IOE511 Project Report

(Team JEKY)

Kai Deng Jinxiang Ma Yubo Shao Elizabeth Smith

1 Introduction

In optimization theory and applications, the efficiency and scalability of algorithms are crucial for tackling large-scale problems. The quasi-Newton methods, including the Broyden-Fletcher-Goldfarb-Shanno (BFGS) and its memory-efficient variant, the Limited-memory BFGS (L-BFGS), are significant advancements in this field. While BFGS circumvents the direct calculation of the Hessian matrix and thus reduces computational intensity, it demands substantial memory resources, which can be computationally heavy for high-dimensional problems. L-BFGS addresses this constraint by storing a limited number of vector approximations of the Hessian, thereby diminishing memory requirements. However, this approximation introduces trade-offs in precision and the number of iterations needed for convergence.

Within this report, we present a structured comparison of optimization algorithms. The first-order methods implemented are Gradient Descent with Backtracking Line Search (GradientDescent) and Gradient Descent with Wolfe Line Search (GradientDescentW). In terms of second-order methods, our implementations include Modified Newton with Backtracking Line Search (Newton) and Modified Newton with Wolfe Line Search (NewtonW), alongside the Trust Region Newton with CG Subproblem Solver (TRNewtonCG). Additionally, we address quasi-Newton methods, with our implementations being SR1 Quasi-Newton with CG Subproblem Solver (TRSR1CG), BFGS Quasi-Newton with Backtracking Line Search (BFGS), BFGS Quasi-Newton with Wolfe Line Search (BFGSW), DFP Quasi-Newton with Backtracking Line Search (DFP), and DFP Quasi-Newton with Wolfe Line Search (DFPW).

Furthermore, We will explore the impact of varying memory sizes on the efficiency of the L-BFGS method. Through comprehensive numerical experiments, we evaluate how memory size affects both computational time and the number of iterations until convergence. Our objective is to establish whether there is a consistent relationship between memory size and the performance of the L-BFGS algorithm and to determine the optimal memory setting that balances speed and iteration count. By comparing the optimal memory size for L-BFGS with the traditional full-memory BFGS approach, we aim to offer insights into the relative performances of these methods.

2 Algorithm

- **Gradient Descent with Backtracking Line Search** This algorithm iteratively finds the minimum of a function by updating steps proportional to the negative gradient. The backtracking line search adjusts the step size to ensure a consistent decrease in the function value.

- **Gradient Descent with Wolfe Line Search** This algorithm iteratively finds the minimum of a function by updating steps proportional to the negative gradient. Similar to the backtracking variant, this method adjusts step sizes based on the Wolfe conditions, which balance sufficient function decrease and curvature.

- **Modified Newton with Backtracking Line Search** The modified Newton's method utilized second-order derivatives, which incorporates the Hessian matrix to accelerate convergence near optima. The backtracking line search adjusts the step size to ensure a consistent decrease in the function value.

- **Modified Newton with Wolfe Line Search** The modified Newton's method utilized second-order derivatives, which incorporates the Hessian Matrix to accelerate convergence near optima. This method ensures step sizes that provide adequate descent and curvature compliance using the Wolfe conditions.

- **Trust region Newton with CG subproblem solver**: A algorithm where a region around the current point is trusted to approximate the function accurately. Within this region, a quadratic model is minimized using the Conjugate Gradient (CG) method.

- **SR1 quasi-Newton with CG subproblem solver** A algorithm that uses the Symmetric Rank 1 (SR1) update to approximate the Hessian in the quasi-Newton method. The CG method handles nonconvexity by adjusting the trust region to find a direction that improves the objective function.

- **BFGS quasi-Newton with backtracking line search** A algorithm that uses the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update to approximate the Hessian matrix. The backtracking line search helps find the appropriate step size to ensure a consistent decrease in the function value.

- **BFGS quasi-Newton with Wolfe line search** A algorithm that uses the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update to approximate the Hessian matrix. Similar to the backtracking variant of BFGS Method, this method adjusts step sizes based on the Wolfe conditions, which balance sufficient function decrease and curvature.

- **DFP quasi-Newton with backtracking line search** A algorithm that uses the Davidon-Fletcher-Powell (DFP) update formula for the Hessian matrix approximation. The backtracking line search helps find the appropriate step size to ensure a consistent decrease in the function value.

- **DFP quasi-Newton with Wolfe line search** A algorithm that uses the Davidon-Fletcher-Powell (DFP) update formula for the Hessian matrix approximation. Similar to the backtracking variant of DFP

Method, this method adjusts step sizes based on the Wolfe conditions, which balance sufficient function decrease and curvature.

The termination conditions used for all algorithms are $\|\nabla f(x_k)\|_{\text{inf}} \leq \epsilon \max\{\|\nabla f(x_0)\|_{\text{inf}}, 1\}$ and $k < \text{max\_iters}$, with $\epsilon = 10^{-6}$ and $\text{max\_iters} = 10^3$. Default values for other necessary parameters of different algorithms can be found in Table 9. All codes can be found in our GitHub repository.

3 Comparison among Algorithms

3.1 Summary of Results

We compared the performance of 10 algorithms across 12 problems in terms of the number of iterations, execution time (CPU seconds), number of function evaluations, and number of gradient evaluations. Additionally, we present the final objective values of each algorithm for different problems (see Appendix A).

