MATH 562/IOE 511: Final Project

Hugh Van Deventer
hughv@umich.edu
Itamar Pres
presi@umich.edu

University of Michigan — Winter 2025

1 Algorithms

1. **GradientDescent (Backtracking Line Search):**
   This is a first-order line search method. It computes the step direction as the negative gradient ($p_k = -\nabla f(x_k)$). The step length α_k is determined using a backtracking procedure: starting with an initial guess (e.g., $\alpha_k = 1$) and iteratively reducing it until the Armijo sufficient decrease condition ($f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k)^T p_k$) is satisfied. It does not explicitly modify computations for nonconvexity but relies on finding decrease along the negative gradient direction to converge towards a stationary point.

2. **GradientDescentW (Weak Wolfe Line Search):**
   Also a first-order line search method, using the negative gradient $p_k = -\nabla f(x_k)$ as the search direction. The step length α_k is chosen to satisfy the weak Wolfe conditions. These consist of the Armijo sufficient decrease condition ($f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k)^T p_k$) and a curvature condition ensuring sufficient slope increase ($\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$), where $0 < c_1 < c_2 < 1$. This aims for a more effective step length than simple backtracking. Nonconvexity is handled similarly to the backtracking version.

3. **Newton (Modified, Backtracking Line Search):**
   This is a second-order line search method. The search direction p_k is computed by solving the Newton system $H_k p_k = -\nabla f(x_k)$, where $H_k = \nabla^2 f(x_k)$ is the Hessian matrix. To handle nonconvexity, if H_k is not positive definite, it is modified by adding a multiple of the identity matrix, $H_k \leftarrow H_k + \mu I$, to ensure the resulting matrix is positive definite, thus guaranteeing p_k is a descent direction. The step length α_k is then found via backtracking search (Armijo condition).

4. **NewtonW (Modified, Weak Wolfe Line Search):**
   A second-order line search method similar to the modified Newton with backtracking. It computes the step direction by solving $H_k p_k = -\nabla f(x_k)$, modifying the Hessian H_k if necessary to ensure positive definiteness and a descent direction. The step length α_k is determined by satisfying the weak Wolfe conditions (sufficient decrease and curvature: $\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$).

5. **TRNewtonCG (Trust Region Newton, CG Subproblem Solver):**
   This is a second-order trust region method. At each iteration k, it constructs a quadratic model $m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T H_k p$ using the exact Hessian H_k. The step p_k is found by approximately minimizing $m_k(p)$ within a trust region defined by $\|p\| \leq \Delta_k$,

where Δ_k is the trust region radius. This subproblem is solved using the Steihaug-CG method. CG inherently handles potential nonconvexity (indefinite H_k) by detecting negative curvature or stopping at the trust region boundary. The radius Δ_k is adjusted based on the agreement between the model and the actual function reduction.

6. **TRSR1CG (Trust Region SR1, CG Subproblem Solver):**
This is a quasi-Newton trust region method. It employs the Symmetric Rank-1 (SR1) update formula to maintain an approximation B_k to the Hessian matrix. The step p_k is computed by approximately solving the trust region subproblem $\min m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2} p^T B_k p$ subject to $\|p\| \leq \Delta_k$, using the Steihaug-CG algorithm. The SR1 update does not guarantee B_k is positive definite, but the CG subproblem solver is equipped to handle indefinite matrices, making the method suitable for nonconvex problems without explicit Hessian modification.

7. **BFGS (BFGS Quasi-Newton, Backtracking Line Search):**
A popular quasi-Newton line search method. It uses the Broyden–Fletcher–Goldfarb–Shanno (BFGS) update formula to iteratively build an approximation $H_k \approx B_k^{-1}$ to the inverse Hessian. The search direction is computed as $p_k = -H_k \nabla f(x_k)$. The BFGS update preserves positive definiteness if the initial matrix H_0 is positive definite and the curvature condition $y_k^T s_k > 0$ holds (where $s_k = x_{k+1} - x_k$ and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$). A simple backtracking line search finds the step length α_k. If the curvature condition fails, the update may be skipped.

8. **BFGSW (BFGS Quasi-Newton, Weak Wolfe Line Search):**
This quasi-Newton line search method also uses the BFGS update to approximate the Hessian (or its inverse) and computes the step direction $p_k = -H_k \nabla f(x_k)$. It employs a weak Wolfe line search to determine the step length α_k. Satisfying the weak Wolfe conditions (specifically the curvature condition $\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$) ensures that the curvature condition $y_k^T s_k > 0$ is met. This guarantees that the BFGS update preserves the positive definiteness of the Hessian approximation H_k, making the algorithm robust in handling nonconvexity by ensuring descent directions are generated.

