Optimization Homework 2: Unconstrained Optimization

Mark C. Anderson

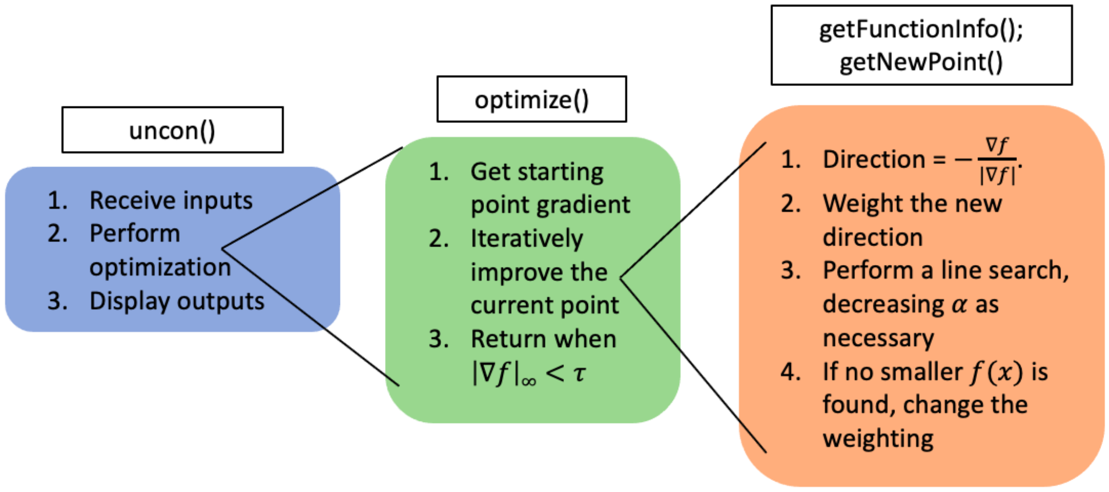
February 10, 2021

1 Introduction

For this project, I created an unconstrained gradient-based optimization function. The code that I wrote uses a conjugate gradient line search method. This was chosen because I started with a simple line search in the direction of steepest descent, got that working, and then started the conjugate gradient line search, with the plan to build up in complexity to more sophisticated methods. Due to time constraints, I decided to stay with this method after it started working and use it to generate all of the data for this report. The optimizer finds accurate solutions to three test problems, shown below in the Results section. The primary drawback to the optimizer that I created is the large number of function calls relative to fminunc, a built-in MATLAB optimizer. A number of potential ways to improve my code, as well as lessons learned, are included in the Conclusion section of this report.

2 Methods

The algorithm that I chose was a conjugate gradient line search method. The initial plan was to use a quasi-Newton method, but in the interest of getting good practice, I decided to first do a simple steepest-descent method first and build up to more superior methods. However, on Saturday I decided that the current iteration of the code was functional and that I ought to spend time writing it up rather than experimenting with another method. Here is an outline of my code:[1]



uncon()
1. Receive inputs
2. Perform optimization
3. Display outputs
optimize()
1. Get starting point gradient
2. Iteratively improve the current point
3. Return when $|\nabla f|_\infty < \tau$
getFunctionInfo(); getNewPoint()
1. Direction = $-\frac{\nabla f}{|\nabla f|}$.
2. Weight the new direction
3. Perform a line search, decreasing α as necessary
4. If no smaller $f(x)$ is found, change the weighting

Figure 1: An outline of the unconstrained optimizer that I wrote. There are some details missing from this diagram, but the overall flow is present. Above each colored box is another box containing the most relevant functions relating to those processes.

[1]The code can be found online at https://github.com/Classes-MCA/MEEN575-Optimization/tree/main/Homework%202/Code.

Some key features to point out include:

1. My function comes with the ability to input multiple different parameters for visualizing and creating GIF files of the convergence process. These are all listed as parameters using the input parser object, so if they are omitted then the function still works. This means that the function will still work with exactly the same signature as initially provided.

