ISYE 6663 PROJECT REPORT

BRANT CALLAWAY, JOHN ROGERS

1. Project Implementation

For our project, we implemented the Gradient Descent method, two versions of Conjugate Gradient method (Fletcher-Reeves and Polak-Ribiere), the Quasi-Newton BFGS method, and the limited memory BFGS method. In order to test our algorithms, we implemented four functions with known properties. These were the Rosenbrock function, the Trigonometric function, Wood function, and a large quadratic function. Exact line search was used with the quadratic problem, and a strong Wolfe-Powell inexact line search was used on all of the problems.

1.1. Optimization Routine Implementation.

Both of the conjugate gradient methods (CGFR and CGPR) often did not converge to sufficient precision on some of the harder nonlinear problems. This is due to round off errors from finite precision computer arithmetic as well as local quadratic approximation of the function changes for nonlinear problems as we move from one step to the next. Each of the search directions explored by CG is Q-Conjugate to the prior search directions. Since we are using an inexact line search procedure, one search along this direction may not be sufficient to reach the global minimum. To combat this shortcoming, we reset to a gradient descent step whenever two consecutive gradients are not sufficiently orthogonal, as was suggested in the textbook. This effectively resets the conjugate-gradient method to allow it to minimize along all search directions again.

The L-BFGS quasi newton optimization algorithm was implemented with a variable amount of memory storage. Our original implementation had a fixed history of 5, but we chose to implement variable history to perform an analysis of how the history size affects the algorithm performance. Our expectation is that the optimal history size is problem dependent. That is, generally, the number of iterations required to find a solution decreases with the an increasing history size; however, the amount of storage increases (and hence the cost of each iteration) with increasing history size. Thus, whichever of these effects is stronger will determine what the optimal history size to choose will be, and this depends on the function being minimized.

1.2. Test Function Implementation.

Another issue that we had was in testing the Trigonometric function. We had major concerns that we had implemented the function wrong or not gotten the gradient correct. Finally, we found that from the starting point given($xo = (1/n, 1/n, \ldots, 1/n)$) in the notes, there is a local minimum that occurs between that starting point and what the notes give as the correct solution $x* = (0, 0, \ldots, 0)$. To further illustrate the point, here are the graphs for the trigonometric function with n=1 in figure 1 and n=2 in figure 2. We can see that for $n = 1$, there is a local minimum close to 0.927 which is where our each of our algorithms terminate (and further notice that this occurs between $x^* = 0$ and $x_o = 1/1 = 1$. Similarly, for $n = 2$, although slightly more difficult to see in the graph, we find a local minimum at $x = (0.243, 0.613)$ rather than at $x^* = (0, 0)$. In our results section, we will present the convergence to this local minimum, plus the convergence to the true global minimum from a different starting point.

We also implemented the Wood function from the test problems sheet and we run on the Rosenbrock function which was provided. In addition, we prepared a large quadratic function primarily to test the exact line search performance compared to inexact line search. The Rosenbrock function was tried with multiple sizes up to 20 dimensions, as was the Trigonometric function.

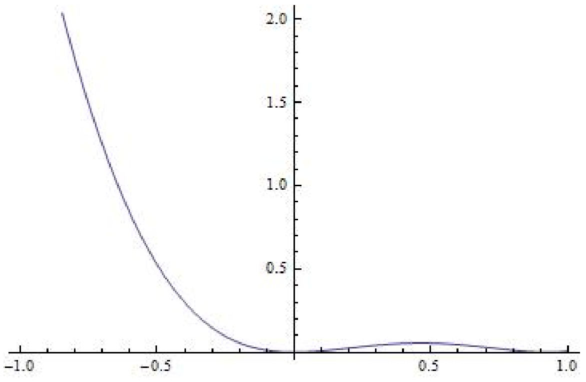

Figure 1. The trigonometric function, shown in 1 dimension. The optimization routine gets stuck in a local minimum.