9. **DFP (DFP Quasi-Newton, Backtracking Line Search):**
An earlier quasi-Newton line search method. It uses the Davidon–Fletcher–Powell (DFP) update formula, which primarily approximates the inverse Hessian H_k. The step direction is $p_k = -H_k \nabla f(x_k)$. Similar to BFGS, the DFP update preserves positive definiteness under the curvature condition $y_k^T s_k > 0$. Step length α_k is found via backtracking. DFP is generally considered less numerically stable and efficient than BFGS in modern practice.

10. **DFPW (DFP Quasi-Newton, Weak Wolfe Line Search):**
This quasi-Newton line search method uses the DFP update for the inverse Hessian approximation ($p_k = -H_k \nabla f(x_k)$). It employs a weak Wolfe line search for the step length α_k. As with BFGSW, the weak Wolfe conditions ensure the curvature condition $y_k^T s_k > 0$ holds (via the curvature condition $\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$), guaranteeing that the DFP update maintains positive definiteness.

2 Default Values

Table 1: Simplified Default Optimization Parameters

Parameter	Gradient Descent	Modified Newton	Trust Region	Quasi-Newton (BFGS/DFP)
`term_tol`	10^{-6}	10^{-6}	10^{-6}	10^{-6}
`max_iterations`	10^3	10^3	10^3	10^3
`alpha`	1	1	N/A	1
`tau`	0.5	0.5	N/A	0.5
`c_1_ls`	10^{-4}	10^{-4}	N/A	10^{-4}
`c_2_ls`	0.9	0.9	N/A	0.9
`beta (Newton regularize)`	N/A	10^{-6}	N/A	N/A
`c_1_tr`	N/A	N/A	10^{-3}	N/A
`c_2_tr`	N/A	N/A	0.75	N/A
`term_tol_CG`	N/A	N/A	10^{-10}	N/A
`max_iterations_CG`*	N/A	N/A	n*	N/A
`epsilon_sy`	N/A	N/A	10^{-6} (SR1)	10^{-6}

Notes: (SR1) applies only to TR-SR1-CG variant. N/A: Not Applicable. *Default for `max_iterations_CG` is equal to the dimension of the input.