In Table 4, it's evident that gradient descent (GD) algorithms fall short compared to others. Both GD algorithms fail within the allowed 1,000 iterations. Some problems, like `Rosenbrock_2`, terminate near the optimal solution, while others, such as `P2`, `P3`, and `P4`, terminate far from it. This discrepancy likely stems from the step size; linear convergence drives each iteration closer to the solution, but the problem's condition number can significantly prolong this process.

Trust region, BFGS, and DFP algorithms performed as anticipated, generally outperforming gradient descent but lagging behind the Newton methods across most comparison metrics. Newton's method notably excelled, demonstrating significantly faster convergence in terms of iterations, although it was significantly slower in terms of execution time. Despite being slower in some cases compared to BFGS and DFP, Newton's method shines due to its ability to reach a solution with remarkably fewer iterations, leading to a reduced number of function and gradient evaluations.

If we were to pick a preferred algorithm, Newton's method with a Wolfe Line Search stands out. Despite its demand for full second-order information, the added computational cost proves worthwhile for these specific problems. However, we acknowledge that with larger datasets, such as those encountered in platforms like ChatGPT and other AI applications, utilizing full second-order information may not be feasible or optimal. Nonetheless, based on our classroom learning and the problem set for this project, the enhanced speed and precision achieved through the combination of Newton's method and the Wolfe Line Search justify the additional computational overhead.

3.2 Performance Profiles

Across all performance metrics examined, Newton's method with Wolfe Line Search emerged as the standout among all algorithms (see Figure 5). When assessing the number of iterations required, Wolfe Line Search achieved success in 58.33% of the problems, outclassing its counterparts. Remarkably, with a slight adjustment

in τ, it swiftly conquered 100% of the problems compared to other algorithms. Following closely were the Trust Region Newton's method and BFGS with Wolfe Line Search, each effectively addressing 16.67% of the problems in terms of iterations. However, Gradient Descent algorithms, despite tying with TR and BFGSW, notably lacked reliability, evidenced by their failure to complete several problems.

In terms of execution time, Newton's methods with Wolfe Line Search emerged as the unequivocal winner, resolving 33% of problems with $\tau = 1$. BFGS and Newton's method followed suit, addressing 25% and 16% of problems respectively, also with τ equal to 1.

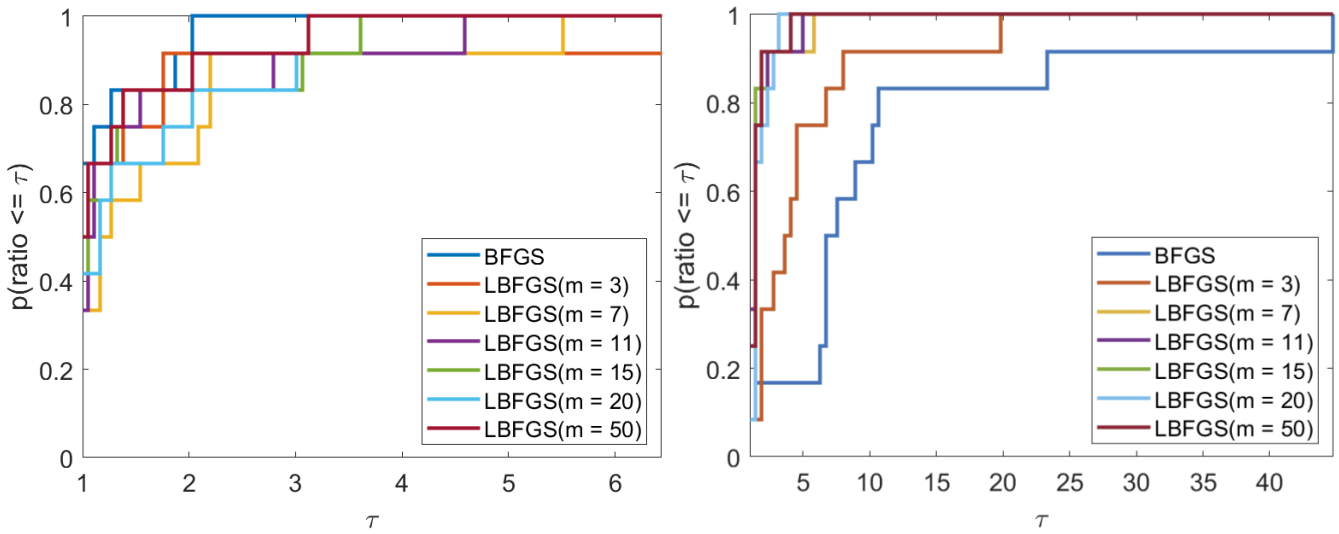
Additionally, when evaluating function and gradient evaluations, Newton's method emerged as the frontrunner, successfully tackling 75% of the problems when $\tau = 1$. Notably, Newton with Wolfe Line Search performed relatively worse under these metrics due to the higher number of gradient and function evaluations compared to other algorithms.

Regarding the τ ratio, while all algorithms, except Gradient Descent, eventually achieved a 100% problem-solving rate with an unbounded tau, setting τ to a maximum of 100 hindered the reliability of many algorithms. When iterations served as the comparison metric, only Newton Methods could solve 100% of the problems under these conditions, with Trust Region with NewtonCG closely following, resolving 92% of the problems. BFGS/BFGSW shared the third spot, successfully addressing 83% of the problems. Similarly, in terms of execution time, Newton methods achieved a perfect 100% problem-solving rate for τ constrained to 100, while BGFS, BFGSW, DFP, and DFPW each solved 83% of problems.

