk;
        btseq(k) = bt;
    end
    xseq = xseq(:, 1:k);
    btseq = btseq(1:k);

    % Convergence rate calculation (vectorized estimation)
    x_star = xseq(:, end);
    e = vecnorm(xseq - x_star, 2, 1);
    valid = (e(1:end-2) > 1e-12) & (e(2:end-1) > 1e-12) & (e(3:end) > 1e-12);
    if any(valid)
        p_est = log(e(3:end)./e(2:end-1)) ./ log(e(2:end-1)./e(1:end-2));
        rate_convergence = mean(p_est(valid));
    else
        rate_convergence = NaN;
    end
end
```

Listing 5: Modified Newton Hessian Free

C Appendix: Truncated Newton Method

```
function [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] = truncated_newton(x0, f,
    gradf, Hessf, kmax, tolgrad, c1, rho, btmax)
% TRUNCATED_NEWTON Truncated Newton method with backtracking (Armijo) without
    preconditioning
%
%   [xk, fk, gradfk_norm, k, xseq, btseq, convergence_rate] = truncated_newton(x0, f, gradf,
     Hessf, ...
%                                                 kmax, tolgrad, c1, rho, btmax)
%
% This function implements the Truncated Newton method for unconstrained optimization
% of a function f: R^n -> R. It uses a backtracking line search based on the Armijo
    condition
% to ensure sufficient decrease in the function value at each iteration.
% The Hessian-vector product is approximated using Conjugate Gradient, making it suitable
    for
% large-scale problems where forming and storing the full Hessian is impractical.
% Preconditioning is NOT used in this version.
%
% INPUTS:
%   x0          - Starting point for the optimization (column vector)
%   f           - Function handle for the objective function f: R^n -> R
%   gradf       - Function handle for the gradient of f
%   Hessf       - Function handle for the Hessian of f (used for Hessian-vector product in
    CG)
%   kmax        - Maximum number of iterations for the outer Newton loop
%   tolgrad     - Tolerance for the gradient norm (stopping criterion)
%   c1          - Parameter for the Armijo condition (in (0,1), e.g., 1e-4)
%   rho         - Reduction factor for backtracking line search (in (0,1), e.g., 0.5)
%   btmax       - Maximum number of iterations for the backtracking line search
%
% OUTPUTS:
%   xk              - The last computed iterate (approximation of the minimizer)
%   fk              - Function value f(xk) at the final iterate
%   gradfk_norm     - Norm of the gradient at xk, ||gradf(xk)||
```

```
%   k               - Total number of iterations performed by the Newton method
%   xseq            - Sequence of iterates generated by the method (each column is an
    iterate)
%   btseq           - Sequence of backtracking iterations performed at each Newton iteration
%   rate_convergence- Estimate of the rate of convergence (average of estimated orders in
    the last iterations)


%% Input Validation
% Ensure the starting point x0 is a column vector
x0 = x0(:);
% Get the dimension of the problem from the starting point
n = numel(x0);


%% Pre-allocation for iteration history
% Initialize an array to store the sequence of iterates
xseq = zeros(n, kmax);
% Initialize an array to store the number of backtracking steps at each iteration
btseq = zeros(1, kmax);


%% Initialization
% Set the initial iterate to the starting point
xk = x0;
% Evaluate the objective function at the starting point
fk = f(xk);
% Evaluate the gradient at the starting point
gradfk = gradf(xk);
% Compute the norm of the gradient at the starting point
gradfk_norm = norm(gradfk);
% Evaluate the Hessian at the starting point (for use in CG)
hessfk = Hessf(xk);
% Initialize iteration counter
k = 0;


% Main loop of the Truncated Newton method
while (k < kmax) && (gradfk_norm >= tolgrad)
    % Dynamic tolerance for Conjugate Gradient based on current gradient norm
    etak = min(0.5, gradfk_norm);

    % Compute the search direction pk by approximately solving Hessf(xk)*pk = -gradf(xk)
    % using Conjugate Gradient method (cg_curvtrun_newt function, assumed to be defined
        elsewhere)
    pk = cg_curvtrun_newt(hessfk, -gradfk, zeros(n,1), etak);

    % Check if the computed direction is a descent direction
    % If not (i.e., not a descent direction), fall back to steepest descent direction (-
        gradf(xk))
    if gradfk' * pk >= 0
        warning('Computed direction is not a descent direction. Using steepest descent
            direction: -gradf.');
        pk = -gradf(xk);
    end

    % Backtracking line search (Armijo condition)
    % Initialize step size to 1
    alpha = 1;
    % Compute the Armijo condition target value
    curr_armijo = fk + c1 * alpha * (gradfk' * pk);
    % Calculate the new point xnew and function value fnew with current step size
    xnew = xk + alpha * pk;
    fnew = f(xnew);
    % Initialize backtracking iteration counter
    bt = 0;
    % Backtracking loop: reduce step size until Armijo condition is satisfied or max
        backtrack iterations reached
```

```
    while (bt < btmax) && (fnew > curr_armijo)
        % Reduce step size by factor rho
        alpha = rho * alpha;
        % Update Armijo condition target value with reduced step size
        curr_armijo = fk + c1 * alpha * (gradfk' * pk);
        % Recalculate xnew and fnew with updated step size
        xnew = xk + alpha * pk;
        fnew = f(xnew);
        % Increment backtracking iteration counter
        bt = bt + 1;
    end

    % Warning if maximum backtracking iterations are reached without satisfying Armijo
        condition
    if bt == btmax && fnew > curr_armijo
        warning('Maximum backtracking iterations (%d) reached at iteration %d: Armijo
            condition not satisfied.', btmax, k+1);
    end

    % Update variables for the next iteration
    % New iterate xk is set to xnew
    xk = xnew;
    % Function value at new iterate
    fk = fnew;
    % Gradient at new iterate
    gradfk = gradf(xk);
    % Gradient norm at new iterate
    gradfk_norm = norm(gradfk);
    % Hessian at new iterate (for use in next CG iteration)
    hessfk = Hessf(xk);

    % Increment iteration counter
    k = k + 1;
    % Store current iterate in xseq
    xseq(:, k) = xk;
    % Store number of backtracking steps in btseq
    btseq(k) = bt;
end


% Resize output arrays to remove pre-allocated space
xseq = xseq(:, 1:k);
btseq = btseq(1:k);


% Estimate rate of convergence (vectorized calculation for efficiency)
x_star = xseq(:, end); % Assume last iterate is close to x*
e = vecnorm(xseq - x_star, 2, 1); % Error at each iteration
% Find indices where errors are sufficiently larger than machine precision to avoid log(0)
    or division by zero
valid = (e(1:end-2) > 1e-12) & (e(2:end-1) > 1e-12) & (e(3:end) > 1e-12);
if any(valid)
    % Estimate convergence order using errors from consecutive iterations
    p_est = log(e(3:end)./e(2:end-1)) ./ log(e(2:end-1)./e(1:end-2));
    % Average the estimated orders for a single rate of convergence value
    rate_convergence = mean(p_est(valid));
else
    rate_convergence = NaN; % Not a Number if rate cannot be reliably estimated
end
end
```

Listing 6: Truncated Newton Without Preconditioning

```
function [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] =
    truncated_newton_hessian_free(x0, f, gradf, kmax, tolgrad, c1, rho, btmax)
% TRUNCATED_NEWTON_HESSIAN_FREE Hessian-Free Truncated Newton method with backtracking (
    Armijo) and no preconditioning
%
```

```
%   [xk, fk, gradfk_norm, k, xseq, btseq, convergence_rate] = truncated_newton_hessian_free(x0, f, gradf, ...
%                                                                                            kmax, tolgrad, c1, rho, btmax)
%
% This function implements the Hessian-Free Truncated Newton method for unconstrained optimization
% of a function f: R^n -> R. It approximates the Hessian-vector product using finite differences,
% making it suitable for problems where the Hessian is expensive or impossible to compute directly.
% It uses a backtracking line search based on the Armijo condition to ensure sufficient decrease
% in the function value at each iteration. Preconditioning is NOT used in this version.
%
% INPUTS:
%   x0          - Starting point for the optimization (column vector)
%   f           - Function handle for the objective function f: R^n -> R
%   gradf       - Function handle for the gradient of f
%   kmax        - Maximum number of iterations for the outer Newton loop
%   tolgrad     - Tolerance for the gradient norm (stopping criterion)
%   c1          - Parameter for the Armijo condition (in (0,1), e.g., 1e-4)
%   rho         - Reduction factor for backtracking line search (in (0,1), e.g., 0.5)
%   btmax       - Maximum number of iterations for the backtracking line search
%
% OUTPUTS:
%   xk              - The last computed iterate (approximation of the minimizer)
%   fk              - Function value f(xk) at the final iterate
%   gradfk_norm     - Norm of the gradient at xk, ||gradf(xk)||
%   k               - Total number of iterations performed by the Newton method
%   xseq            - Sequence of iterates generated by the method (each column is an iterate)
%   btseq           - Sequence of backtracking iterations performed at each Newton iteration
%   rate_convergence- Estimate of the rate of convergence (average of estimated orders in the last iterations)