3 Performance

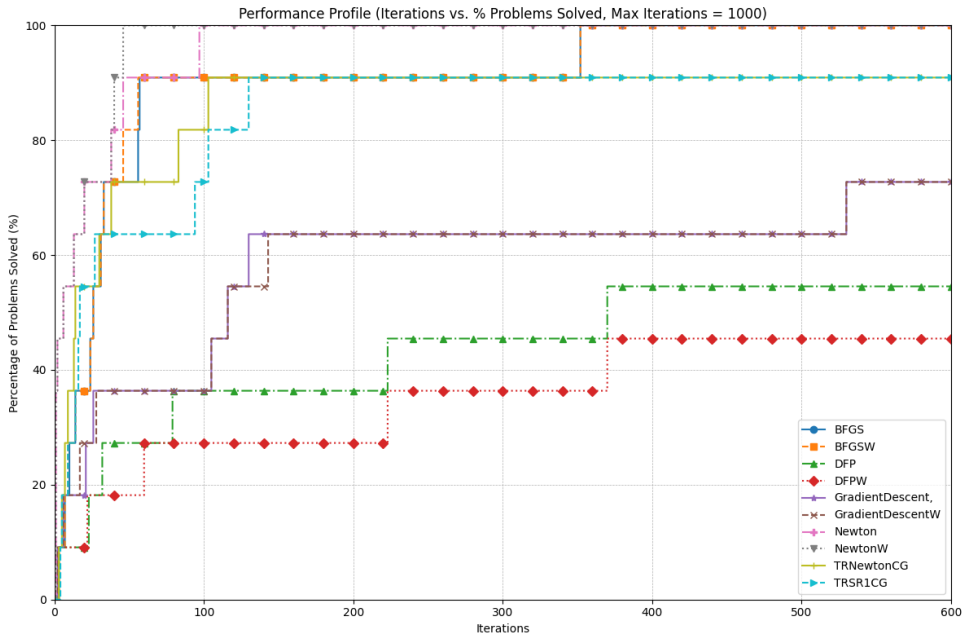

Figure 1: Performance Profile

Table: Iterations Summary

method	BFGS	BFGSW	DFP	DFPW	GradientDescent,	GradientDescentW	Newton	NewtonW	TRNewtonCG	TRSR1CG
problem										
datafit_2	14	14	79	60	530	530	6	6	7	27
exponential_10	19	20	1000	1000	27	27	13	13	13	16
exponential_100	10	9	1000	1000	21	17	13	13	14	29
genhumps_5	57	46	1000	1000	130	143	97	40	83	130
quad_1000_10	31	31	370	370	116	116	1	1	9	9
quad_1000_1000	352	352	1000	1000	1000	1000	1	1	103	103
quad_10_10	26	26	223	223	105	105	1	1	5	5
quad_10_1000	56	56	1000	1000	1000	1000	1	1	14	14
quartic_1	3	3	3	3	2	2	2	2	3	4
quartic_2	24	24	32	1000	6	6	38	38	38	17
rosenbrock_100	7	7	1000	1000	26	28	46	46	1000	1000
rosenbrock_2	33	33	23	22	1000	1000	20	20	30	94

Table: Function Evaluations Summary

method	BFGS	BFGSW	DFP	DFPW	GradientDescent,	GradientDescentW	Newton	NewtonW	TRNewtonCG	TRSR1CG
problem										
datafit_2	6982	7019	7228	7400	3438	6876	6889	6902	6917	6945
exponential_10	311	362	36300	65252	69	138	182	226	253	270
exponential_100	260	288	36058	64877	45	89	133	177	206	236
genhumps_5	1613	1744	37677	66624	376	837	1052	1200	1343	1474
quad_1000_10	564	627	1368	2109	233	466	469	472	491	501
quad_1000_1000	6206	6911	8912	10913	2001	5184	5187	5190	5397	5501
quad_10_10	498	551	998	1445	211	422	425	428	439	445
quad_10_1000	5359	5472	7473	9474	2001	5196	5199	5202	5231	5246
quartic_1	39	46	53	60	5	10	15	20	27	32
quartic_2	548	659	860	3014	94	188	265	342	419	437
rosenbrock_100	2882	2906	14429	41199	279	584	683	782	1857	2858
rosenbrock_2	22044	22130	22210	22289	10834	21717	21765	21813	21863	21958

Table: Gradient Evaluations Summary

method	BFGS	BFGSW	DFP	DFPW	GradientDescent,	GradientDescentW	Newton	NewtonW	TRNewtonCG	TRSR1CG
problem										
datafit_2	1652	1681	1761	1882	531	1592	1599	1612	1620	1637
exponential_10	174	216	1217	2220	28	83	97	124	138	154
exponential_100	145	165	1166	2174	22	58	72	99	114	134
genhumps_5	809	905	1906	2909	131	421	519	611	671	751
quad_1000_10	407	470	841	1582	117	350	352	355	365	375
quad_1000_1000	3765	4470	5471	7472	1001	3199	3201	3204	3308	3412
quad_10_10	361	414	638	1085	106	317	319	322	328	334
quad_10_1000	3293	3406	4407	6408	1001	3201	3203	3206	3221	3236
quartic_1	29	36	40	47	3	8	11	16	20	25
quartic_2	218	267	300	2298	7	20	59	136	175	193
rosenbrock_100	320	335	1336	9127	27	85	132	225	300	312
rosenbrock_2	3173	3240	3264	3310	1001	3003	3024	3065	3085	3139