It's worth noting that for many of these comparison metrics, the Trust Region with rank-1 CG method required the largest τ (excluding Gradient Descent), resulting in the lowest reliability score when τ was constrained to 100.

4 Investigating the Impact of Memory on the Performance of L-BFGS method

4.1 Performance Profiles



(a) Performance Profiles: Iteration

(b) Performance Profiles: Execution Time

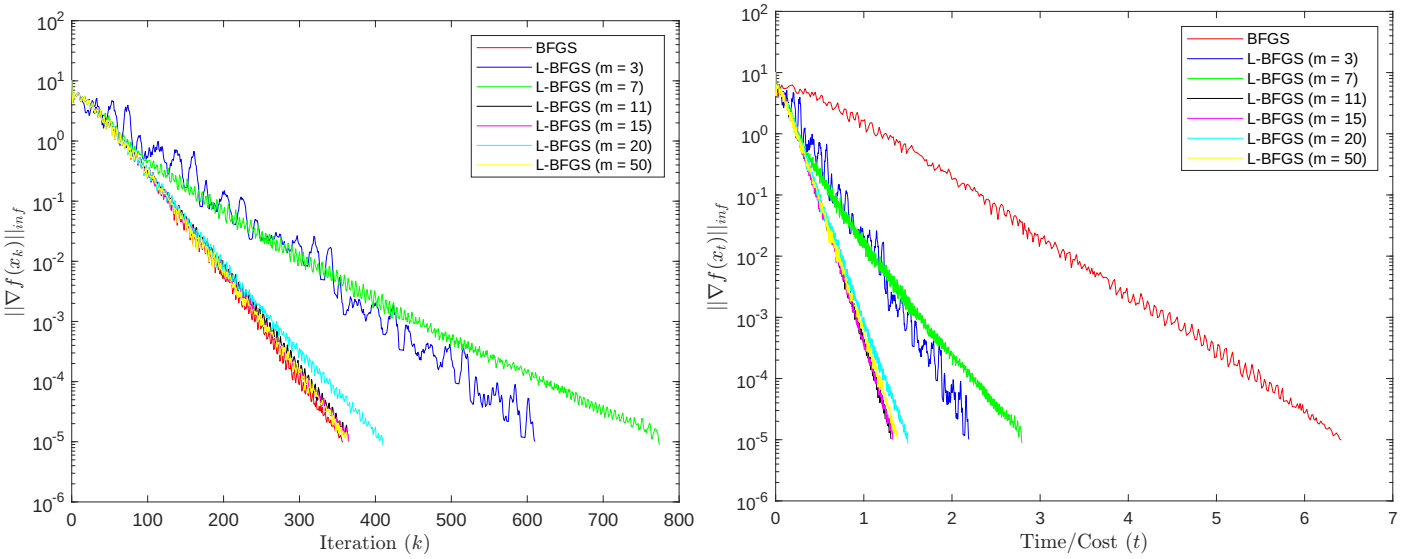
Figure 1: Performance Profiles of BFGS with backtracking line search and L-BFGS with different memory settings ($m = 3, 7, 11, 15, 20, 50$) Across All Problems. The comparison is visualized in terms of the number of iterations for the left figure and the running time for the right figure.

In Figure 1, we observe that BFGS with backtracking line search achieves the best performance in terms of iterations, owing to its better estimation of the Hessian matrix compared to L-BFGS with an limited memory size. However, when running time is utilized as the evaluation criterion, L-BFGS demonstrates superior performance and demands less storage space.

4.2 Convergence Analysis

In this section, we select three representative problems that are relatively difficult to compute (i.e., `P4_quad_1000_1000`, `Rosenbrock_100`, and `Genhump_100`) to show the convergence performance of BFGS with backtracking line search and L-BFGS with backtracking line with different memory settings ($m = 3, 7, 11, 15, 20, 50$). The Y-axis, $\|\nabla f(x)\|_{\text{inf}}$, represents the infinity norm of $\nabla f(x)$ at the current iteration k or current running time t.

4.2.1 P4_quad_1000_1000



(a) Convergence plot vs. Iteration

(b) Convergence plot vs. Execution Time

Figure 2: The performance comparison of BFGS with backtracking line search (red) and L-BFGS with different memory settings ($m = 3$ in blue, $m = 7$ in green, $m = 11$ in black, $m = 15$ in magenta, $m = 20$ in cyan, and $m = 50$ in yellow) for the `P4_quad_1000_1000` problem. The comparison is visualized in terms of the number of iterations (k) for the left figure and the running time (t) for the right figure.

From Figure 2 we can see that for the `P4_quad_1000_1000` problem, BFGS requires the least iterations to reach convergence. For L-BFGS with a relatively large memory size (such as $m = 20$ or 50), the number of iterations needed is similar to that of BFGS. When the memory size of L-BFGS is small, the number of iterations required to reach convergence will be larger; this is because L-BFGS uses only a subset of the historical data to update the Hessian matrix, resulting in a less precise approximation. However, L-BFGS drastically reduces memory demands by only keeping a few vectors that approximate the Hessian. In this specific problem, the memory demand is $2000m$ for L-BFGS with memory size m, while the memory demand for BFGS is 1000^2. Additionally, we also notice that each iteration of BFGS requires more computational time than L-BFGS, resulting in a greater total computational time for BFGS.