%% Input Validation
% Ensure the starting point x0 is a column vector
x0 = x0(:);
% Get the dimension of the problem from the starting point
n = numel(x0);


%% Hessian-Free Approximation Parameter
sigma = sqrt(eps); % Square root of machine precision, used in finite difference approximation


%% Pre-allocation for iteration history
% Initialize an array to store the sequence of iterates
xseq = zeros(n, kmax);
% Initialize an array to store the number of backtracking steps at each iteration
btseq = zeros(1, kmax);


%% Initialization
% Set the initial iterate to the starting point
xk = x0;
% Evaluate the objective function at the starting point
fk = f(xk);
% Evaluate the gradient at the starting point
gradfk = gradf(xk);
% Compute the norm of the gradient at the starting point
gradfk_norm = norm(gradfk);
% Initialize iteration counter
k = 0;

```

```
63
64 while (k < kmax) && (gradfk_norm >= tolgrad)
65     % Dynamic tolerance for Conjugate Gradient based on current gradient norm
66     etak = min(0.5, gradfk_norm);
67
68     % Define the Hessian-vector product operator approximated using finite differences
69     % Hv(p) approximates Hessian(f(xk)) * p
70     Hv = @(p) (gradf(xk + sigma * p) - gradf(xk)) / sigma;
71
72     % Compute the search direction pk by approximately solving Hv(pk) = -gradf(xk)
73     % using Conjugate Gradient method (cg_curvtrun_hessian_free function, defined below)
74     pk = cg_curvtrun_hessian_free(Hv, -gradfk, zeros(n,1), etak);
75
76     % Descent direction check: ensure the computed direction is a descent direction
77     % If not, fall back to steepest descent direction (-gradf(xk))
78     if gradfk' * pk >= 0
79         warning('Computed direction is not a descent direction. Using steepest descent
               direction: -gradf.');
80         pk = -gradf(xk);
81     end
82
83     % Backtracking line search (Armijo condition)
84     % Initialize step size to 1
85     alpha = 1;
86     % Compute the Armijo condition target value
87     curr_armijo = fk + c1 * alpha * (gradfk' * pk);
88     % Calculate the new point xnew and function value fnew with current step size
89     xnew = xk + alpha * pk;
90     fnew = f(xnew);
91     % Initialize backtracking iteration counter
92     bt = 0;
93     % Backtracking loop: reduce step size until Armijo condition is satisfied or max
           backtrack iterations reached
94     while (bt < btmax) && (fnew > curr_armijo)
95         % Reduce step size by factor rho
96         alpha = rho * alpha;
97         % Update Armijo condition target value with reduced step size
98         curr_armijo = fk + c1 * alpha * (gradfk' * pk);
99         % Recalculate xnew and fnew with updated step size
100        xnew = xk + alpha * pk;
101        fnew = f(xnew);
102        % Increment backtracking iteration counter
103        bt = bt + 1;
104    end
105
106    % Warning if maximum backtracking iterations are reached without satisfying Armijo
           condition
107    if bt == btmax && fnew > curr_armijo
108        warning('Maximum backtracking iterations (%d) reached at iteration %d: Armijo
               condition not satisfied.', btmax, k+1);
109    end
110
111    % Update variables for the next iteration
112    % New iterate xk is set to xnew
113    xk = xnew;
114    % Function value at new iterate
115    fk = fnew;
116    % Gradient at new iterate
117    gradfk = gradf(xk);
118    % Gradient norm at new iterate
119    gradfk_norm = norm(gradfk);
120
121    % Increment iteration counter
122    k = k + 1;
123    % Store current iterate in xseq
124    xseq(:, k) = xk;
125    % Store number of backtracking steps in btseq
126    btseq(k) = bt;
```

```
end


%% Output Processing
% Resize output arrays to remove pre-allocated space
xseq = xseq(:, 1:k);
btseq = btseq(1:k);


% Estimate rate of convergence (vectorized calculation for efficiency)
x_star = xseq(:, end); % Assume last iterate is close to x*
e = vecnorm(xseq - x_star, 2, 1); % Error at each iteration
% Find indices where errors are sufficiently larger than machine precision to avoid log(0)
    or division by zero
valid = (e(1:end-2) > 1e-12) & (e(2:end-1) > 1e-12) & (e(3:end) > 1e-12);
if any(valid)
    % Estimate convergence order using errors from consecutive iterations
    p_est = log(e(3:end)./e(2:end-1)) ./ log(e(2:end-1)./e(1:end-2));
    % Average the estimated orders for a single rate of convergence value
    rate_convergence = mean(p_est(valid));
else
    rate_convergence = NaN; % Not a Number if rate cannot be reliably estimated
end
end
```

Listing 7: Truncated Newton Without Preconditioning Hessian Free

```
function xk = cg_curvtrun_newt(A, b, x0, tol)
% CG_CURVTRUN_NEWT Solves the system A*x = b using the truncated CG method
% (without preconditioning) handling potential negative curvature cases.
%
%   xk = cg_curvtrun_newt(A, b, x0, tol)
%
% INPUTS:
%   A    - Matrix (Hessian) of the system
%   b    - Right-hand side vector (typically -gradf)
%   x0   - Initial vector
%   tol  - Tolerance on the relative residual
%
% OUTPUT:
%   xk   - Approximate solution (computed direction)


% Initializations
xk = x0;
rk = b - A * xk;
pk = rk;
norm_b = norm(b);
if norm_b == 0
    norm_b = 1;  % Avoid division by zero if b is zero vector
end
relres = norm(rk) / norm_b;
kmax = 100;
k = 0;


while (relres > tol) && (k < kmax)
    Apk = A * pk;

    % If negative or near-zero curvature is detected, terminate the loop
    if pk' * Apk <= 0
        if k == 0
            % If negative curvature is detected at the first step,
            % proceed with a step calculated along pk
            alphak = (rk' * pk) / max(pk' * Apk, eps);
            xk = xk + alphak * pk;
        end
        break;
```

```
    end

    % Calculate step size using the classical CG method formula
    alphak = (rk' * rk) / (pk' * Apk);
    xk = xk + alphak * pk;

    rk_new = rk - alphak * Apk;
    beta = (rk_new' * rk_new) / (rk' * rk);

    % Update the search direction
    pk = rk_new + beta * pk;

    rk = rk_new;
    relres = norm(rk) / norm_b;
    k = k + 1;
end


end
```

Listing 8: CG Without Preconditioning

```
function xk = cg_curvtrun_hessian_free(Hv, b, x0, tol)
% CG_CURVTRUN_HESSIAN_FREE Solves the system Hv(x) = b using the Hessian-Free truncated CG
    method
% (without preconditioning) handling potential negative curvature cases.
% Hessian-Free Version: Hv is a function handle that computes the
% Hessian-vector product, NOT the explicit Hessian matrix.
%
%   xk = cg_curvtrun_hessian_free(Hv, b, x0, tol)
%
% INPUTS:
%   Hv  - Function handle for the Hessian-vector product, Hv(p) = Hessian(f(xk)) * p
%   b   - Right-hand side vector (typically -gradf)
%   x0  - Initial vector
%   tol - Tolerance on the relative residual
%
% OUTPUT:
%   xk  - Approximate solution (computed direction)


% Initializations
xk = x0;
rk = b - Hv(xk); % Use Hv to calculate the Hessian-vector product
pk = rk;
norm_b = norm(b);
if norm_b == 0
    norm_b = 1;  % Avoid division by zero if b is zero vector
end
relres = norm(rk) / norm_b;
kmax = 100;
k = 0;


while (relres > tol) && (k < kmax)
    Apk = Hv(pk); % Use Hv to calculate the Hessian-vector product

    % Negative curvature check
    if pk' * Apk <= 0
        if k == 0
            % If negative curvature is detected at the first step,
            % proceed with a step calculated along pk
            alphak = (rk' * pk) / max(pk' * Apk, eps);
            xk = xk + alphak * pk;
        end
        break;
    end

```

```
    % Calculate step size using the classical CG method formula
    alphak = (rk' * rk) / (pk' * Apk);
    xk = xk + alphak * pk;

    rk_new = rk - alphak * Apk;
    beta = (rk_new' * r_new) / (rk' * rk);