Table: CPU Time (seconds) Summary

method	BFGS	BFGSW	DFP	DFPW	GradientDescent,	GradientDescentW	Newton	NewtonW	TRNewtonCG	TRSR1CG
problem										
datafit_2	0.001s	0.002s	0.005s	0.004s	0.035s	0.048s	0.001s	0.001s	0.001s	0.001s
exponential_10	0.001s	0.002s	0.381s	0.330s	0.001s	0.001s	0.017s	0.002s	0.002s	0.002s
exponential_100	0.005s	0.005s	1.359s	1.062s	0.002s	0.003s	0.004s	0.004s	0.003s	0.005s
genhumps_5	0.006s	0.006s	0.578s	0.458s	0.008s	0.015s	0.024s	0.012s	0.012s	0.020s
quad_1000_10	6.146s	7.039s	73.934s	89.337s	12.699s	17.035s	0.296s	0.325s	1.714s	1.259s
quad_1000_1000	65.551s	79.030s	165.912s	196.274s	116.630s	201.013s	0.282s	0.311s	18.646s	11.511s
quad_10_10	0.070s	0.074s	0.421s	0.421s	0.241s	0.329s	0.005s	0.009s	0.016s	0.008s
quad_10_1000	0.078s	0.118s	1.396s	1.870s	1.390s	2.468s	0.003s	0.003s	0.032s	0.019s
quartic_1	0.000s	0.001s	0.000s	0.000s	0.000s	0.001s	0.000s	0.000s	0.001s	0.000s
quartic_2	0.003s	0.003s	0.004s	0.081s	0.001s	0.001s	0.005s	0.007s	0.007s	0.002s
rosenbrock_100	0.006s	0.006s	1.411s	3.538s	0.025s	0.030s	0.032s	0.042s	0.207s	0.142s
rosenbrock_2	0.002s	0.002s	0.002s	0.001s	0.065s	0.073s	0.001s	0.002s	0.002s	0.005s

Figure 2: Table: Summary of Results (red means convergence criterion not met)

4 Discussion

Newton (with both line searches) is the closest to "consistently" being the best, with BFGS and the trust region methods being close seconds. Newton converged on every problem and was among the fastest other than for genhumps_5 and rosenbrock_100.

Distinct performance differences arose on more challenging problems. Gradient Descent consistently required significantly more iterations and CPU time, particularly struggling with ill-conditioned quadratics and complex functions like Rosenbrock, often hitting the iteration limit due to its slow linear convergence rate. DFP variants also frequently underperformed compared to BFGS, taking substantially more time and iterations, and hitting the iteration limit on ill-conditioned quadratics and non-linear functions. This aligns with the known theoretical limitations of DFP, which can be less robust and efficient than BFGS, especially when dealing with ill-conditioning or requiring many iterations. Even robust methods like TRNewtonCG occasionally hit iteration limits on difficult problems like Rosenbrock, highlighting the inherent challenge posed by some optimization landscapes regardless of the algorithm used.

4.1 Algorithm of choice

We select NewtonW as the algorithm of choice, balancing observed performance metrics effectively. Its primary strength lies in its rapid convergence, consistently requiring significantly fewer iterations than other methods like Gradient Descent or DFP, particularly on quadratic problems, by leveraging second-order information. This rapid convergence in iterations often translated to competitive or superior overall CPU times, despite the higher computational cost per iteration associated with Hessian calculation and processing. While Quasi-Newton methods like BFGS and TRSR1CG sometimes achieved faster wall-clock times on larger problems due to their lower cost per iteration, Modified Newton demonstrated superior robustness in these tests, successfully solving all problems without hitting iteration limits. Therefore, given its exceptional performance in minimizing iterations and its reliability across the tested problems, Modified Newton represents the most compelling choice when Hessian information is available and computationally manageable.

5 The Big Question

As the vibecoders, we decided to investigate the big question of "How does memory affect the performance of quasi-Newton methods?" Memory refers to the number of curvature pairs $\{s, y\}$ pairs we are storing to update our Hessian. We investigate the trade-offs between storing more curvature pairs and the resulting quality of the Hessian approximation on our problems. Specifically, we run L-BFGS with backtracking with memory sizes of $[1, 3, 5, 7, 10, 15, 20]$ on the project problems and record iteration history.

5.1 Quadratic Functions

On the Quadratic Function, all L-BFGS methods achieved essentially the same function value. Thus, we will only view iteration vs memory size plots as the same trends are found for the number of f-evals and g-evals vs memory size.

From Figure 3, we see that performance gain from increasing memory is stronger when the condition number is higher. For quad_10_10 and quad_1000_10 we see that there is not much change from the graph of iteration reduction when we go from $n = 10$ to $n = 1000$. The greatest performance gain is from $m = 1$ to $m = 3$ and then we quickly see diminishing gains. We see the importance of larger memory in quad_1000_1000, where we obtain strong performance gains