2. The new direction on each iteration is weighted against the previous iteration as

$$p_{weighted} = (1 - \omega)p_{old} + \omega p_{new} \tag{1}$$

   where $p_{weighted}$ is the direction in which to perform the line search, p_{old} is the direction from the previous iteration, p_{new} is the steepest descent direction from the current iteration, and ω is the weighting value starting at 0.5 and increased to 1.0 as needed to find a lower function value in the direction of $p_{weighted}$.

3. The value for α starts at 1.0 and is decreased by a factor of 2 each time that the line search fails to return a smaller function value. This is done up to 50 times, when machine precision on alpha is approximately reached. If a smaller function value is not found, then the weighting between the old and new directions is changed to give more weight to the steepest descent direction.

3 Results

3.1 Quadratic Function

The quadratic function was minimized by both methods, and the number of function calls for both methods given three starting points is shown in Table 1. We see here, as we will see in the other two examples, that uncon requires at least an order of magnitude more function calls than the built-in fminunc solver.

Figures 2 and 3 show the convergence for the case starting at (-4,10). The optimizer could definitely benefit from allowing larger values for α, which would decrease the total number of iterations required to descend toward the minimum.

Table 1: The required number of function calls to minimize the quadratic function for both uncon and fminunc.

	Function Calls	
x_0	**uncon**	**fminunc**
(-4,10)	251	21
(-5,5)	132	9
(3,7)	317	30

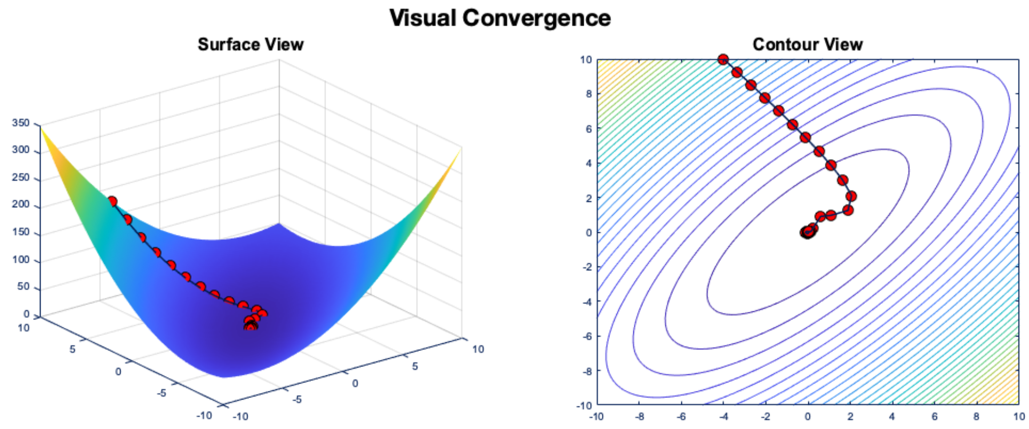


Figure 2: A visual representation of the convergence starting from (-4,10)

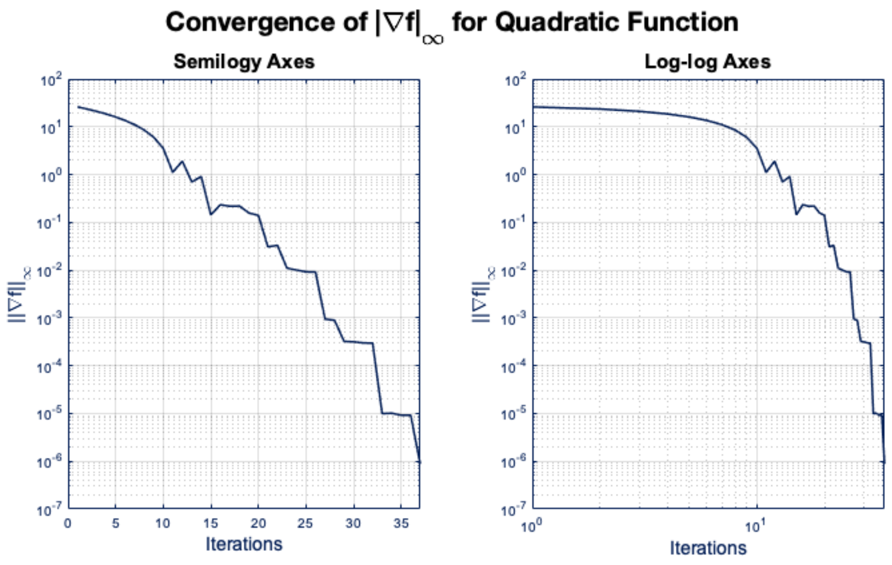


Figure 3: A graphical representation of the convergence starting from (-4,10) for the infinity norm of the gradient.