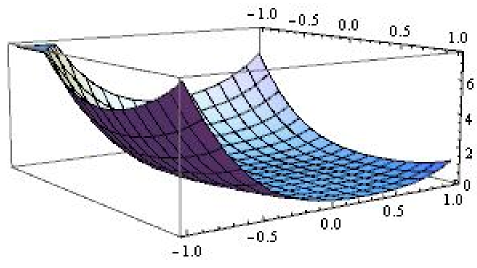

Figure 2. The trigonometric function, shown with 2 dimensions. The optimization routine gets stuck in a local minimum.

2. Results

The number of iterations and CPU time are compared between each of the tested algorithms on each of the test problems. In addition, the number of restarts are presented for the CG methods. The gradient descent algorithm is usually only able to reach a lower precision than the other routines due to its inefficiency with poorly conditioned problems. All graphs are presented in a logarithmic scale to better illustrate the progress of the algorithms.

2.1. **Quadratic Results.**

The optimization routines were first tried on a quadratic problem of the form $minx^T Ax - b^T x$. The matrix A was generated by the rand function in Matlab and made to be positive definite by computing $A^T A$. The vector b was also computed using the rand function. The results for exact line search can be seen in table 1. Comparing these results to table 2, we see that the gradient descent routine achieves the desired precision in exactly the same number of iterations, though exact line search is much quicker than inexact line search for CPU time. Further inspection led us to realize that the inexact line search routine was generating the exact same α_k as the exact line search routine. The difference in CPU time is due to the simplicity in the exact line search – gradient descent spends a lot of CPU cycles on inexact line search. A few other notable differences are that the conjugate gradient methods terminate in about 20 iterations and don't require restarts when using exact line search. When they use inexact line search, more iterations are necessary and restarts are also used multiple times. Our implementation of strong Wolfe-Powell appears to generate exact line search results when the gradients are parallel to the search directions, so since the CG methods do not use gradient directions the two line search routines generate different α_k. The BFGS routine in figure 3 terminates in 21 iterations, which makes sense because after the first 20 iterations the Hessian approximation is exact and the final step is a Newton step. On a quadratic function, this results in the solution being reached in only one more iteration. This is slightly faster than the result with inexact line search in figure 4.

2.2. **Trigonometric Results.**

The Trigonometric function from the test problems sheet presented us with more difficulty than the other functions. For the start point specified in the problem sheet, all of the optimization routines become stuck on a suboptimal local minimum, as was shown in figures 1 and 2. The algorithms get stuck here with $\eta = 1e - 8$ pretty quickly as can be seen in Table 3. We used a different starting point which is closer to the global optimum point and achieved rapid convergence in Table 4.

2.3. **Wood Results.**

The Wood function was a good test for the Gradient Descent algorithm because it was able to converge to the same η value in a reasonable amount of time as the more advanced optimization routines. Table 5 shows the results which show BFGS as clearly superior with LBFGS with limited memory

TABLE 1. Experimental Results - Quadratic Function, exact line search, 20 dimensional, $\eta = 1e - 4$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	61368	15.776	0	
CG-FR	26	0.110	0	
CG-PR	26	0.078	0	
BFGS	21	0.081	0	
L-BFGS-5	496	0.427	0	
L-BFGS-7	392	0.537	0	
L-BFGS-9	284	0.482	0	
L-BFGS-11	321	0.804	0	

TABLE 2. Experimental Results - Quadratic Function, inexact line search, 20 dimensional, $\eta = 1e - 4$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	61368	41.357	0	
CG-FR	77	0.157	5	
CG-PR	202	0.238	14	
BFGS	24	0.140	0	
L-BFGS-5	310	0.490	0	
L-BFGS-7	330	0.648	0	
L-BFGS-9	413	0.899	0	
L-BFGS-11	268	0.771	0	