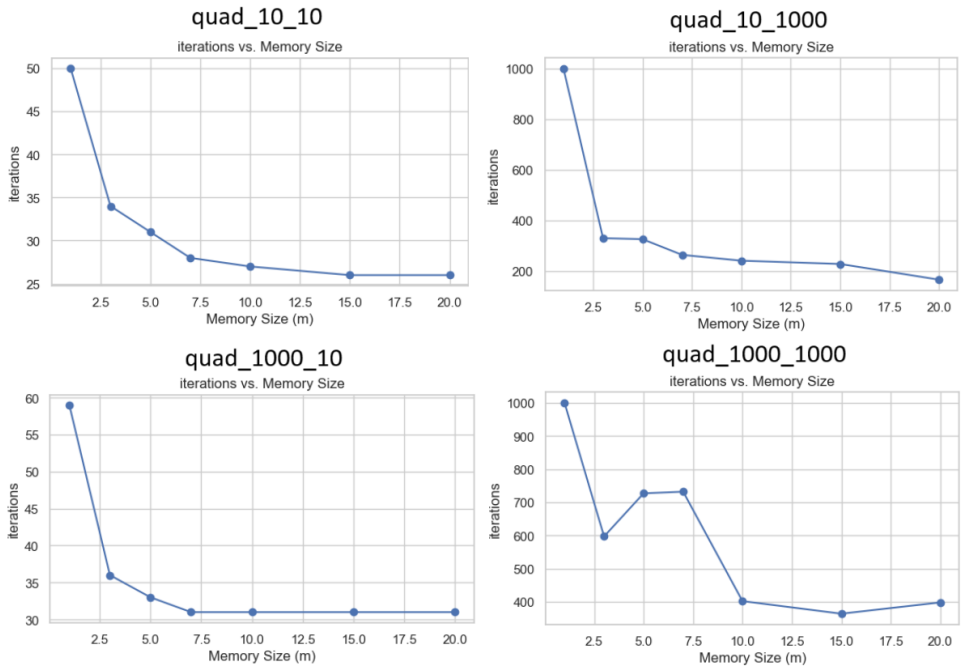

Figure 3: L-BFGS with varying memory sizes on the Quadratic problems

at $m = 10$. This highlights that for ill-conditioned quadratic problems, investing in larger memory sizes for L-BFGS yields substantial performance improvements, whereas well-conditioned problems benefit little beyond minimal memory.

5.2 Quartic Functions

On the quartic_1 function, all L-BFGS methods converged in 3 iterations. There was no significant different across all memory sizes, likely because the quartic component has a very small constribution ($\sigma = 10^{-4}$), meaning that the curvature (captured by the Hessian) plays little role in minimizing the function.

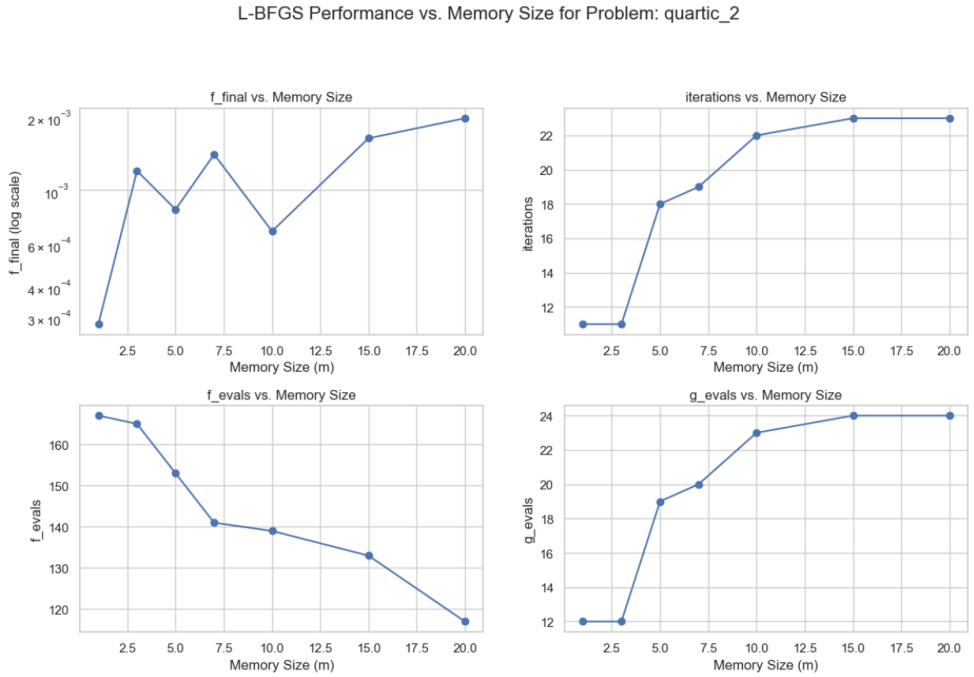

Figure 4: L-BFGS with varying memory sizes on the quartic_2 problem

Conversely, for quartic_2 where the large $\sigma = 10^4$ makes the quartic term dominant, Figure 4 reveals a strong and complex dependence on memory size. Increasing memory leads to significantly more iterations and gradient evaluations, yet paradoxically fewer function evaluations, suggesting intricate interactions between the Hessian approximation quality and the line search performance on this highly non-quadratic problem.