3.2 Rosenbrock Function

Similar to the quadratic function case, Table 2 shows that uncon requires at least an order of magnitude more function calls than fminunc. In particular, choosing a starting point near the valley such as (1.5,2) causes two orders of magnitude more function calls. This is likely due to the fact that the valley is so narrow and the optimizer can get caught going back and forth between the steep walls of the valley.

Figures 4 and 5 show the convergence for the starting point (-1,-1). We can see here an example of the optimizer getting ”stuck” in the valley as it tries to find the minimum.

Table 2: The required number of function calls to minimize the Rosenbrock function for both uncon and fminunc.

	Function Calls	
x_0	**uncon**	**fminunc**
(-1,-1)	498	90
(0,0)	474	81
(1.5,2)	1503	78

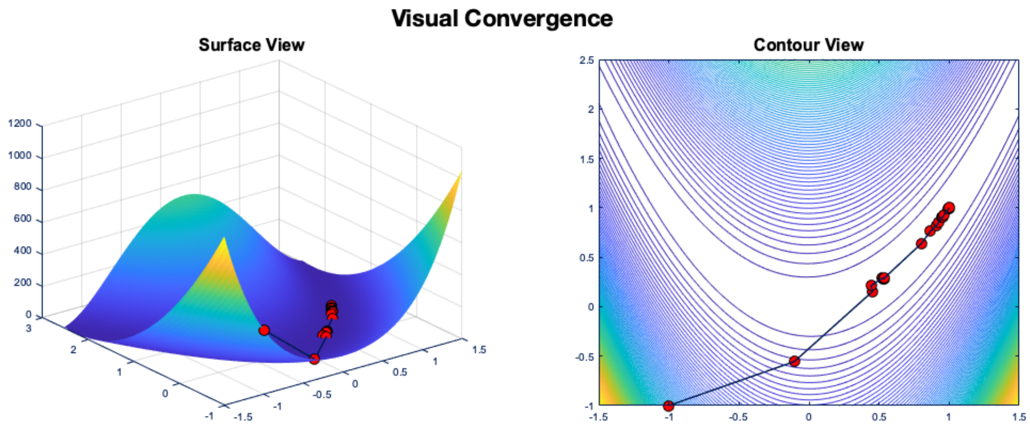


Figure 4: A visual representation of the convergence starting from (-4,10)

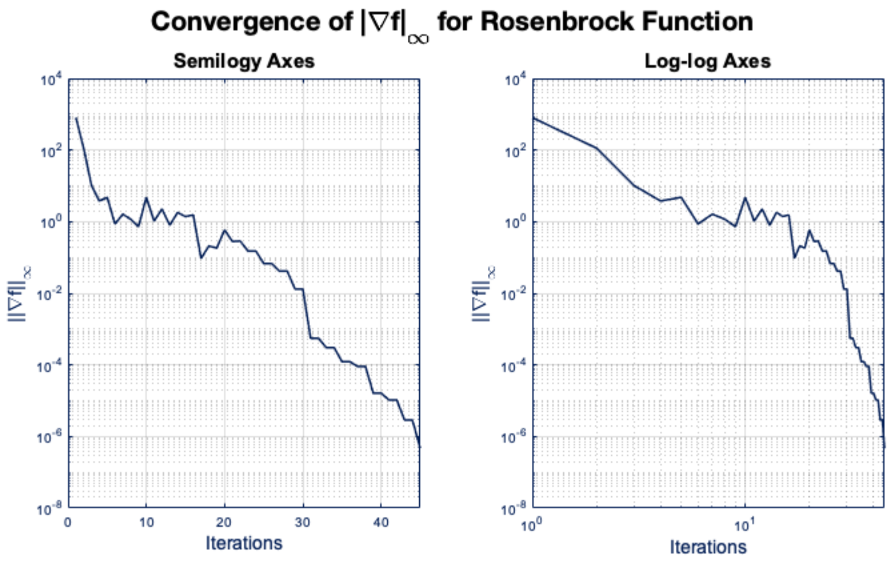


Figure 5: A graphical representation of the convergence starting from (-4,10) for the infinity norm of the gradient.