TABLE 3. Experimental Results - Trigonometric Function 10 dimensional, 1/n starting point, $\eta = 1e - 8$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	167	0.087	0	Stuck
CG-FR	54	.150	15	Stuck
CG-PR	49	.152	17	Stuck
BFGS	24	.139	0	Stuck
L-BFGS-5	59	0.181	0	Stuck
L-BFGS-7	66	0.196	0	Stuck
L-BFGS-9	77	0.24	0	Stuck
L-BFGS-11	80	0.274	0	Stuck

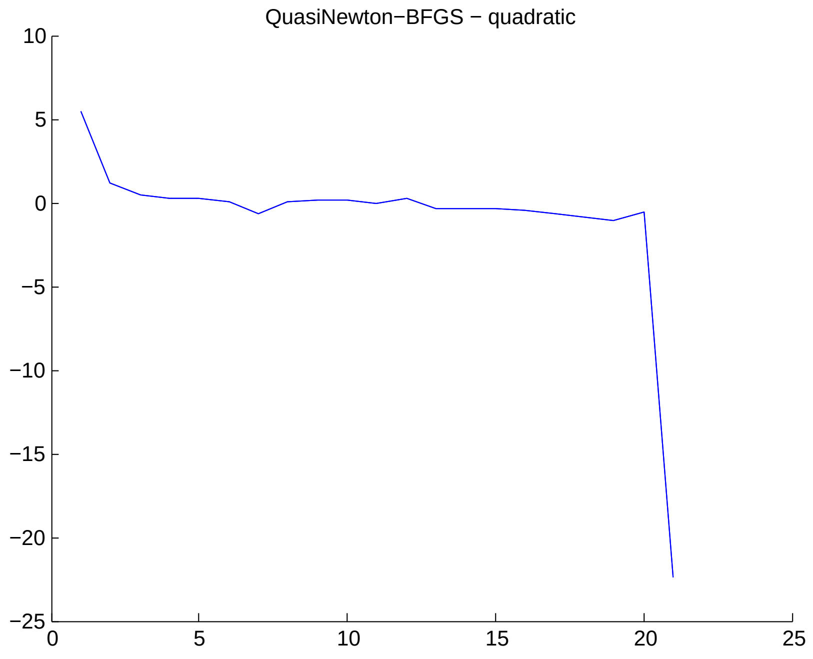

FIGURE 3. BFGS with exact line search, completes in 21 iterations.

TABLE 4. Experimental Results - Trigonometric Function, 40 dimensional, starting point 1/(10n), $\eta = 1e - 12$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	6	0.0144	0	
CG-FR	5	0.0492	3	
CG-PR	5	0.048	3	
BFGS	11	0.682	0	
L-BFGS-5	14	0.0896	0	
L-BFGS-7	12	0.089	0	
L-BFGS-9	13	0.117	0	
L-BFGS-11	13	0.129	0	

working well also. Both of the CG routines must restart very often due to the limited dimensionality of this problem.

2.4. **Rosenbrock Results.** The Rosenbrock function was tested with multiple dimensions up to 20 to test the performance of the optimization routines when the problem gets very large. The gradient descent algorithm had a lot of trouble with