5.3 The Rosenbrock Problem

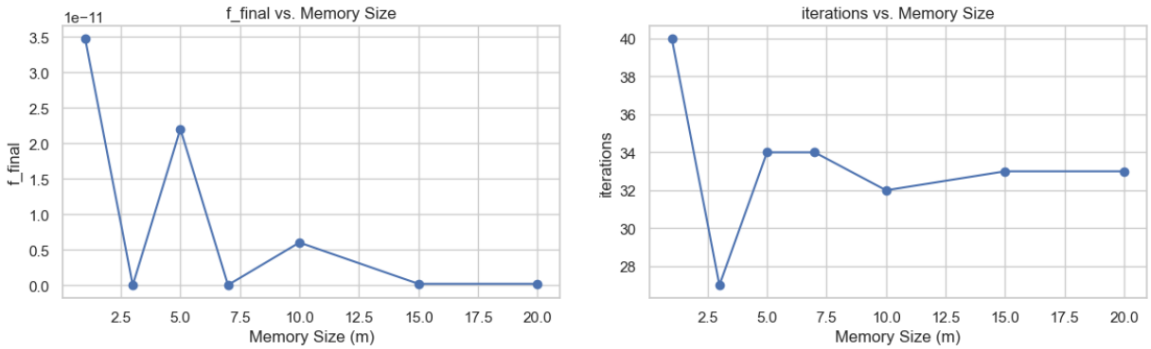

Figure 5: L-BFGS with varying memory sizes on the Rosenbrock problem (n=2)

For the 2-dimensional Rosenbrock problem, Figure 5 shows significant variability in performance depending on the L-BFGS memory size m. Both the final function value and the number

of iterations behave non-monotonically as m increases. Notably, memory sizes $m = 3, 7, 15, 20$ successfully converge to the minimum with near-zero final function values, while $m = 1, 5, 10$ terminate significantly further away, despite requiring a similar number of iterations in some cases (e.g., compare $m = 5$ and $m = 7$). The fastest convergence to the minimum occurred with $m = 3$ (27 iterations).

This contrasts sharply with the rosenbrock_100 results (not shown), where all memory sizes yielded identical performance, converging in only 7 iterations but to a point far from the true minimum ($f \approx 4.5$). This suggests the rosenbrock_100 run terminated prematurely based on gradient tolerance, likely before the effects of different memory sizes could manifest, unlike the more revealing optimization process in the 2D case which clearly shows sensitivity to the memory parameter.

5.4 Exponential Functions

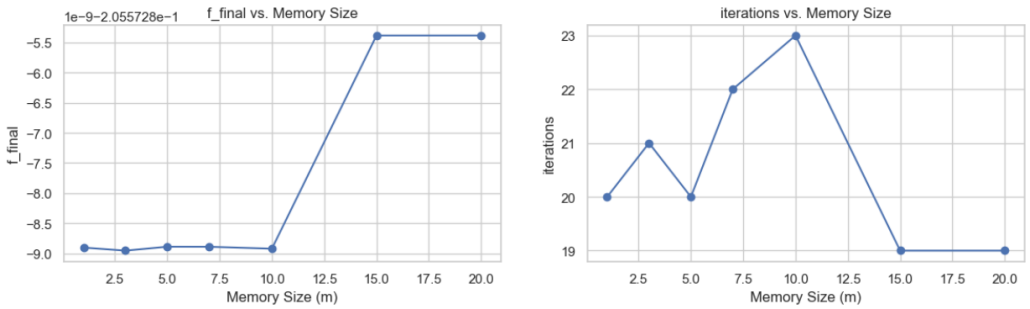

Figure 6: L-BFGS with varying memory sizes on the exponential_10 problem

For the exponential_10 problem, Figure 6 shows relatively consistent performance across different L-BFGS memory sizes m, with only mild variability. The final function value remains nearly constant and very close to the true minimum for all tested values of m, indicating that memory size has little impact on solution accuracy. However, the number of iterations needed to converge exhibits non-monotonic behavior as m varies.