    % Update the search direction
    pk = rk_new + beta * pk;

    rk = r_new;
    relres = norm(rk) / norm_b;
    k = k + 1;
end


end
```

Listing 9: CG Without Preconditioning Hessian Free

```
function [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] = truncated_newton_pre(x0,
    f, gradf, Hessf, kmax, tolgrad, c1, rho, btmax, varargin)
% TRUNCATED_NEWTON_PRE - Truncated Newton method with backtracking and preconditioning.
% Optimized for large-scale problems and sparse Hessian matrices.
%
%   [xk, fk, gradfk_norm, k, xseq, btseq, convergence_rate] = truncated_newton_pre(x0, f,
    gradf, Hessf, kmax, tolgrad, c1, rho, btmax, varargin)
%
% This function implements a Preconditioned Truncated Newton method for unconstrained
    optimization
% of a function f: R^n -> R. It is specifically designed for large-scale problems where the
    Hessian
% matrix is sparse, and employs preconditioning to accelerate convergence.  A backtracking
    line search
% based on the Armijo condition is used to ensure sufficient decrease in the function value
    at each iteration.
% The Hessian-vector product within the Conjugate Gradient (CG) solver is used, and a
    preconditioner
% can be applied within the CG iterations (though the provided code for cg_curvtrun_newt_pre
     is not given,
% it's assumed that preconditioning is applied there).
%
% INPUTS:
%   x0          - Starting point for the optimization (column vector)
%   f           - Function handle for the objective function f: R^n -> R
%   gradf       - Function handle for the gradient of f
%   Hessf       - Function handle that, given x, returns the sparse Hessian matrix, or
%                 a constant sparse matrix if the Hessian is independent of x.
%   kmax        - Maximum number of iterations for the outer Newton loop
%   tolgrad     - Tolerance for the gradient norm (stopping criterion)
%   c1          - Parameter for the Armijo condition (typically 1e-4)
%   rho         - Step reduction factor for backtracking (0 < rho < 1, e.g., 0.5)
%   btmax       - Maximum number of step reductions in backtracking line search
%   varargin    - (Optional) Variable input arguments; may be used to pass preconditioning
    matrix
%                 or parameters to the cg_curvtrun_newt_pre function (not directly used in
    this code snippet,
%                 but included for potential future extensibility).
%
% OUTPUTS:
%   xk              - The last computed iterate (approximation of the minimizer)
%   fk              - Function value f(xk) at the final iterate
%   gradfk_norm     - Norm of the gradient at xk, ||gradf(xk)||
%   k               - Total number of iterations performed by the Newton method
%   xseq            - Sequence of iterates generated by the method (each column is an
    iterate)
%   btseq           - Sequence of backtracking iterations performed at each Newton iteration
```

```
%   rate_convergence- Estimate of the rate of convergence (average of estimated orders in
    the last iterations)


%% Input Preparation
% Ensure the starting point x0 is a column vector
x0 = x0(:);
% Get the dimension of the problem from the starting point
n = numel(x0);


%% Pre-allocation for iteration history
% Initialize an array to store the sequence of iterates
xseq = zeros(n, kmax);
% Initialize an array to store the number of backtracking steps at each iteration
btseq = zeros(1, kmax);


%% Initialization
% Set the initial iterate to the starting point
xk = x0;
% Evaluate the objective function at the starting point
fk = f(xk);
% Evaluate the gradient at the starting point
gradfk = gradf(xk);
% Compute the norm of the gradient at the starting point
gradfk_norm = norm(gradfk);


% Check if Hessf is a function handle or a matrix
if isa(Hessf, 'function_handle')
    hessfk = Hessf(xk);  % If function handle, evaluate Hessian at xk (expecting a sparse
        matrix)
else
    hessfk = Hessf;      % If not a function handle, assume Hessf is already the sparse
        Hessian matrix (constant)
end


% Initialize iteration counter
k = 0;
% Define a function handle for the Armijo condition right-hand side for efficiency
armijo_rhs = @(fk, alpha, gradfk, pk) fk + c1 * alpha * (gradfk' * pk);


% Main loop of the Preconditioned Truncated Newton method
while (k < kmax) && (gradfk_norm >= tolgrad)
    % Dynamic tolerance for Conjugate Gradient based on current gradient norm
    etak = min(0.5, gradfk_norm);

    % Solve the system Hessf(xk)*p = -gradf(xk) using Preconditioned Truncated CG
    % (cg_curvtrun_newt_pre function, assumed to be defined elsewhere and implementing
        preconditioning)
    pk = cg_curvtrun_newt_pre(hessfk, -gradfk, zeros(n,1), etak);


    % ------------------------
    % Backtracking Line Search (Armijo condition)
    % ------------------------
    % Initialize step size to 1
    alpha = 1;
    % Calculate the new point xnew with full step size
    xnew = xk + alpha * pk;
    % Evaluate the function at the new point
    fnew = f(xnew);
    % Initialize backtracking iteration counter
    bt = 0;
    % Backtracking loop: reduce step size until Armijo condition is satisfied or max
```

```
        backtrack iterations reached
    while (bt < btmax) && (fnew > armijo_rhs(fk, alpha, gradfk, pk))
        % Reduce step size by factor rho
        alpha = rho * alpha;
        % Recalculate the new point xnew with reduced step size
        xnew = xk + alpha * pk;
        % Re-evaluate the function at the new point
        fnew = f(xnew);
        % Increment backtracking iteration counter
        bt = bt + 1;
    end

    % Warning if maximum backtracking iterations are reached without satisfying Armijo
        condition
    if (bt == btmax) && (fnew > armijo_rhs(fk, alpha, gradfk, pk))
        warning('Maximum backtracking iterations (%d) reached at iteration %d: Armijo
            condition not satisfied.', btmax, k+1);
    end

    % Update variables for the next iteration
    % New iterate xk is set to xnew
    xk = xnew;
    % Function value at new iterate
    fk = fnew;
    % Gradient at new iterate
    gradfk = gradf(xk);
    % Gradient norm at new iterate
    gradfk_norm = norm(gradfk);

    % Update Hessian matrix (if Hessf is a function handle, re-evaluate at new xk)
    if isa(Hessf, 'function_handle')
        hessfk = Hessf(xk);  % Re-evaluate Hessian at xk if it's a function handle
    else
        hessfk = Hessf;      % Keep Hessian as constant matrix if it's not a function handle
    end


    % Increment iteration counter
    k = k + 1;
    % Store current iterate in xseq
    xseq(:, k) = xk;
    % Store number of backtracking steps in btseq
    btseq(k) = bt;
end


%% Output Processing
% Resize output arrays to remove pre-allocated space
xseq = xseq(:, 1:k);
btseq = btseq(1:k);


% Estimate rate of convergence (vectorized calculation for efficiency)
x_star = xseq(:, end); % Assume last iterate is close to x*
e = vecnorm(xseq - x_star, 2, 1); % Error at each iteration
% Find indices where errors are sufficiently larger than machine precision
valid = (e(1:end-2) > 1e-12) & (e(2:end-1) > 1e-12) & (e(3:end) > 1e-12);
if any(valid)
    % Estimate convergence order using errors from consecutive iterations
    p_est = log(e(3:end)./e(2:end-1)) ./ log(e(2:end-1)./e(1:end-2));
    % Average the estimated orders for a single rate of convergence value
    rate_convergence = mean(p_est(valid));
else
    rate_convergence = NaN; % Return NaN if rate cannot be reliably estimated
end


end
```

Listing 10: Truncated Newton With Preconditioning

```
function [xk, fk, gradfk_norm, k, xseq, btseq, rate_convergence] =
    truncated_newton_pre_matrixfree(x0, f, gradf, kmax, tolgrad, c1, rho, btmax)
    % This version solves the system H(x)*p = -grad using a "matrix-free" CG method.
    % Here H(x) = a*I + 4*x*x' (with 'a' defined as above) and
    % the product H(x)*v is computed without forming H explicitly.

    xk = x0(:);
    n = numel(xk);
    xseq = zeros(n, kmax);
    btseq = zeros(1, kmax);

    fk = f(xk);
    gradfk = gradf(xk);
    gradfk_norm = norm(gradfk);
    k = 0;

    armijo_rhs = @(fk, alpha, gradfk, pk) fk + c1 * alpha * (gradfk' * pk);

    while (k < kmax) && (gradfk_norm >= tolgrad)
        % Calculate 'a' and define the matrix-free Hessian-vector product
        a = 1/100000 + 2*(sum(xk.^2)-0.25);
        % We do not use a shift here, as truncated CG handles potential indefiniteness
        Afun = @(v) (a*v + 4 * xk * (xk' * v)); % Matrix-free Hessian-vector product: H(x) *
            v

        % Choose an inner tolerance parameter for CG
        etak = min(0.5, gradfk_norm);
        % Solve H*p = -gradfk using matrix-free truncated CG
        p = cg_truncated_matrix_free(Afun, -gradfk, zeros(n,1), etak);