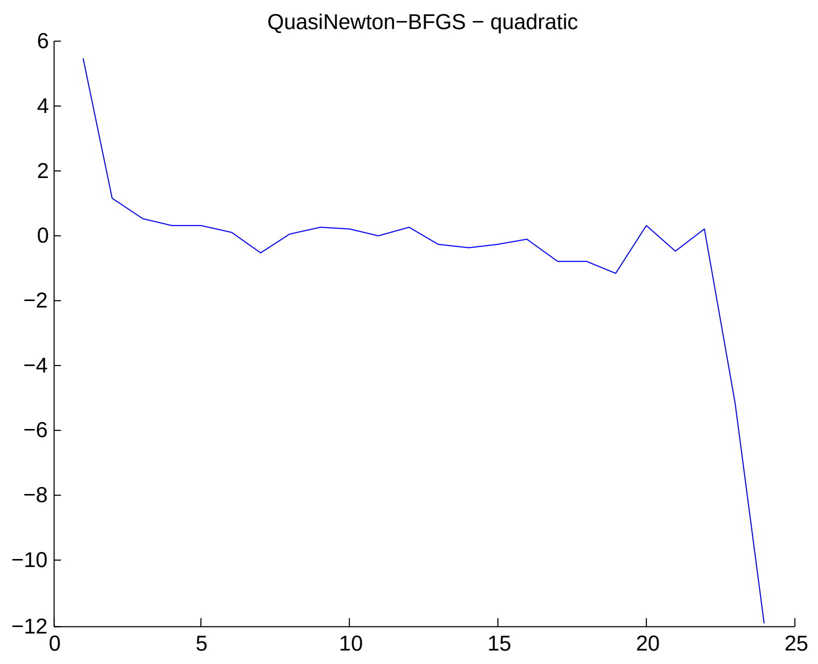

FIGURE 4. BFGS with inexact line search, completes in 24 iterations.

TABLE 5. Experimental Results - Wood Function, $\eta = 1e - 10$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	6707	1.64	0	
CG-FR	115	0.163	43	
CG-PR	139	0.172	55	
BFGS	62	0.165	0	
L-BFGS-5	111	0.202	0	
L-BFGS-7	141	0.232	0	
L-BFGS-9	152	0.271	0	
L-BFGS-11	170	0.312	0	

this problem and could only achieve $1e - 2$ gradient norm in a reasonable amount of time. The BFGS algorithm is seen to always be superior to other routines and L-BFGS with small memory is also very good. The CG routines restart approximately every n iterations where n is the dimensionality of the problem. Table 6 shows the results for the 6 dimensional problem, Table 7 for 8 dimensional, Table 8 for 10 dimensional and Table 9 for 20 dimensional.

Graphical results showing the logarithm of the gradient norm per iteration step for the Rosenbrock 20 function can be seen at Figures 5,6, 7, 8, 9, 10, 11, and 12.