Specifically, iteration counts peak around $m = 10$ before dropping sharply for larger memory sizes ($m = 15, 20$), where convergence occurs in fewer than 20 iterations. Nonetheless, because the final function values are uniformly excellent across all settings, the effect of m is primarily on efficiency rather than solution quality for this problem. Overall, L-BFGS demonstrates robustness to memory size variation on the exponential_10 function.

We do not include plots for exponential_100 for similar reasons as rosenbrock_100. Performance metrics were constant across all memory sizes, converging rapidly and precisely in 10 iterations. Thus, while L-BFGS successfully minimized the function value for both dimensions across tested memory sizes, the 10-dimensional problem reveals sensitivity to m in terms of computational effort (iterations) and final precision achieved, effects not observed in the higher-dimensional run likely due to its structure or rapid convergence characteristics.

5.5 Data Fit Problem

From Figure 7, we observe that all tested memory sizes successfully converge to the minimum function value, achieving f_{final} on the order of 10^{-11}. However, significant differences emerge in efficiency and final gradient precision. The $m = 1$ case is notably inefficient, requiring considerably more iterations (≈ 26) and evaluations (≈ 140) while achieving relatively poor final precision ($norm\_g_{final} \approx 10^{-2.5}$). Memory sizes $m \geq 3$ are much more efficient in terms of iterations and

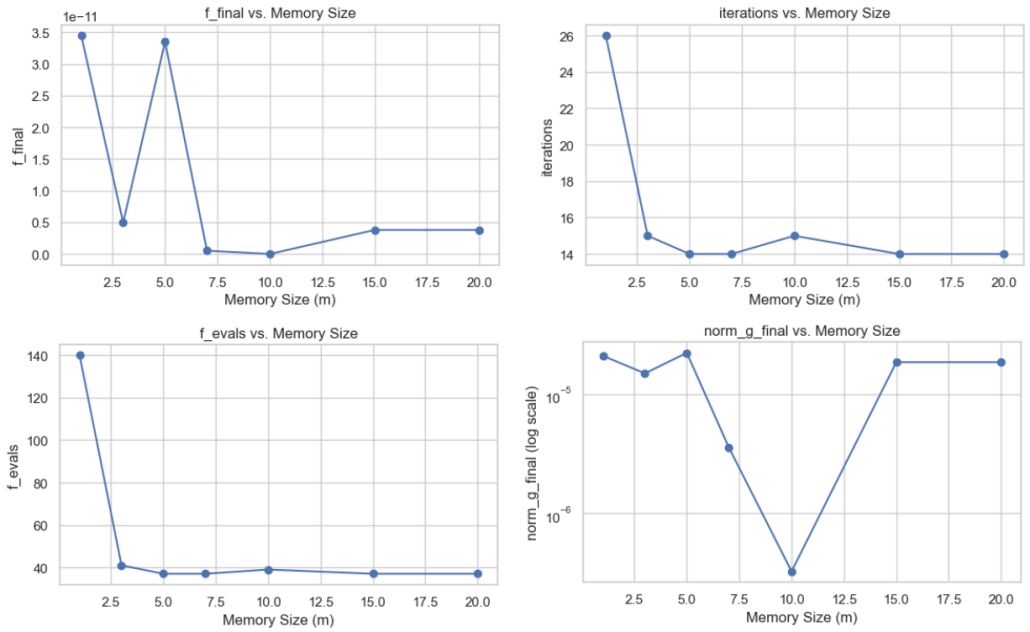

Figure 7: L-BFGS with varying memory sizes on the Datafit-2 problem

evaluations (13-15 iterations, $\approx$ 40 f_evals). Among these efficient runs, $m = 3, 7, 10$ demonstrate superior final precision according to the gradient norm, with $m = 10$ achieving the lowest value (near $10^{-4.5}$). Interestingly, $m = 5$ and the larger memory sizes $m = 15, 20$, despite their efficiency in iteration counts, result in substantially worse final precision ($norm\_g_{final} \approx 10^{-2.5}$), similar to $m = 1$. Therefore, for the Datafit_2 problem, while nearly all memory sizes find the minimum value, intermediate memory sizes ($m = 7$ or $m = 10$ appear best here) offer the optimal combination of low iteration/evaluation counts and high final precision in the gradient norm.