        % Backtracking line search (Armijo condition)
        alpha = 1;
        bt = 0;
        while (bt < btmax)
            xnew = xk + alpha * p;
            fnew = f(xnew);
            if fnew <= armijo_rhs(fk, alpha, gradfk, p)
                break;
            end
            alpha = rho * alpha;
            bt = bt + 1;
        end
        if (bt == btmax) && (fnew > armijo_rhs(fk, alpha, gradfk, p))
            warning('Maximum backtracking iterations (%d) reached at iteration %d: Armijo
                condition not satisfied.', btmax, k+1);
        end

        % Iterate update
        xk = xnew;
        fk = fnew;
        gradfk = gradf(xk);
        gradfk_norm = norm(gradfk);

        k = k + 1;
        xseq(:, k) = xk;
        btseq(k) = bt;
    end
    xseq = xseq(:, 1:k);
    btseq = btseq(1:k);

    % Convergence rate calculation (vectorized estimation)
    x_star = xseq(:, end);
    e = vecnorm(xseq - x_star, 2, 1);
    valid = (e(1:end-2) > 1e-12) & (e(2:end-1) > 1e-12) & (e(3:end) > 1e-12);
```

```
    if any(valid)
        p_est = log(e(3:end)./e(2:end-1)) ./ log(e(2:end-1)./e(1:end-2));
        rate_convergence = mean(p_est(valid));
    else
        rate_convergence = NaN;
    end
end
```

Listing 11: Truncated Newton With Preconditioning Hessian Free

```
function xk = cg_curvtrun_newt_pre(A_f, b, x0, tol)
% CG_CURVTRUN_NEWT_PRE Solves A*x = b using truncated CG with sparse preconditioning.
%   A_f can be a sparse matrix or a handle that, given a point x, returns
%   the Hessian matrix (NOT used here: in truncated_newton_pre we already pass the matrix).
%
% Preconditioning Strategy:
%   - Use ichol as a preconditioner (Incomplete Cholesky) if possible.
%   - If ichol fails, fall back to a diagonal preconditioner.
%   - If negative curvature is detected (p'Ap <= 0), terminate CG iteration.


persistent L_ichol_factor  % Persistent preconditioner factor


% Preconditioner construction only on the first function call
if isempty(L_ichol_factor)
    try
        opts.michol = 'lower';
        A0 = sparse(A_f);  % In this version, A_f is already a matrix (not a handle)
        L_ichol_factor = ichol(A0, opts);
    catch
        % If ichol fails, use diagonal preconditioning
        d = diag(A_f);
        d(d==0) = 1;
        L_ichol_factor = diag(sqrt(d));
    end
end


L = L_ichol_factor; % Assign the preconditioner factor (either ichol or diagonal) to L


% Initialization
xk = x0;
% Calculate the initial residual: r = b - A*xk
r = b - A_f * xk;
% Apply preconditioner: solve L*z = r  and L'*precond_z = z, where preconditioner M = L*L'
z = L \ r;      % Solve L*z = r (forward substitution since L is lower triangular)
p = L' \ z;     % Solve L'*p = z (backward substitution since L' is upper triangular) - p is
     the preconditioned residual


normb = norm(b);
if normb == 0, normb = 1; end % Avoid division by zero if norm of b is zero


relres = norm(r) / normb; % Relative residual
n = numel(x0);
maxit = min(100, n);    % Maximum CG iterations (capped at 100 or problem dimension)
it = 0;                 % Iteration counter


while (relres > tol) && (it < maxit)
    Ap = A_f * p; % Matrix-vector product A*p

    % Negative curvature check
    if p' * Ap <= 0
```

```
        % If negative curvature is detected, especially at the first iteration, take a step along p
        if it == 0
            alpha = (r' * p) / max(p' * Ap, eps); % Step size calculation, ensure denominator is not zero
            xk = xk + alpha * p;                  % Update solution with step along p
        end
        break; % Exit CG iteration if negative curvature is detected
    end

    % Standard Conjugate Gradient updates
    alpha = (r' * p) / (p' * Ap);     % Step size
    xk = xk + alpha * p;              % Update solution

    r_new = r - alpha * Ap;           % New residual
    z_new = L \ r_new;                % Apply preconditioner to new residual: solve L*z_new = r_new
    q = L' \ z_new;                   % Solve L'*q = z_new - q is the preconditioned new residual

    beta = (r_new' * q) / (r' * p);   % Calculate beta for direction update (Fletcher-Reeves version)
    p = q + beta * p;                 % Update search direction using preconditioned residuals

    r = r_new;                        % Update residual
    relres = norm(r) / normb;         % Update relative residual norm
    it = it + 1;                      % Increment iteration counter
end


end
```

Listing 12: CG With Preconditioning

```
function xk = cg_truncated_matrix_free(Afun, b, x0, tol)
% CG_TRUNCATED_MATRIX_FREE Solves the system Afun(x) = b using the matrix-free truncated CG method
% (without preconditioning) handling potential negative curvature cases.
% In this Hessian-free version, Afun is a function handle that computes the
% matrix-vector product A*v, WITHOUT explicitly forming the matrix A.
%
%   xk = cg_truncated_matrix_free(Afun, b, x0, tol)
%
% INPUTS:
%   Afun - Function handle for the matrix-vector product, Afun(v) = A * v
%          where A is the system matrix (e.g., Hessian approximation).
%          This function should compute the product WITHOUT explicitly forming A.
%   b    - Right-hand side vector (typically -gradf)
%   x0   - Initial vector for the CG iteration
%   tol  - Tolerance on the relative residual norm (stopping criterion)
%
% OUTPUT:
%   xk   - Approximate solution vector (computed direction)

    xk = x0; % Initialize the solution vector with the starting vector
    r = b - Afun(xk); % Calculate the initial residual: r = b - Afun(xk)
    normb = norm(b); % Calculate the norm of the right-hand side vector b
    if normb == 0, normb = 1; end % Avoid division by zero if norm of b is zero
    relres = norm(r) / normb; % Calculate the initial relative residual
    p = r; % Initialize the search direction p with the initial residual
    maxit = 100;  % Maximum number of iterations for CG (modify if needed)
    it = 0; % Initialize iteration counter

    while (relres > tol) && (it < maxit)
        Ap = Afun(p); % Compute the matrix-vector product Ap = A * p using the provided function handle Afun

        % Negative curvature check: if p'*Ap <= 0, terminate CG loop
```

```
        if p' * Ap <= 0
            if it == 0
                % If negative curvature is detected at the first iteration,
                % proceed with a step calculated along p
                alpha = (r' * p) / max(p' * Ap, eps); % Step size calculation, ensure
                    denominator is not zero
                xk = xk + alpha * p; % Update solution with step along p
            end
            break; % Exit CG iteration if negative curvature is detected
        end

        alpha = (r' * r) / (p' * Ap); % Calculate step size alpha for CG update
        xk = xk + alpha * p; % Update solution vector xk
        r_new = r - alpha * Ap; % Calculate the new residual r_new = r - alpha * A*p
        beta = (r_new' * r_new) / (r' * r); % Calculate beta for updating the search
            direction (Fletcher-Reeves version)
        p = r_new + beta * p; % Update search direction p
        r = r_new; % Update residual r to r_new
        relres = norm(r) / normb; % Update relative residual norm
        it = it + 1; % Increment iteration counter
    end
end
```

Listing 13: CG With Preconditioning Matrix Free

D Appendix: Finite Differences

```
function grad = findiff_grad(F, x, h, scheme)
% FINDIFF_GRAD Approximates the gradient of a function using finite differences.
%
%   grad = findiff_grad(F, x, h, scheme)
%
% This function calculates an approximation of the gradient of a scalar function F
% at a point x using finite difference methods. It supports central and forward difference
    schemes.
%
% INPUTS:
%   F       - Function handle of the objective function, F: R^n -> R.
%   x       - Point (vector) at which to compute the gradient approximation.
%   h       - Step size for the finite difference approximation.
%           - Can be a scalar (uniform step size for all components)
%           - or a vector of length n (step size for each component).
%   scheme  - String specifying the finite difference scheme to use:
%           - 'c' for central difference (second-order accurate)
%           - 'fw' for forward difference (first-order accurate)
%
% OUTPUT:
%   grad    - Approximated gradient of F at x (column vector).

    n = length(x); % Determine the dimension of the input vector x
    grad = zeros(n,1); % Initialize the gradient vector with zeros

    % Loop through each component to compute the partial derivatives
    for i = 1:n
        x_plus = x; % Create a copy of x for the positive perturbation
        x_minus = x; % Create a copy of x for the negative perturbation