From Figure 2, it's obvious that the shortest total computational time is achieved when $m = 11$. However, we still aim to explore the relationship between the improvement in time efficiency and the increase in storage space for different values of m. We seek to identify the value of m for which the average increase in time efficiency per unit of memory size increment P_{inc} is maximized. Here, the BFGS method serves as the baseline, and the following equation is utilized to compute the index P_{inc} mentioned above:

$$P_{\text{inc}} = \frac{T_{\text{BFGS}} - T_{\text{m}}}{m \cdot T_{\text{BFGS}}} \cdot 100\%, \tag{1}$$

where T_{BFGS} denote the total computational time of BFGS, T_{m} denote the total computational time of L-BFGS

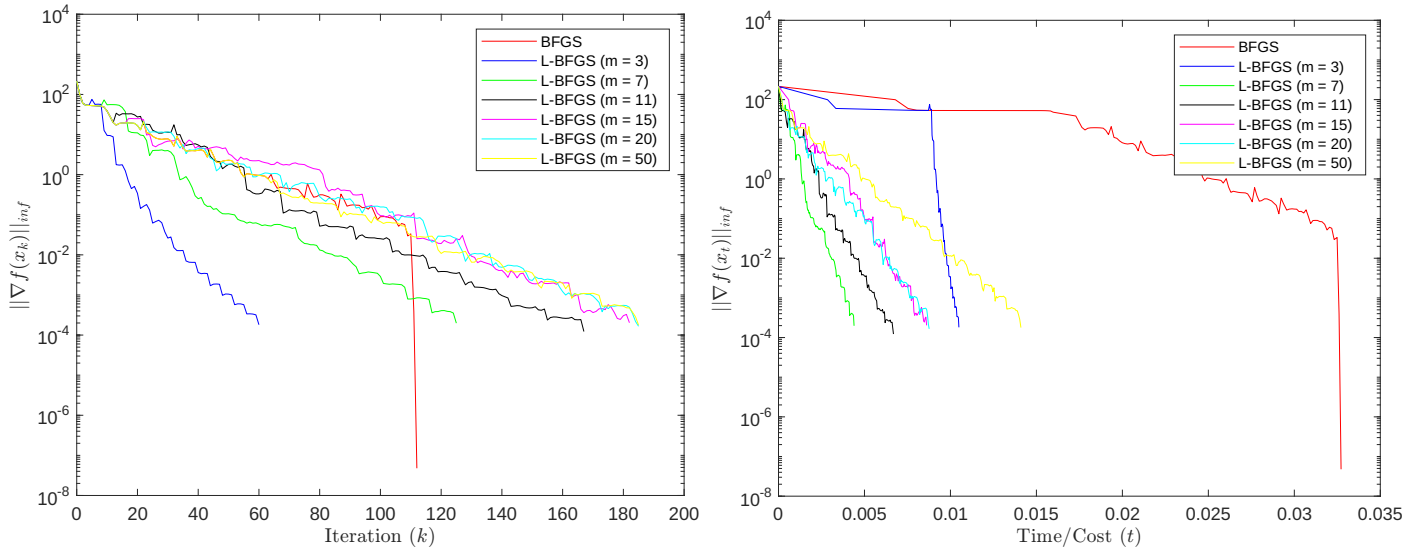
with different memory size m.

The results are shown in Table 1. We can see that for the problem `P4_quad_1000_1000`, the L-BFGS method with $m = 3$ demonstrates the highest P_{inc}, rather than the L-BFGS method with $m = 11$. Therefore, the L-BFGS method with $m = 3$ can be selected as the optimal choice when considering the trade-off between time efficiency and memory size.

	Memory size m					
	3	7	11	15	20	50
P_{inc}	23.12%	8.83%	7.41%	5.45%	3.96%	1.61%

Table 1: Average Increase in Time Efficiency per Unit of Memory Size Increment for the L-BFGS Method with Different Memory Sizes.

4.2.2 Rosenbrock_100



(a) Convergence plot vs. Iteration

(b) Convergence plot vs. Execution Time

Figure 3: The performance comparison of BFGS with backtracking line search (red) and L-BFGS with different memory settings ($m = 3$ in blue, $m = 7$ in green, $m = 11$ in black, $m = 15$ in magenta, $m = 20$ in cyan, and $m = 50$ in yellow) for the `Rosenbrock_100` problem. The comparison is visualized in terms of the number of iterations (k) for the left figure and the running time (t) for the right figure.