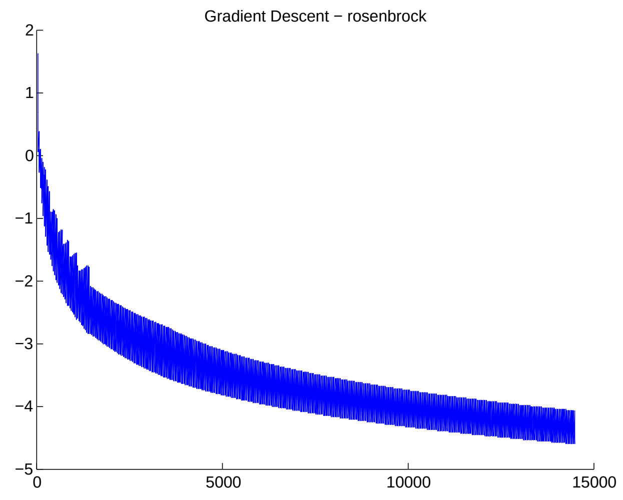

FIGURE 5. Gradient Descent

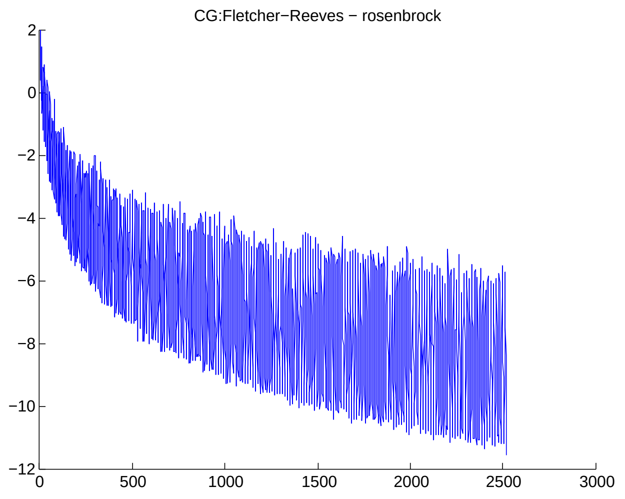

FIGURE 6. Conjugate Gradients- Fletcher Reeves

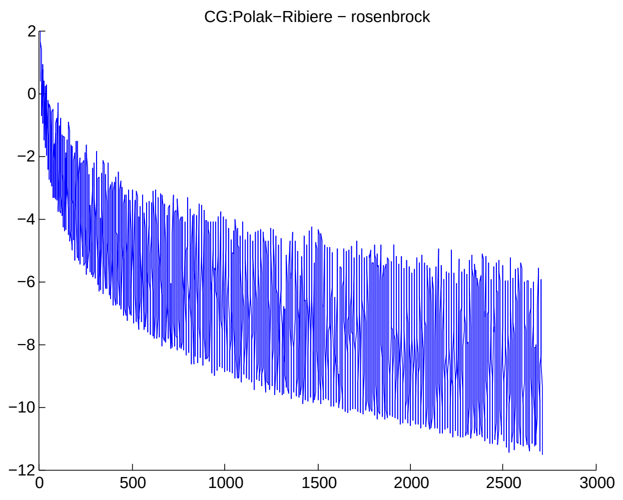

FIGURE 7. Conjugate Gradients- Polack Ribiere

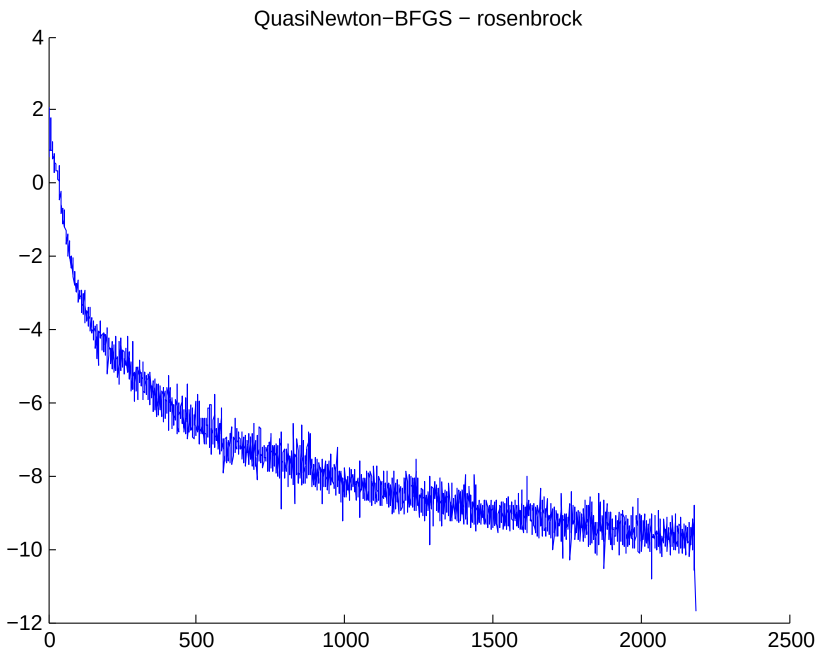

FIGURE 8. BGFS

Table 6. Experimental Results - Rosenbrock Function 6 dimensional, $\eta = 1e-10$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	> 14718	$> 4sec$	0	$\eta = 1e-2$
CG-FR	371	0.157	65	
CG-PR	387	0.231	60	
BFGS	116	0.172	0	
L-BFGS-5	266	0.287	0	
L-BFGS-7	327	0.343	0	
L-BFGS-9	372	0.413	0	
L-BFGS-11	449	0.561	0	