        % Determine the step size hi for the current component i
        % If h is a vector, use the i-th component h(i) as step size for the i-th partial
            derivative
        if numel(h) > 1
            hi = h(i);
        else
            hi = h; % If h is a scalar, use the same step size for all components
        end
```

```
37
38          x_plus(i) = x_plus(i) + hi;   % Perturb the i-th component of x by +hi
39          x_minus(i) = x_minus(i) - hi; % Perturb the i-th component of x by -hi
40
41          % Calculate the finite difference approximation based on the chosen scheme
42          if strcmp(scheme,'c')
43              % Central difference scheme: (F(x + hi*e_i) - F(x - hi*e_i)) / (2*hi)
44              grad(i) = (F(x_plus) - F(x_minus))/(2*hi);
45          elseif strcmp(scheme,'fw')
46              % Forward difference scheme: (F(x + hi*e_i) - F(x)) / hi
47              grad(i) = (F(x_plus) - F(x))/(hi);
48          else
49              error('Unrecognized finite difference scheme. Please use ''c'' for central or ''
                    fw'' for forward.');
50          end
51      end
52  end
```

Listing 14: Gradient with Finite Differences

```
1   function [Hessfx] = findiff_Hess(f, x, h)
2   % FINDIFF_HESS Approximates the Hessian of a function using finite differences.
3   %
4   % [Hessfx] = findiff_Hess(f, x, h)
5   %
6   % Optimized function to approximate the Hessian matrix of the function f at the point x (
        column vector)
7   % using finite difference methods. This version supports both scalar step size h and a
        vector step size h,
8   % where h(i) is used as the step size for the i-th coordinate.
9   %
10  % This implementation assumes the Hessian matrix is sparse with nonzero entries primarily
11  % on the main diagonal, the first sub- and super-diagonals, and the two corner elements
12  % at positions (1,n) and (n,1). This specific sparsity pattern is exploited for efficiency,
13  % calculating only these potentially nonzero entries.
14  %
15  % INPUTS:
16  %   f       - Function handle describing a function f: R^n -> R;
17  %   x       - n-dimensional column vector representing the point at which to approximate the
         Hessian.
18  %   h       - Step size for the finite difference approximation. This can be:
19  %             - A scalar: a uniform step size applied to all components.
20  %             - An n-dimensional vector: component-specific step sizes, where h(i) is used for
         the i-th coordinate.
21  %
22  % OUTPUT:
23  %   Hessfx  - n-by-n sparse matrix representing the approximate Hessian of f at x,
24  %             calculated using finite differences and exploiting the assumed sparsity
        pattern.
25  %
26
27
28  n = length(x); % Get the dimension of the input vector x
29  f0 = f(x); % Evaluate the function f at the point x (base function value)
30
31
32  % Preallocate arrays to store function evaluations at perturbed points.
33  f_plus  = zeros(n,1); % Array to store f(x + h_i*e_i) for each coordinate i
34  f_minus = zeros(n,1); % Array to store f(x - h_i*e_i) for each coordinate i
35
36
37  % Compute f(x + h_i*e_i) and f(x - h_i*e_i) for each coordinate direction e_i.
38  for i = 1:n
39      % Determine step size hi for the i-th coordinate
40      if numel(h) > 1
41          hi = h(i); % Use component-specific step size if h is a vector
42      else
43          hi = h;  % Use uniform step size if h is a scalar
```

```
    end
    x_temp = x; % Create a temporary copy of x to avoid modifying the original x
    x_temp(i) = x_temp(i) + hi; % Perturb the i-th component by +hi
    f_plus(i) = f(x_temp); % Evaluate f at the positively perturbed point

    x_temp = x; % Reset x_temp back to x for the negative perturbation
    x_temp(i) = x_temp(i) - hi; % Perturb the i-th component by -hi
    f_minus(i) = f(x_temp); % Evaluate f at the negatively perturbed point
end


% Preallocate the Hessian matrix as a sparse matrix to efficiently store and compute only nonzero elements.
Hessfx = spalloc(n, n, 4*n); % Allocate space for a sparse n x n matrix, expecting approximately 4*n non-zero elements (based on sparsity assumption)


% Compute diagonal entries of the Hessian using second-order central finite differences.
for i = 1:n
    % Determine step size hi for the i-th diagonal entry
    if numel(h) > 1
        hi = h(i); % Use component-specific step size if h is a vector
    else
        hi = h;  % Use uniform step size if h is a scalar
    end
    % Second-order central difference formula for diagonal elements:
    % H(i,i)     (f(x + h_i*e_i) - 2*f(x) + f(x - h_i*e_i)) / (h_i^2)
    Hessfx(i,i) = (f_plus(i) - 2*f0 + f_minus(i)) / (hi^2);
end


% Compute off-diagonal entries for adjacent variables (first sub- and super-diagonals).
for i = 1:n-1
    % Determine step sizes hi and hj for the off-diagonal entry (i, i+1)
    if numel(h) > 1
        hi = h(i);   % Step size for the i-th component
        hj = h(i+1); % Step size for the (i+1)-th component
    else
        hi = h;    % Use uniform step size if h is a scalar
        hj = h;    % Use uniform step size if h is a scalar
    end
    x_temp = x; % Create a temporary copy of x
    x_temp(i)   = x_temp(i)   + hi;   % Perturb the i-th component by +hi
    x_temp(i+1) = x_temp(i+1) + hj;   % Perturb the (i+1)-th component by +hj
    f_pair = f(x_temp); % Evaluate f at the point perturbed in both i-th and (i+1)-th directions

    % Finite difference formula for off-diagonal elements (i, i+1) and (i+1, i):
    % H(i, i+1) = H(i+1, i)     (f(x + h_i*e_i + h_j*e_j) - f(x + h_i*e_i) - f(x + h_j*e_j) + f(x)) / (h_i * h_j)
    value = (f_pair - f_plus(i) - f_plus(i+1) + f0) / (hi * hj);
    Hessfx(i, i+1) = value; % Assign the computed value to the (i, i+1) entry
    Hessfx(i+1, i) = value; % Assign the same value to the symmetric (i+1, i) entry (Hessian is symmetric)
end


% Compute the corner elements: (1,n) and (n,1), assuming wrap-around connection.
if numel(h) > 1
    h1 = h(1);  % Step size for the first component
    hn = h(n);  % Step size for the n-th (last) component
else
    h1 = h;   % Use uniform step size if h is a scalar
    hn = h;   % Use uniform step size if h is a scalar
end
x_temp = x; % Create a temporary copy of x
x_temp(1) = x_temp(1) + h1;   % Perturb the first component by +h1
x_temp(n) = x_temp(n) + hn;   % Perturb the n-th component by +hn
```

```
f_pair = f(x_temp); % Evaluate f at the point perturbed in both first and n-th directions


% Finite difference formula for corner elements (1, n) and (n, 1):
% H(1, n) = H(n, 1)     (f(x + h_1*e_1 + h_n*e_n) - f(x + h_1*e_1) - f(x + h_n*e_n) + f(x))
    / (h_1 * h_n)
value = (f_pair - f_plus(1) - f_plus(n) + f0) / (h1 * hn);
Hessfx(1, n) = value; % Assign the computed value to the (1, n) entry
Hessfx(n, 1) = value; % Assign the same value to the symmetric (n, 1) entry


end
```

Listing 15: Hessian with Finite Differences

```
function [output_str] = gridSearchFDParameters(initial_conditions, F, gradF, hessF)
%GRIDSEARCHFDPARAMETERS Executes a grid search for finite difference parameters.
%   This function performs a grid search to evaluate the performance of
%   optimization algorithms (Modified Newton and Truncated Newton) using
%   finite difference (FD) approximations for gradients and Hessians.
%   It explores different step sizes for finite differences and compares
%   central and forward difference schemes.
%
%   INPUTS:
%     initial_conditions : Cell array of initial points for optimization algorithms.
%     F                  : Function handle for the objective function F(x).
%     gradF              : Function handle for the analytical gradient of F, gradF(x).
%     hessF              : Function handle for the analytical Hessian of F, hessF(x).
%
%   OUTPUTS:
%     output_str         : String containing formatted results of the grid search,
%                          summarizing performance metrics for different FD parameters.
%     Results            : Structure (currently not returned, but could be implemented)
%                          containing detailed numerical results, including
%                          iterations, function values, and gradient norms for different
%                          parameter settings and methods.


n       = 50;       % Dimension of the problem (vector x is of size n)
kmax    = 1000;      % Maximum number of iterations for optimization algorithms
tolgrad = 1e-6;     % Tolerance for the gradient norm (stopping criterion)
c1      = 1e-4;     % Armijo condition parameter for line search
rho     = 0.5;      % Backtracking factor for line search
btmax   = 100;       % Maximum number of backtracking iterations in line search