From Figure 3 we can see that for the `Rosenbrock_100` problem, although BFGS uses a more accurate Hessian approximation, L-BFGS with memory size 3 needs even less iterations than BFGS. However, it's important to note that the memory size of L-BFGS requiring minimum convergence time is 7, instead of 3. This could be because $m = 3$ necessitates more frequent adjustments of storage vectors, thus increasing the time cost per iteration. The current problem is computationally very fast, so convergence time can fluctuate across different

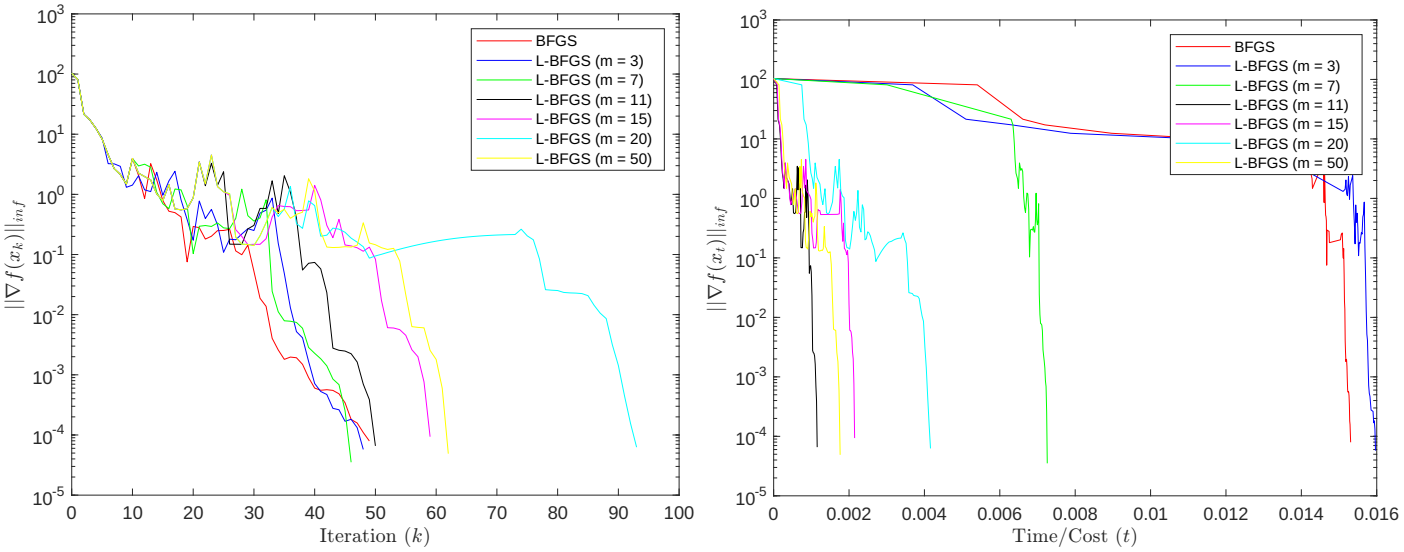
experiments.

The results for the index P_{inc} are shown in Table 2. We can see that for the problem `Rosenbrock_100`, the L-BFGS method with $m = 3$ demonstrates the highest P_{inc}, rather than the L-BFGS method with $m = 11$. Therefore, the L-BFGS method with $m = 3$ can be selected as the optimal choice when considering the trade-off between time efficiency and memory size.

	Memory size m					
	3	7	11	15	20	50
P_{inc}	14.69%	8.95%	7.16%	4.99%	3.59%	1.28%

Table 2: Average Increase in Time Efficiency per Unit of Memory Size Increment for the L-BFGS Method with Different Memory Sizes.

4.2.3 Genhump_5



(a) Convergence plot vs. Iteration

(b) Convergence plot vs. Execution Time

Figure 4: The performance comparison of BFGS with backtracking line search (red) and L-BFGS with different memory settings ($m = 3$ in blue, $m = 7$ in green, $m = 11$ in black, $m = 15$ in magenta, $m = 20$ in cyan, and $m = 50$ in yellow) for the `Genhump_5` problem. The comparison is visualized in terms of the number of iterations (k) for the left figure and the running time for the right figure.

From Figure 4 we can see that for the `Genhump_5` problem, BFGS requires the fewest iterations to converge. However, L-BFGS with a memory size of 3 or 7 demonstrates comparable performance in terms of iteration number. L-BFGS with a memory size of 11 also outperforms other algorithms in computational time for this particular problem. L-BFGS with a memory size of 3 has a computational time similar to that of BFGS. Like the previous `Rosenbrock_100` problem, the current problem computes very quickly, which may result in

variations in convergence time across different experiments.

The results for the index P_{inc} are shown in Table 3. We can see that for the problem `Genhump_5`, the L-BFGS method with $m = 11$ demonstrates the highest P_{inc}. Therefore, the L-BFGS method with $m = 11$ can be selected as the optimal choice in terms of time efficiency and memory size.

	Memory size m					
	3	7	11	15	20	50
P_{inc}	-1.60%	6.82%	8.40%	5.80%	4.19%	1.60%

Table 3: Average Increase in Time Efficiency per Unit of Memory Size Increment for the L-BFGS Method with Different Memory Sizes.

5 Discussion

This project presented a comprehensive evaluation of a series of optimization algorithms, each offering unique approaches to identifying the minima of various functions. Throughout the project, our team engaged in a methodical implementation process, which was both intellectually demanding and enlightening. We conclude that the Modified Newton Method (with Wolfe line Search) is the winner among all algorithms we implemented in this project. One of the key aspects that set the Modified Newton Method apart was its robust performance across all sets of problems. Even though accurately computing the Hessian matrix and its subsequent inversion may pose a substantial challenge for a novice coder; however, the resulting precision in descent direction and the rapid convergence rates affirmed the method's potential.