3.3 Brachistochrone Problem

The results for this problem are shown in Table 3. Here we see that for both starting conditions uncon requires two orders of magnitude more function calls than fminunc. However, as seen in Figure 6, both optimizers work well and ultimately find the same solution.

Looking at Figure 7, we see that the optimizer bounces around a lot in a staircase shape, hinting that we could likely improve this result a great deal by getting it to descend to lower "stairs" quicker.

Table 3: The required number of function calls to minimize the Brachistochrone problem for both uncon and fminunc

	Function Calls	
x_0	**uncon**	**fminunc**
Downward Linear Slope	248966	6018
$1/2\cos(6\pi x)$	254334	6195

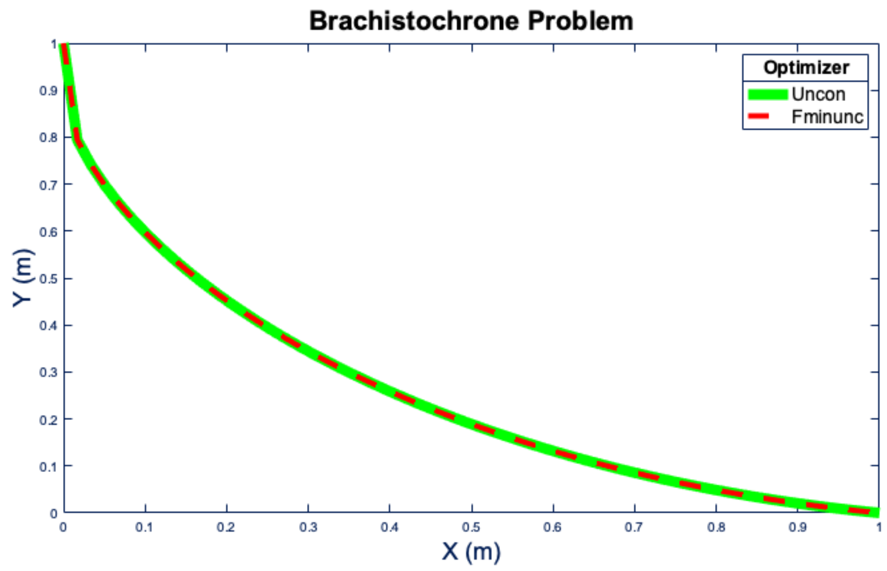


Figure 6: A visual representation of the converged values after starting from a linear downward slope.