%% 3. DEFINITION OF THE PARAMETER "GRID"
num_starting_points = length(initial_conditions); % Number of initial starting points to
    test
exponent_list = [2, 4, 6, 8, 10, 12];  % Exponents for step size h = 10^(-k), where k is
    from this list
n_params = length(exponent_list); % Number of different finite difference parameters to test


%% 4. PREALLOCATION OF RESULTS MATRICES
% Matrices to store iteration counts and final gradient norms for each configuration.
% The results are categorized by:
% - Finite Difference Type: Uniform (h is constant for all components) and Variable (h
    varies with x components)
% - Optimization Method: Modified Newton and Truncated Newton
% - Finite Difference Scheme: Central ('c') and Forward ('fw')

% --- Uniform FD (h = 10^(-k)) ---
% [MODIFIED NEWTON METHOD]
iter_uniform_mod_c   = zeros(num_starting_points, n_params); % Iterations for Modified
    Newton, Uniform FD, Central scheme
grad_uniform_mod_c   = zeros(num_starting_points, n_params); % Gradient norm for Modified
    Newton, Uniform FD, Central scheme
```

```
iter_uniform_mod_fw  = zeros(num_starting_points, n_params); % Iterations for Modified Newton, Uniform FD, Forward scheme
grad_uniform_mod_fw  = zeros(num_starting_points, n_params); % Gradient norm for Modified Newton, Uniform FD, Forward scheme


% [TRUNCATED NEWTON METHOD]
iter_uniform_trunc_c  = zeros(num_starting_points, n_params); % Iterations for Truncated Newton, Uniform FD, Central scheme
grad_uniform_trunc_c  = zeros(num_starting_points, n_params); % Gradient norm for Truncated Newton, Uniform FD, Central scheme
iter_uniform_trunc_fw = zeros(num_starting_points, n_params); % Iterations for Truncated Newton, Uniform FD, Forward scheme
grad_uniform_trunc_fw = zeros(num_starting_points, n_params); % Gradient norm for Truncated Newton, Uniform FD, Forward scheme


% --- FD with Variable h (h is proportional to |x|) ---
% [MODIFIED NEWTON METHOD]
iter_var_mod_c   = zeros(num_starting_points, n_params); % Iterations for Modified Newton, Variable FD, Central scheme
grad_var_mod_c   = zeros(num_starting_points, n_params); % Gradient norm for Modified Newton, Variable FD, Central scheme
iter_var_mod_fw  = zeros(num_starting_points, n_params); % Iterations for Modified Newton, Variable FD, Forward scheme
grad_var_mod_fw  = zeros(num_starting_points, n_params); % Gradient norm for Modified Newton, Variable FD, Forward scheme


% [TRUNCATED NEWTON METHOD]
iter_var_trunc_c  = zeros(num_starting_points, n_params); % Iterations for Truncated Newton, Variable FD, Central scheme
grad_var_trunc_c  = zeros(num_starting_points, n_params); % Gradient norm for Truncated Newton, Variable FD, Central scheme
iter_var_trunc_fw = zeros(num_starting_points, n_params); % Iterations for Truncated Newton, Variable FD, Forward scheme
grad_var_trunc_fw = zeros(num_starting_points, n_params); % Gradient norm for Truncated Newton, Variable FD, Forward scheme


%% 5. GRID SEARCH: LOOP OVER STARTING POINTS AND PARAMETERS
% This section performs the core grid search. It iterates through:
% - Initial starting points (defined in 'initial_conditions')
% - Exponents for step size 'h' (defined in 'exponent_list')
% For each combination, it tests four scenarios:
% 1. Modified Newton with Uniform FD (Central Difference)
% 2. Modified Newton with Uniform FD (Forward Difference)
% 3. Truncated Newton with Uniform FD (Central Difference)
% 4. Truncated Newton with Uniform FD (Forward Difference)
% 5. Modified Newton with Variable FD (Central Difference)
% 6. Modified Newton with Variable FD (Forward Difference)
% 7. Truncated Newton with Variable FD (Central Difference)
% 8. Truncated Newton with Variable FD (Forward Difference)


for i = 1:num_starting_points % Loop through each initial starting point
    x0 = initial_conditions{i}; % Get the current initial starting point
    for j = 1:n_params % Loop through each finite difference parameter (exponent k)
        kexp = exponent_list(j); % Get the current exponent k for h = 10^(-k)


        %% 5.1 Uniform FD: h = 10^(-kexp)
        h_uniform = 10^(-kexp); % Calculate the uniform step size h
        % Define function handles for finite difference gradient approximations using central and forward schemes
        grad_fdiff_c = @(x) findiff_grad(F, x, h_uniform, 'c'); % Central difference gradient approximation
        grad_fdiff_fw = @(x) findiff_grad(F, x, h_uniform, 'fw'); % Forward difference
```

```
        gradient approximation
Hess_fdiff   = @(x) findiff_Hess(F, x, h_uniform); % Hessian approximation (using
    the same uniform h)


% --- Modified Newton Method with Uniform FD ---
% Modified Newton, Uniform FD, Central Difference Scheme
try
    [~, ~, grad_norm_mod, iter_mod, ~, ~, ~] = ...
        modified_newton(x0, F, grad_fdiff_c, Hess_fdiff, kmax, tolgrad, c1, rho,
            btmax); % Run Modified Newton
    iter_uniform_mod_c(i,j) = iter_mod; % Store iterations
    grad_uniform_mod_c(i,j) = grad_norm_mod; % Store final gradient norm
catch ME
    disp(['Error in modified_newton (uniform, central) for starting point ' num2str(
        i) ', k = ' num2str(kexp) ': ' ME.message]); % Display error message if
        method fails
    iter_uniform_mod_c(i,j) = NaN; % Store NaN for iterations in case of error
    grad_uniform_mod_c(i,j) = NaN; % Store NaN for gradient norm in case of error
end


% Modified Newton, Uniform FD, Forward Difference Scheme
try
    [~, ~, grad_norm_mod, iter_mod, ~, ~, ~] = ...
        modified_newton(x0, F, grad_fdiff_fw, Hess_fdiff, kmax, tolgrad, c1, rho,
            btmax); % Run Modified Newton
    iter_uniform_mod_fw(i,j) = iter_mod; % Store iterations
    grad_uniform_mod_fw(i,j) = grad_norm_mod; % Store final gradient norm
catch ME
    disp(['Error in modified_newton (uniform, forward) for starting point ' num2str(
        i) ', k = ' num2str(kexp) ': ' ME.message]); % Display error message if
        method fails
    iter_uniform_mod_fw(i,j) = NaN; % Store NaN for iterations in case of error
    grad_uniform_mod_fw(i,j) = NaN; % Store NaN for gradient norm in case of error
end


% --- Truncated Newton Method with Uniform FD ---
% Truncated Newton, Uniform FD, Central Difference Scheme
try
    [~, ~, grad_norm_trunc, iter_trunc, ~, ~, ~] = ...
        truncated_newton_pre(x0, F, grad_fdiff_c, Hess_fdiff, kmax, tolgrad, c1, rho
            , btmax); % Run Truncated Newton
    iter_uniform_trunc_c(i,j) = iter_trunc; % Store iterations
    grad_uniform_trunc_c(i,j) = grad_norm_trunc; % Store final gradient norm
catch ME
    disp(['Error in truncated_newton_pre (uniform, central) for starting point '
        num2str(i) ', k = ' num2str(kexp) ': ' ME.message]); % Display error message
         if method fails
    iter_uniform_trunc_c(i,j) = NaN; % Store NaN for iterations in case of error
    grad_uniform_trunc_c(i,j) = NaN; % Store NaN for gradient norm in case of error
end


% Truncated Newton, Uniform FD, Forward Difference Scheme
try
    [~, ~, grad_norm_trunc, iter_trunc, ~, ~, ~] = ...
        truncated_newton_pre(x0, F, grad_fdiff_fw, Hess_fdiff, kmax, tolgrad, c1,
            rho, btmax); % Run Truncated Newton
    iter_uniform_trunc_fw(i,j) = iter_trunc; % Store iterations
    grad_uniform_trunc_fw(i,j) = grad_norm_trunc; % Store final gradient norm
catch ME
    disp(['Error in truncated_newton_pre (uniform, forward) for starting point '
        num2str(i) ', k = ' num2str(kexp) ': ' ME.message]); % Display error message
         if method fails
    iter_uniform_trunc_fw(i,j) = NaN; % Store NaN for iterations in case of error
    grad_uniform_trunc_fw(i,j) = NaN; % Store NaN for gradient norm in case of error
```