Furthermore, the algorithm's sensitivity to the initial guess and parameter settings led us to a deeper exploration of the methods for tuning these parameters. Specifically, for the DFP and DFPW algorithms, we need to fine-tune the line search parameters because the default parameters ($\alpha_0 = 1$, and $c_2 = 1e-2$ for DFPW) led to the step size reducing to zero after several iterations. This reduction prevented the algorithm from taking sufficient steps to converge optimally. After extensive testing on all problem sets, we adjusted the parameters to $\alpha_0 = 10$ and $c_2 = 0.5$. This adjustment allowed the algorithm to consistently converge to the optimal solution across all test problems. While this fine-tuning process added complexity to the implementation, it allows us to customize each optimization algorithm to meet specific problem requirements.

For practitioners with a strong background in coding and optimization, the Modified Newton Method is highly recommended due to its superior performance observed in our experiments. However, for practitioners who has difficulties computing the second derivatives, quasi-Newton methods offer a viable alternative. Among these, the L-BFGS method, notable for its memory efficiency, stands out as a practical choice. Nevertheless, it is crucial to fine-tune the memory size parameter (m) based on the specific problem at hand because the effectiveness of the method can vary significantly across different challenges. For instance, while L-BFGS with a

memory size of $m = 3$ is highly effective for problems like `Rosenbrock_100` and `P4_quad_1000_1000`, this setting did not yield optimal results for all problems. In cases such as the `Genhump_5` problem, a more substantial memory size of $m = 11$ was necessary to achieve the best performance. This variability underscores the importance of empirical testing and adjustment in the application of L-BFGS. Based on what we learned in this project, our advice is that practitioners should be prepared to engage in a trial-and-error process to determine the most effective memory size(as well as other important parameters) for their particular optimization problems.

Even though we choose the Modified Newton Method as the winner, it should not overshadow the merits of other algorithms. The Gradient Descent method, which is easier to understand and implement, would be useful for situations where simplicity and computational efficiency are more important.

In summary, our project emphasized the importance of matching the algorithm to the problem characteristics and the practitioner's expertise. It has reinforced the notion that there is no 'one-size-fits-all' in optimization; rather, a nuanced understanding of each method's strengths and limitations are important for selecting the best algorithm.

Appendices

A Summary of Results

A.1 Iterations

Problems	Algorithms									
	BFGS	**BFGSW**	**DFP**	**DFPW**	**Gradient Descent**	**Gradient DescentW**	**Newton**	**NewtonW**	**TRNewtonCG**	**TRSR1CG**
DataFit_2	14	16	20	21	530	530	6	11	7	13
Exponential_10	19	39	26	26	27	33	13	10	12	17
Exponential_1000	10	10	32	34	21	26	13	15	12	20
Genhumps_5	49	35	149	84	147	47	107	38	72	249
P1_quad_10_10	26	33	32	32	119	46	1	1	35	34
P2_quad_10_1000	56	71	193	198	1000	1000	1	1	89	119
P3_quad_1000_10	351	423	779	676	1000	1000	1	1	28	140
P4_quad_1000_1000	356	412	744	642	1000	1000	1	1	136	177
P5_quartic_1	3	3	10	10	2	2	2	2	2	3
P6_quartic_2	28	31	65	55	5	5	12	11	12	17
Rosenbrock_100	112	112	111	111	42	42	4	6	4	5
Rosenbrock_2	33	40	80	86	1000	1000	20	10	19	55

Table 4: Performance of 10 Algorithms Across 12 Problems in Terms of Total Number of Iterations.

A.2 Execution Time

Problems	Algorithms									
	BFGS	**BFGSW**	**DFP**	**DFPW**	**Gradient Descent**	**Gradient DescentW**	**Newton**	**NewtonW**	**TRNewtonCG**	**TRSR1CG**
DataFit_2	0.00116	0.00169	0.00117	0.00113	0.01715	0.01556	0.00595	0.00246	0.00568	0.00514
Exponential_10	0.00218	0.00404	0.00205	0.00219	0.00381	0.00375	0.00282	0.00173	0.00782	0.00939
Exponential_1000	0.00253	0.00305	0.00596	0.00748	0.00506	0.00599	0.00568	0.00667	0.02407	0.03647
Genhumps_5	0.00224	0.00186	0.00271	0.00234	0.00799	0.00522	0.01013	0.00273	0.01964	0.09212
P1_quad_10_10	0.03027	0.08613	0.04113	0.04247	0.11621	0.14442	0.02602	0.00375	0.53482	0.47070
P2_quad_10_1000	0.03917	0.18841	0.19958	0.23443	0.65513	2.70911	0.00311	0.00332	1.25613	1.56413
P3_quad_1000_10	11.31838	17.13954	7.18171	6.96455	2.00543	8.05536	0.05311	0.03027	3.01608	8.61414
P4_quad_1000_1000	11.96164	17.54431	6.99163	6.89333	2.14386	8.71653	0.02619	0.02954	13.98807	11.44019
P5_quartic_1	0.00096	0.00099	0.00093	0.00133	0.00219	0.00250	0.00542	0.00532	0.00859	0.00272
P6_quartic_2	0.00152	0.00210	0.00216	0.00208	0.00319	0.00294	0.00251	0.00227	0.00833	0.00849
Rosenbrock_100	0.01390	0.01350	0.01100	0.01214	0.00443	0.00406	0.00345	0.00283	0.01071	0.01255
Rosenbrock_2	0.00134	0.00168	0.00180	0.00186	0.02736	0.02120	0.00227	0.00223	0.00794	0.01599

Table 5: Performance of 10 Algorithms Across 12 Problems in Terms of Execution Time.