Table 7. Experimental Results - Rosenbrock Function 8 dimensional, $\eta = 1e-10$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	> 14508	> 3.86	0	$\eta = 1e-2$
CG-FR	1120	0.353	146	
CG-PR	1186	0.465	150	
BFGS	258	0.245	0	
L-BFGS-5	572	0.444	0	
L-BFGS-7	762	0.675	0	
L-BFGS-9	992	0.987	0	
L-BFGS-11	1062	1.24	0	

Table 8. Experimental Results - Rosenbrock Function 10 dimensional, $\eta = 1e-10$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	> 14508	> 3.875	0	$\eta = 1e-2$
CG-FR	3543	1.031	358	
CG-PR	3465	1.105	384	
BFGS	589	0.373	0	
L-BFGS-5	1286	0.922	0	
L-BFGS-7	1888	1.65	0	
L-BFGS-9	2250	2.458	0	
L-BFGS-11	3383	4.59	0	

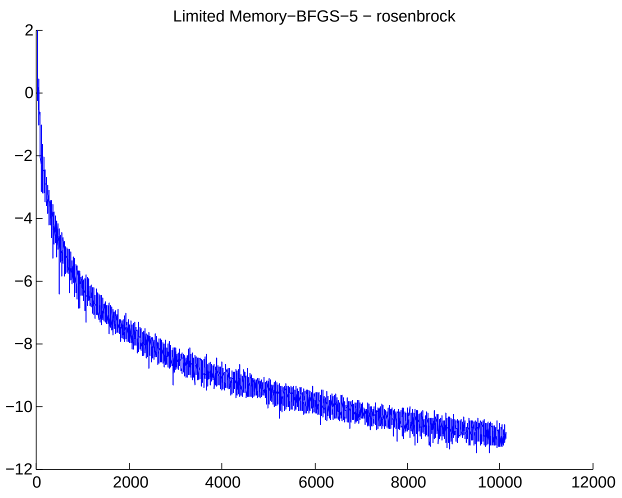

Figure 9. L-BFGS-5

3. Conclusion

The main thing that we learned in working on this project was that between almost any of the methods, there are tradeoffs. Here are several examples.

The Gradient Descent method was the easiest of the methods to implement and is intuitively easy to understand. However, as discussed frequently in class,

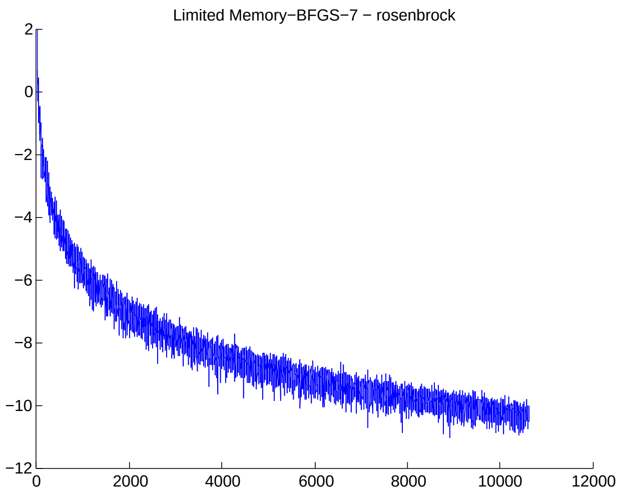

FIGURE 10. L-BFGS-7

the method suffers from severe performance issues when given an ill-conditioned problem. In fact, the Gradient Descent algorithm was usually too slow to terminate at all for the same convergence threshold as the more sophisticated optimization routines that we implemented. By comparison, on the 20 dimensional Rosenbrock