```
155         end
156
157
158         %% 5.2 Variable FD: h_var = h_uniform * abs(x0)
159         h_var = h_uniform * abs(x0); % Calculate variable step size h based on initial point
                x0
160         grad_fdiff_var_c = @(x) findiff_grad(F, x, h_var, 'c'); % Central difference
                gradient with variable h
161         grad_fdiff_var_fw = @(x) findiff_grad(F, x, h_var, 'fw'); % Forward difference
                gradient with variable h
162         Hess_fdiff_var   = @(x) findiff_Hess(F, x, h_var); % Hessian approximation with
                variable h
163
164
165         % --- Modified Newton Method with Variable FD ---
166         % Modified Newton, Variable FD, Central Difference Scheme
167         try
168             [~, ~, grad_norm_mod, iter_mod, ~, ~, ~] = ...
169                 modified_newton(x0, F, grad_fdiff_var_c, Hess_fdiff_var, kmax, tolgrad, c1,
                        rho, btmax); % Run Modified Newton
170             iter_var_mod_c(i,j) = iter_mod; % Store iterations
171             grad_var_mod_c(i,j) = grad_norm_mod; % Store final gradient norm
172         catch ME
173             disp(['Error in modified_newton (variable, central) for starting point ' num2str
                    (i) ', kesp = ' num2str(kexp) ': ' ME.message]); % Display error message if
                    method fails
174             iter_var_mod_c(i,j) = NaN; % Store NaN for iterations in case of error
175             grad_var_mod_c(i,j) = NaN; % Store NaN for gradient norm in case of error
176         end
177
178
179         % Modified Newton, Variable FD, Forward Difference Scheme
180         try
181             [~, ~, grad_norm_mod, iter_mod, ~, ~, ~] = ...
182                 modified_newton(x0, F, grad_fdiff_var_fw, Hess_fdiff_var, kmax, tolgrad, c1,
                        rho, btmax); % Run Modified Newton
183             iter_var_mod_fw(i,j) = iter_mod; % Store iterations
184             grad_var_mod_fw(i,j) = grad_norm_mod; % Store final gradient norm
185         catch ME
186             disp(['Error in modified_newton (variable, forward) for starting point ' num2str
                    (i) ', kesp = ' num2str(kexp) ': ' ME.message]); % Display error message if
                    method fails
187             iter_var_mod_fw(i,j) = NaN; % Store NaN for iterations in case of error
188             grad_var_mod_fw(i,j) = NaN; % Store NaN for gradient norm in case of error
189         end
190
191
192         % --- Truncated Newton Method with Variable FD ---
193         % Truncated Newton, Variable FD, Central Difference Scheme
194         try
195             [~, ~, grad_norm_trunc, iter_trunc, ~, ~, ~] = ...
196                 truncated_newton_pre(x0, F, grad_fdiff_var_c, Hess_fdiff_var, kmax, tolgrad,
                        c1, rho, btmax); % Run Truncated Newton
197             iter_var_trunc_c(i,j) = iter_trunc; % Store iterations
198             grad_var_trunc_c(i,j) = grad_norm_trunc; % Store final gradient norm
199         catch ME
200             disp(['Error in truncated_newton_pre (variable, central) for starting point '
                    num2str(i) ', kesp = ' num2str(kexp) ': ' ME.message]); % Display error
                    message if method fails
201             iter_var_trunc_c(i,j) = NaN; % Store NaN for iterations in case of error
202             grad_var_trunc_c(i,j) = NaN; % Store NaN for gradient norm in case of error
203         end
204
205
206         % Truncated Newton, Variable FD, Forward Difference Scheme
207         try
208             [~, ~, grad_norm_trunc, iter_trunc, ~, ~, ~] = ...
209                 truncated_newton_pre(x0, F, grad_fdiff_var_fw, Hess_fdiff_var, kmax, tolgrad
```

```
                    , c1, rho, btmax); % Run Truncated Newton
            iter_var_trunc_fw(i,j) = iter_trunc; % Store iterations
            grad_var_trunc_fw(i,j) = grad_norm_trunc; % Store final gradient norm
        catch ME
            disp(['Error in truncated_newton_pre (variable, forward) for starting point '
                num2str(i) ', kesp = ' num2str(kexp) ': ' ME.message]); % Display error
                message if method fails
            iter_var_trunc_fw(i,j) = NaN; % Store NaN for iterations in case of error
            grad_var_trunc_fw(i,j) = NaN; % Store NaN for gradient norm in case of error
        end
    end
end




%% 6. CALCULATION OF AVERAGES (OVER STARTING POINTS)
% Average the iteration counts and gradient norms over all starting points for each
    parameter setting.
% Uniform FD Averages
avg_iter_uniform_mod_c  = mean(iter_uniform_mod_c, 1); % Average iterations for Modified
    Newton, Uniform FD, Central
avg_grad_uniform_mod_c  = mean(grad_uniform_mod_c, 1); % Average gradient norm for Modified
    Newton, Uniform FD, Central
avg_iter_uniform_mod_fw = mean(iter_uniform_mod_fw, 1); % Average iterations for Modified
    Newton, Uniform FD, Forward
avg_grad_uniform_mod_fw = mean(grad_uniform_mod_fw, 1); % Average gradient norm for Modified
     Newton, Uniform FD, Forward


avg_iter_uniform_trunc_c  = mean(iter_uniform_trunc_c, 1); % Average iterations for
    Truncated Newton, Uniform FD, Central
avg_grad_uniform_trunc_c  = mean(grad_uniform_trunc_c, 1); % Average gradient norm for
    Truncated Newton, Uniform FD, Central
avg_iter_uniform_trunc_fw = mean(iter_uniform_trunc_fw, 1); % Average iterations for
    Truncated Newton, Uniform FD, Forward
avg_grad_uniform_trunc_fw = mean(grad_uniform_trunc_fw, 1); % Average gradient norm for
    Truncated Newton, Uniform FD, Forward


% Variable FD Averages
avg_iter_var_mod_c  = mean(iter_var_mod_c, 1); % Average iterations for Modified Newton,
    Variable FD, Central
avg_grad_var_mod_c  = mean(grad_var_mod_c, 1); % Average gradient norm for Modified Newton,
    Variable FD, Central
avg_iter_var_mod_fw = mean(iter_var_mod_fw, 1); % Average iterations for Modified Newton,
    Variable FD, Forward
avg_grad_var_mod_fw = mean(grad_var_mod_fw, 1); % Average gradient norm for Modified Newton,
     Variable FD, Forward


avg_iter_var_trunc_c  = mean(iter_var_trunc_c, 1); % Average iterations for Truncated Newton
    , Variable FD, Central
avg_grad_var_trunc_c  = mean(grad_var_trunc_c, 1); % Average gradient norm for Truncated
    Newton, Variable FD, Central
avg_iter_var_trunc_fw = mean(iter_var_trunc_fw, 1); % Average iterations for Truncated
    Newton, Variable FD, Forward
avg_grad_var_trunc_fw = mean(grad_var_trunc_fw, 1); % Average gradient norm for Truncated
    Newton, Variable FD, Forward


%% 7. SELECTION OF THE BEST PARAMETERS (for Modified Newton, Uniform FD)
% For demonstration, select the "best" parameter based on Modified Newton with Uniform FD.
% "Best" is defined as the parameter that achieves the minimum gradient norm,
% and among those, the minimum number of iterations.

% Central Difference - find best k based on minimum gradient norm, then minimum iterations
    among candidates
```

```
[min_grad_uniform_mod_c, best_index_uniform_mod_c] = min(avg_grad_uniform_mod_c); % Find minimum average gradient norm and its index
candidates = find(abs(avg_grad_uniform_mod_c - min_grad_uniform_mod_c) < 1e-12); % Find indices with gradient norm close to minimum
if length(candidates) > 1 % If multiple candidates with similar gradient norms, choose based on minimum iterations
    [~, idx] = min(avg_iter_uniform_mod_c(candidates)); % Find index of minimum iterations among candidates
    best_index_uniform_mod_c = candidates(idx); % Update best index to the one with minimum iterations
end
best_k_uniform_mod_c = exponent_list(best_index_uniform_mod_c); % Get the best k exponent for Uniform FD, Central scheme

% Forward Difference - find best k based on minimum gradient norm, then minimum iterations among candidates
[min_grad_uniform_mod_fw, best_index_uniform_mod_fw] = min(avg_grad_uniform_mod_fw); % Find minimum average gradient norm and its index
candidates = find(abs(avg_grad_uniform_mod_fw - min_grad_uniform_mod_fw) < 1e-12); % Find indices with gradient norm close to minimum
if length(candidates) > 1 % If multiple candidates with similar gradient norms, choose based on minimum iterations
    [~, idx] = min(avg_iter_uniform_mod_fw(candidates)); % Find index of minimum iterations among candidates
    best_index_uniform_mod_fw = candidates(idx); % Update best index to the one with minimum iterations
end
best_k_uniform_mod_fw = exponent_list(best_index_uniform_mod_fw); % Get the best k exponent for Uniform FD, Forward scheme