A.3 Function Evaluations

Problems	Algorithms									
	BFGS	**BFGSW**	**DFP**	**DFPW**	**Gradient Descent**	**Gradient DescentW**	**Newton**	**NewtonW**	**TRNewtonCG**	**TRSR1CG**
DataFit_2	37	144	100	126	3438	3968	13	75	295	547
Exponential_10	41	447	104	130	69	294	44	98	505	715
Exponential_1000	24	69	126	175	45	303	44	167	505	841
Genhumps_5	126	368	589	453	363	385	228	306	3025	10459
P1_quad_10_10	53	395	132	164	239	569	3	4	1471	1429
P2_quad_10_1000	113	851	714	1041	2001	12969	3	4	3739	4999
P3_quad_1000_10	703	5251	2884	3533	2001	12968	3	4	1177	5881
P4_quad_1000_1000	713	5119	2747	3407	2001	12964	3	4	5713	7435
P5_quartic_1	7	10	51	61	5	7	5	7	85	127
P6_quartic_2	125	414	300	325	78	83	25	122	505	715
Rosenbrock_100	1124	1236	1090	1201	504	546	9	39	169	211
Rosenbrock_2	86	317	325	451	10834	11883	48	70	799	2311

Table 6: Performance of 10 Algorithms Across 12 Problems in Terms of Function Evaluations.

A.4 Gradient Evaluations

Problems	Algorithms									
	BFGS	**BFGSW**	**DFP**	**DFPW**	**Gradient Descent**	**Gradient DescentW**	**Newton**	**NewtonW**	**TRNewtonCG**	**TRSR1CG**
DataFit_2	15	43	21	44	531	1061	7	27	15	534
Exponential_10	20	113	27	53	28	87	14	26	25	698
Exponential_1000	11	25	33	70	22	78	14	37	25	821
Genhumps_5	50	99	150	178	148	125	108	110	145	10210
P1_quad_10_10	27	97	33	65	120	138	2	3	71	1395
P2_quad_10_1000	57	211	194	425	1001	3001	2	3	179	4880
P3_quad_1000_10	352	1270	780	1452	1001	3001	2	3	57	5741
P4_quad_1000_1000	357	1237	745	1389	1001	3001	2	3	273	7258
P5_quartic_1	4	7	11	21	3	5	3	5	5	124
P6_quartic_2	29	90	66	114	6	11	13	34	25	698
Rosenbrock_100	113	225	112	223	43	85	5	15	9	206
Rosenbrock_2	34	103	81	180	1001	2002	21	26	39	2256

Table 7: Performance of 10 Algorithms Across 12 Problems in Terms of Gradient Evaluations.

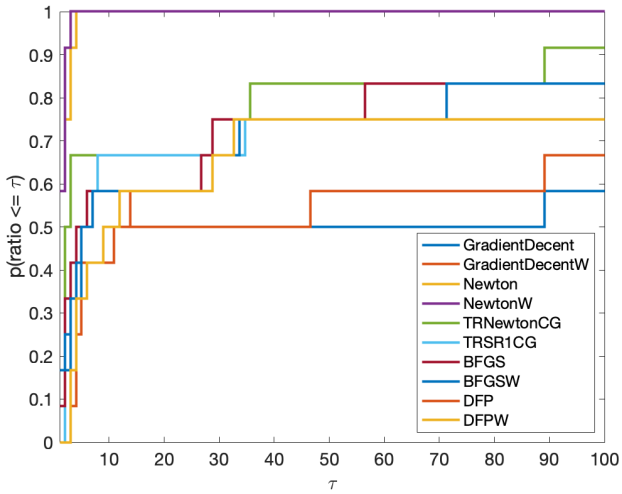
A.5 Final Objective Value

Problems	Algorithms									
	BFGS	**BFGSW**	**DFP**	**DFPW**	**Gradient Descent**	**Gradient DescentW**	**Newton**	**NewtonW**	**TRNewtonCG**	**TRSR1CG**
DataFit_2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Exponential_10	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056
Exponential_1000	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056	-0.2056
Genhumps_5	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
P1_quad_10_10	-67.9575	-67.9575	-67.9575	-67.9575	-67.9575	-67.9575	-67.9575	-67.9575	-67.9575	-67.9575
P2_quad_10_1000	-4449.2009	-4449.2009	-4449.2009	-4449.2009	-3957.0313	-4384.2388	-4449.2009	-4449.2009	-4449.2009	-4449.2009
P3_quad_1000_10	-71054.5918	-71054.5918	-71054.5918	-71054.5918	-68000.8831	-70774.0771	-71054.5918	-71054.5918	-71054.5918	-71054.5918
P4_quad_1000_1000	-71054.5918	-71054.5918	-71054.5918	-71054.5918	-68002.5967	-70775.4382	-71054.5918	-71054.5918	-71054.5918	-71054.5918
P5_quartic_1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
P6_quartic_2	0.0013	0.0012	0.0011	0.0019	0.0010	0.0010	0.0007	0.0006	0.0007	0.0007
Rosenbrock_100	3.9866	3.9866	3.9866	3.9866	3.9866	3.9866	3.9866	3.9866	3.9866	3.9866
Rosenbrock_2	0.0000	0.0000	0.0000	0.0000	0.0007	0.0003	0.0000	0.0000	0.0000	0.0000