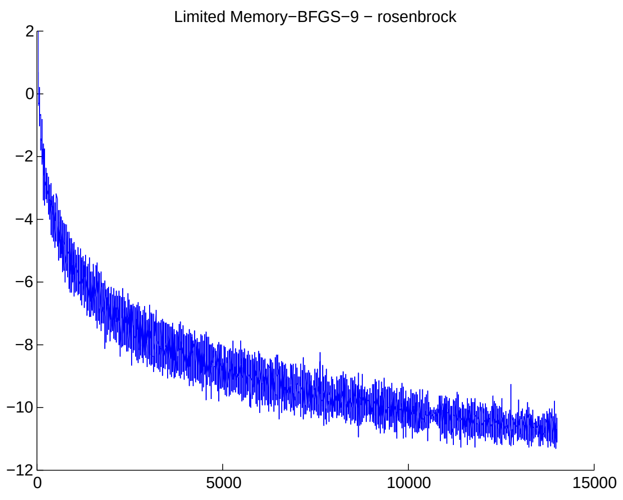

FIGURE 11. L-BFGS-9

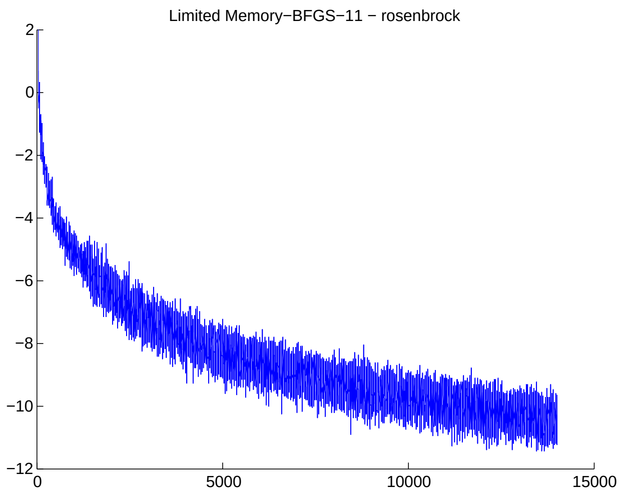

FIGURE 12. L-BFGS-11

function, the BFGS routine was about to achieve 8 more decimal points of accuracy than the GD with far less computation.

Similarly, before we implemented restarts in the CG methods, the Polak-Ribiere Conjugate Gradient method performed much better than the Fletcher-Reeves

Table 9. Experimental Results - Rosenbrock Function 20 dimensional, $\eta = 1e - 5$

Algorithm	Iterations	CPU-Time	Restarts	Notes
GD	> 14508	> 3.9	0	$\eta = 1e - 2$
CG-FR	2522	0.818	243	
CG-PR	2713	0.887	262	
BFGS	2184	1.18	0	
L-BFGS-5	10124	8.42	0	
L-BFGS-7	10625	12.8	0	
L-BFGS-9	14011	23.4	0	
L-BFGS-11	14030	32.6	0	

method which we expected. When we implemented restarts, both methods improved; however, the CGFR improved so much more that it seems to now be more effective than the CGPR on our test problems.

Finally, we found that the BFGS method took the fewest number of iterations to converge. However, it was not always the fastest - the limited memory BFGS method took more iterations to complete, but was sometimes faster than the BFGS method. Further, the performance of the limited BFGS method depended strongly upon the number of vectors to remember. Although we expected the number of iterations to decrease with increases to the history size, this was not the case for most of the functions we tested. The BFGS routine typically performs best with around 5 vectors except in the quadratic function case where up to 20 vectors would probably be beneficial. In fact, for the Wood and Rosenbrock functions the smallest history (m=5)took the least number of iterations for all memory sizes to complete. And for all of the functions, the smallest history (m=5) had the best performance in terms of CPU time. Thus, it seems like, at least at the scale that we were able to achieve, the cost of additional memory size had a stronger effect on the performance than did the benefit of having more vector history. This is due to the changing landscape of the optimization problem and the loss of utility from information of the past in hard nonlinear optimization problems.