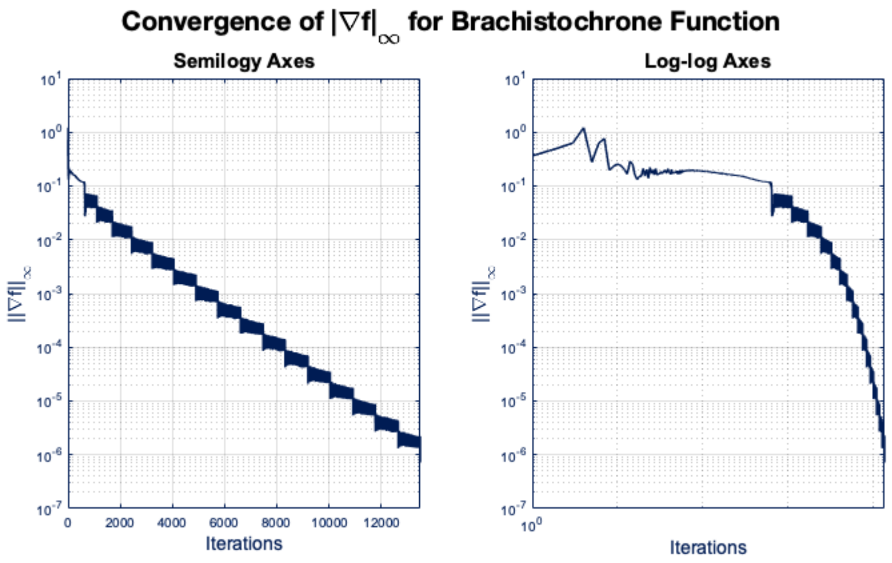


Figure 7: A graphical representation of the convergence starting from a linear downward slope for the infinity norm of the gradient.

4 Conclusion

The optimizer that I wrote does a good job of finding the mimina of the functions. It successfully and reliably finds all of the solutions from multiple different starting points. The primary drawback to my function is the massive number of function calls required to solve each problem compared to the number of function calls required by fminunc.

Main challenges faced:

1. I put considerable effort into visualizing the optimization. This helped in most ways, but caused me to lose some time due to incorrect plotting. The plots I created were rotated relative to what the optimizer was doing, which created a confusing result that appeared to show a disconnect between the function values and the optimization. I spent lots of time checking and double-checking my functions and gradients, particularly for the Rosenbrock function. As it turns out, these were fine, and the issue was in the visualization.
2. Trying to determine how best to organize my code was also a slight challenge. I did plan it out somewhat from the beginning so that it wasn't a total mess, but I did ultimately commit my work and then deleted most of the code and rewrote it with much better organization and modularity.
3. Since the value of α sometimes went to effectively zero without finding a lower function value, the optimizer would get stuck making tiny steps or not moving at all. This was resolved by including the variable weighting on the new direction vector. This means that if the value of α goes to zero then the new direction will be given more weight, and ultimately a lower function value must be found.

Some ideas on how my code can be improved are

1. Make α proportional to the gradient. This would automatically cause α to become small near the minimum and large on steep descents. This would not necessarily solve the issues though because a minimum may be found near steep slopes and there may still be lots of bouncing around.
2. Make the weighting value based on some criteria from the previous iteration(s). This would help to orient the search direction in the most useful directions. I'm not sure how to best do this, but it seems like something that would be beneficial.
3. Reduce the problem dimensionality, and then interpolate to higher dimensions. This would be particularly useful for the Brachistochrone problem, where we could solve it with maybe 4 design variables, then interpolate that solution and use it as an initial guess for 20 design variables, and so on until we reach our desired number of design variables.
4. Use a superior method like Newton, quasi-Newton, or other more advanced methods. The only reason that one of these methods was not used was because I reached the current code status on Saturday and decided to produce a nice write-up rather than pursue alternative methods. Because this project is under source control, it would be easy to improve the code without corrupting the current working version.

Some of the important lessons learned include

1. Creating functions that can easily be swapped out is an excellent idea. I have a function in my code called getFunctionInfo() that I could easily swap out for a function that does finite differencing to get the derivatives. I could also swap out the function getNewPoint() for a function that does a quasi-Newton solver or something else. I could even make a switch statement that changes the type of solver being used. Writing modular code is more difficult to plan out initially, but it really does pay out in the end.
2. Visualizations are instructive to create. The ability for me to watch the optimizer find the solution (after the plotting issue was resolved) enabled me to make more informed decisions about what the errors or shortcomings were in my code.

3. The simple methods like the one I used do indeed work, but they are slow compared to the better methods used in more advanced solvers.
4. I learned what the infinity norm is, and that was useful. I no longer have to solve for the max value in an array to determine whether something is converged.
5. Using Git is a must. I already use Git for almost every coding project I have, and the ability to have a fully-functional code in the main branch and still be able to experiment with different methods in another branch is great, and enables the peace of mind that your functioning code is safe.

References

[1] Martins, J. R. R. A., & Ning, A. (2021). Engineering Design Optimization. http://flowlab.groups.et.byu.net/mdobook.pdf