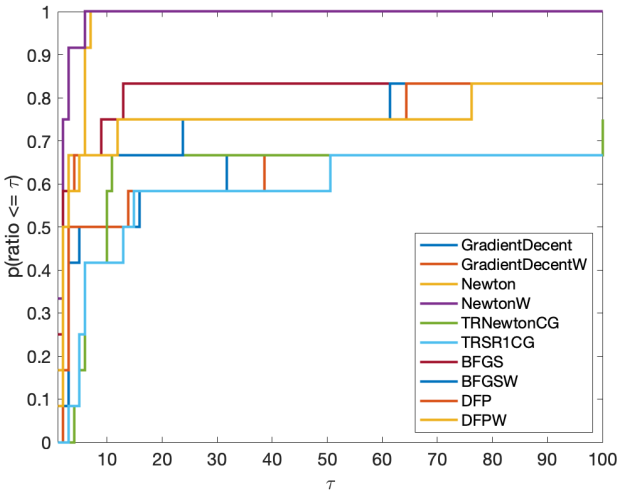
Table 8: Performance of 10 Algorithms Across 12 Problems in Terms of Final Objective Values.

Method	Default Options				
	Optimality tolerance:	Armijo line search parameters:	(Weak) Wolfe line search parameters:	TR radius update and CG parameters:	Modified Newton parameters:
	$\epsilon = 10^{-6}$, $max\_iterations = 10^3$	α_0, $\tau = 0.5$, $c\_1\_ls = 10^{-4}$;	α_0, $c\_1\_ls = 10^{-4}$, c_2_ls, $\alpha_{low} = 0$, $\alpha_{high} = 1000$, $c = 0.5$;	$c\_1\_tr = 0.3$, $c\_2\_tr = 1$, $term\_tol\_CG = 10^{-6}$, $max\_iterations\_CG = 40$, $\delta_0 = 1$;	$\beta = 10^{-6}$;
1. GradientDescent	✓	✓ $\alpha_0 = 1$			
2. GradientDescentW			✓ $\alpha_0 = 1$, $c\_2\_ls = 10^{-4}$,		
3. Newton		✓ $\alpha_0 = 1$			✓
4. NewtonW			✓ $\alpha_0 = 1$, $c\_2\_ls = 10^{-4}$		✓
5. TRNewtonCG				✓	
6. TRSR1CG				✓	
7. BFGS		✓ $\alpha_0 = 1$			
8. BFGSW			✓ $\alpha_0 = 1$, $c\_2\_ls = 10^{-4}$		
9. DFP		✓ $\alpha_0 = 10$			
10. DFPW			✓ $\alpha_0 = 10$, $c\_2\_ls = 0.5$		

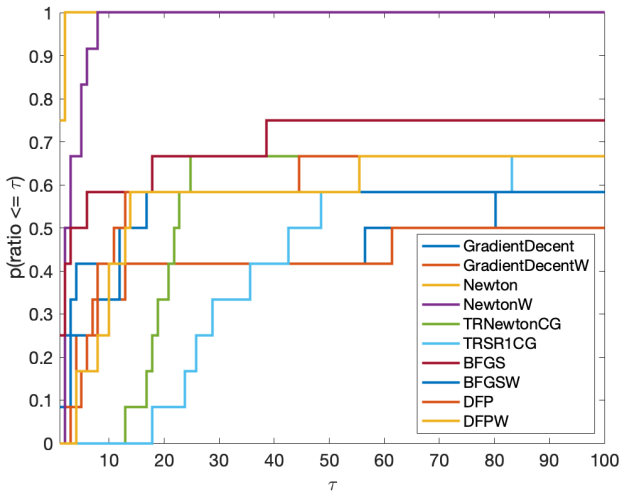
Table 9: Default Options for the 10 Algorithms. The ✓ symbol indicates that the corresponding algorithm uses the default parameter values specified above.



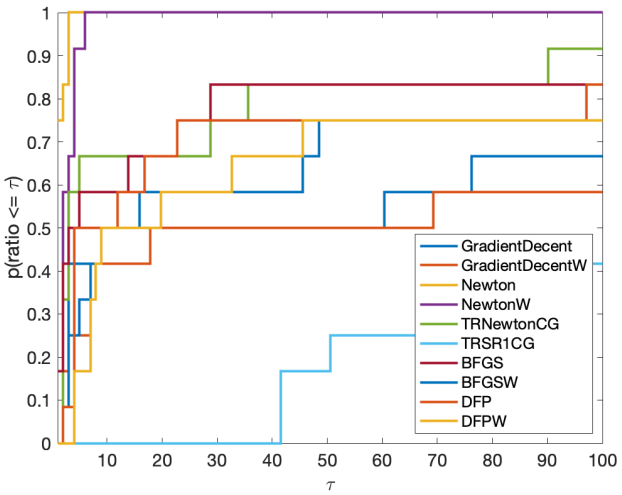
(a) Performance Profiles: Iteration



(b) Performance Profiles: Execution Time



(c) Performance Profiles: Number of Function Evaluations



(d) Performance Profiles: Number of Gradient Evaluations

Figure 5: Performance Profiles of 10 Algorithms Across All Problems.