5.6 Genhumps Problem

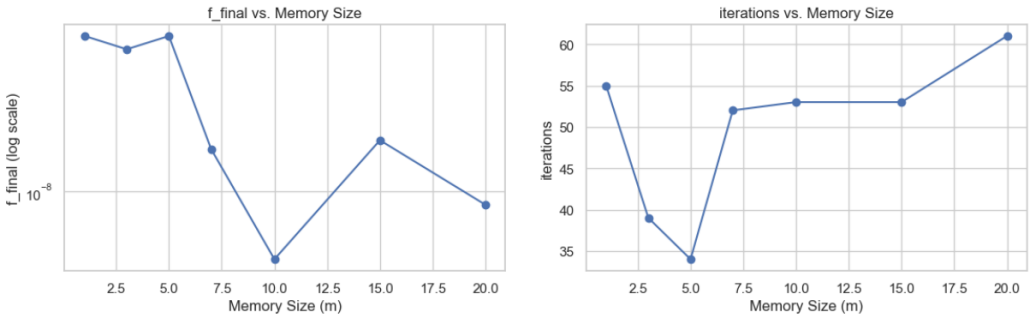

Figure 8: L-BFGS with varying memory sizes on the Genhumps_5 problem

From Figure 8, we see Genhumps_5 performance depends strongly and non-monotonically on memory size m. Both the final function value and iteration counts vary significantly. Memory $m = 10$ achieved the best f_{final} by an order of magnitude compared to other values, but required relatively high iterations (≈ 53). In contrast, $m = 5$ was fastest (≈ 34 iterations) but resulted in a worse f_{final}. This indicates performance on the complex Genhumps landscape is highly sensitive to m, with $m = 10$ being optimal for solution quality among tested values.

5.7 Conclusion

This investigation into the L-BFGS memory parameter m reveals that its impact on performance is highly dependent on the specific optimization problem's characteristics, including conditioning, non-linearity, and potentially dimension interacting with convergence behavior. For convex quadratic problems, increased memory primarily benefits ill-conditioned cases by significantly reducing iterations, while well-conditioned problems show diminishing returns beyond small m. On non-quadratic problems, the behavior varies widely: some functions with strong non-linearity or complex landscapes (Genhumps_5, Datafit_2, Exponential_10, Rosenbrock_2, Quartic_2) exhibit significant sensitivity, often with non-monotonic performance where intermediate memory sizes frequently provide the best results and large m can sometimes be detrimental to solution quality or precision. Conversely, certain problems, including the near-quadratic Quartic_1 and high-dimensional runs that converged very rapidly (Rosenbrock_100, Exponential_100), showed minimal sensitivity to memory size within the tested range, likely because the optimization process concluded before differences in the Hessian approximation quality could substantially alter the outcome. Ultimately, while larger memory theoretically allows for better Hessian approximations, potentially improving convergence, this benefit must be weighed against increased computational cost per iteration and does not guarantee better practical performance; no single m value is universally optimal, but moderate values often represent a reasonable starting point, with problem-specific tuning potentially offering significant gains.

6 Comments

This project was fun and provided valuable hands-on experience implementing and comparing a range of continuous optimization algorithms. Based on a combination of performance, ease of implementation, and parameter tuning effort, the Modified Newton method with a Weak Wolfe line search stands out as the overall "winner" for this problem set. Coding the Modified Newton logic, particularly leveraging a completed Cholesky factorization to handle potential indefiniteness and efficiently solve the Newton system, proved relatively straightforward. Implementing the Weak Wolfe line search was simple, and importantly, it required minimal tuning, as standard parameters recommended in literature (e.g., by Nocedal & Wright) yielded consistently good performance across the test problems. While Gradient Descent was simpler to code, its performance was generally poor. Trust-region methods offered strong performance but involved greater implementation complexity, particularly managing the subproblem solve. Quasi-Newton methods struck a balance but didn't consistently outperform the well-implemented Modified Newton in these tests.

For non-experts or those needing a quick implementation, basic Gradient Descent (accepting its slow convergence) or perhaps a standard Newton method (if the Hessian is easily available and well-behaved) might be simplest. However, for expert coders seeking high performance, Modified Newton (potentially upgrading to a Strong Wolfe search or using a Trust-Region approach like TRNewtonCG) is highly recommended when second derivatives are computationally feasible. If coding second derivatives is impractical or too costly, robust Quasi-Newton methods like BFGS or TRSR1CG become the algorithms of choice, offering a good compromise between performance and computational demands.