%% 8. CONSTRUCTION OF THE OUTPUT STRING (output_str)
% Format the results into a readable string for display.
output_str = ""; % Initialize output string
output_str = output_str + newline + "%% VISUALIZATION OF RESULTS" + newline; % Add section header for visualization
output_str = output_str + "*** Grid Search Results ***" + newline + newline; % Add title for grid search results


% --- Results for Uniform FD ---
output_str = output_str + "--- Uniform Finite Differences ---" + newline; % Sub-section header for Uniform FD
output_str = output_str + "Modified Newton Method:" + newline; % Method sub-header
output_str = output_str + "Central Difference Scheme:" + newline; % Scheme sub-header
output_str = output_str + "Tested Parameters (k): " + string(mat2str(exponent_list)) + newline; % List of tested k parameters
output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_uniform_mod_c, 4)) + newline; % Average iterations array as string
output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_uniform_mod_c, 4)) + newline; % Average gradient norm array as string
output_str = output_str + "=> Best k = " + string(num2str(best_k_uniform_mod_c)) + " (Average Iterations = " + ...
    string(num2str(avg_iter_uniform_mod_c(best_index_uniform_mod_c), '%.2f')) + ", Average Gradient Norm = " + ...
    string(num2str(avg_grad_uniform_mod_c(best_index_uniform_mod_c), '%.3e')) + ")" + newline + newline; % Best k and its performance metrics


output_str = output_str + "Forward Difference Scheme:" + newline; % Scheme sub-header
output_str = output_str + "Tested Parameters (k): " + string(mat2str(exponent_list)) + newline; % List of tested k parameters
output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_uniform_mod_fw, 4)) + newline; % Average iterations array as string
output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_uniform_mod_fw, 4)) + newline; % Average gradient norm array as string
output_str = output_str + "=> Best k = " + string(num2str(best_k_uniform_mod_fw)) + " (Average Iterations = " + ...
```

```
    string(num2str(avg_iter_uniform_mod_fw(best_index_uniform_mod_fw), '%.2f')) + ", Average Gradient Norm = " + ...
    string(num2str(avg_grad_uniform_mod_fw(best_index_uniform_mod_fw), '%.3e')) + ")" + newline + newline; % Best k and its performance metrics


output_str = output_str + "Truncated Newton Method:" + newline; % Method sub-header
output_str = output_str + "Central Difference Scheme:" + newline; % Scheme sub-header
output_str = output_str + "Tested Parameters (k): " + string(mat2str(exponent_list)) + newline; % List of tested k parameters
output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_uniform_trunc_c, 4)) + newline; % Average iterations array as string
output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_uniform_trunc_c, 4)) + newline; % Average gradient norm array as string
output_str = output_str + "=> Best k = " + string(num2str(best_k_uniform_mod_c)) + " (Average Iterations = " + ...
    string(num2str(avg_iter_uniform_trunc_c(best_index_uniform_mod_c), '%.2f')) + ", Average Gradient Norm = " + ...
    string(num2str(avg_grad_uniform_trunc_c(best_index_uniform_mod_c), '%.3e')) + ")" + newline + newline; % Best k and its performance metrics


output_str = output_str + "Forward Difference Scheme:" + newline; % Scheme sub-header
output_str = output_str + "Tested Parameters (k): " + string(mat2str(exponent_list)) + newline; % List of tested k parameters
output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_uniform_trunc_fw, 4)) + newline; % Average iterations array as string
output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_uniform_mod_fw, 4)) + newline; % Average gradient norm array as string
output_str = output_str + "=> Best k = " + string(num2str(best_k_uniform_mod_fw)) + " (Average Iterations = " + ...
    string(num2str(avg_iter_uniform_trunc_fw(best_index_uniform_mod_fw), '%.2f')) + ", Average Gradient Norm = " + ...
    string(num2str(avg_grad_uniform_trunc_fw(best_index_uniform_mod_fw), '%.3e')) + ")" + newline + newline; % Best k and its performance metrics


% --- Results for Variable FD ---
output_str = output_str + newline + "--- Finite Differences with Variable h ---" + newline; % Sub-section header for Variable FD
output_str = output_str + "Modified Newton Method:" + newline; % Method sub-header
output_str = output_str + "Central Difference Scheme:" + newline; % Scheme sub-header
output_str = output_str + "Tested Parameters (kesp): " + string(mat2str(exponent_list)) + newline; % List of tested k parameters
output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_var_mod_c, 4)) + newline; % Average iterations array as string
output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_var_mod_c, 4)) + newline; % Average gradient norm array as string
output_str = output_str + "=> Best kesp = " + string(num2str(exponent_list(best_index_uniform_mod_c))) + " (Average Iterations = " + ...
    string(num2str(avg_iter_var_mod_c(best_index_uniform_mod_c), '%.2f')) + ", Average Gradient Norm = " + ...
    string(num2str(avg_grad_var_mod_c(best_index_uniform_mod_c), '%.3e')) + ")" + newline + newline; % Best k and its performance metrics


output_str = output_str + "Forward Difference Scheme:" + newline; % Scheme sub-header
output_str = output_str + "Tested Parameters (kesp): " + string(mat2str(exponent_list)) + newline; % List of tested k parameters
output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_var_mod_fw, 4)) + newline; % Average iterations array as string
output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_var_mod_fw, 4)) + newline; % Average gradient norm array as string
output_str = output_str + "=> Best kesp = " + string(num2str(exponent_list(best_index_uniform_mod_fw))) + " (Average Iterations = " + ...
    string(num2str(avg_iter_var_mod_fw(best_index_uniform_mod_fw), '%.2f')) + ", Average Gradient Norm = " + ...
    string(num2str(avg_grad_var_mod_fw(best_index_uniform_mod_fw), '%.3e')) + ")" + newline
```

```
        + newline; % Best k and its performance metrics
341
342
343 output_str = output_str + "Truncated Newton Method:" + newline; % Method sub-header
344 output_str = output_str + "Central Difference Scheme:" + newline; % Scheme sub-header
345 output_str = output_str + "Tested Parameters (kesp): " + string(mat2str(exponent_list)) +
        newline; % List of tested k parameters
346 output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_var_trunc_c, 4))
        + newline; % Average iterations array as string
347 output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_var_trunc_c,
        4)) + newline; % Average gradient norm array as string
348 output_str = output_str + "=> Best kesp = " + string(num2str(exponent_list(
        best_index_uniform_mod_c))) + " (Average Iterations = " + ...
349     string(num2str(avg_iter_var_trunc_c(best_index_uniform_mod_c), '%.2f')) + ", Average
            Gradient Norm = " + ...
350     string(num2str(avg_grad_var_trunc_c(best_index_uniform_mod_c), '%.3e')) + ")" + newline
            + newline; % Best k and its performance metrics
351
352
353 output_str = output_str + "Forward Difference Scheme:" + newline; % Scheme sub-header
354 output_str = output_str + "Tested Parameters (kesp): " + string(mat2str(exponent_list)) +
        newline; % List of tested k parameters
355 output_str = output_str + "Average Iterations: " + string(mat2str(avg_iter_var_trunc_fw, 4))
         + newline; % Average iterations array as string
356 output_str = output_str + "Average Gradient Norm: " + string(mat2str(avg_grad_var_fw, 4)) +
        newline; % Average gradient norm array as string
357 output_str = output_str + "=> Best kesp = " + string(num2str(exponent_list(
        best_index_uniform_mod_fw))) + " (Average Iterations = " + ...
358     string(num2str(avg_iter_var_trunc_fw(best_index_uniform_mod_fw), '%.2f')) + ", Average
            Gradient Norm = " + ...
359     string(num2str(avg_grad_var_trunc_fw(best_index_uniform_mod_fw), '%.3e')) + ")" +
            newline + newline; % Best k and its performance metrics
360
361
362 output_str = output_str + "*** End of Grid Search Analysis ***" + newline; % Footer for
        output string
363 disp(output_str); % Display the formatted output string in the command window
364
365 end
```

Listing 16: Grid search - Finite Differences

References

[1] Test Problems for Unconstrained Optimization. Available at: https://www.researchgate.net/publication/325314497_Test_Problems_for_Unconstrained_Optimization

[2] Pierraccini, Sandra. *Numerical Optimization for Large-Scale Problems*. Lecture notes, Politecnico di Torino, A.Y. 2024/2025.

[3] Della Santa, Francesco. *Numerical Optimization for Large Scale Problems*. Lecture Notes, Politecnico di Torino, A.Y. 2024/2025.