AA 222 Spring 2022 Project 1

Jacob Beardslee – jbeards3@stanford.edu

April 8, 2022

Part 1: Method Description

Methods Attempted

Several methods were attempted throughout this project to improve the results presented by the auto-tester. Rosenbrock's function, simple1, was the most difficult to produce good results for. I assume this is due to the magnitude of the gradients being so large around the global minimum as well as the low number of evaluations allowed. This can be seen in Part 2 on the contour plot.

Initially, I tried a simple gradient descent method with a fixed learning method based on the gradient descent method provided on pg. 71 in the course book, "Algorithms for Optimization" by Mykel J. Kochenderfer and Tim A. Wheeler. This method produced good results for Himmelblau and Powell, but I struggled to find a learning rate that would provide over 60% on the auto-grader for Rosenbrock.

Next, I tried the momentum method with a fixed learning rate and momentum decay based on the momentum method seen on pg. 75 in the course book. Similar to the last method, I struggled to find a learning rate/momentum decay that would provide good results for the Rosenbrock function. I was able to obtain 60-65% on the auto-grader but wanted to improve more.

Next, I tweaked my momentum method to align with the Nesterov-Momentum method (pg. 76 in course book). I also wanted to vary my learning rate through each iteration after seeing how my method progressed on the contour plot. If I used too low of a learning rate, each iteration made too small of jumps to get close to the minimum, but if my learning rate was too high, my solution would zig-zag heavily when it got closer to the minimum. I initially tried to implement a backtracking line search (pg 56 in course book), but this method required too many evaluations. Instead, I implemented a decaying learning rate (pg. 55 in course book). The decaying learning rate allowed me to take big steps initially to get

close to the minimum, then smaller steps to avoid zig-zagging as much. This produced the best results of any method, providing around 77% for Rosenbrock as well as near 100% for all other functions, including the secret functions.

Lastly, I tried the Adam method, (pg. 79 in course book) but the default hyperparameters in the course book didn't provide good results and I struggled to find good alternatives with trial and error. Since I was happy with my nesterov-momentum results, I stuck with that method for my final results.

Overview of Chosen Method

For reference, below is a screenshot of the method I used for all functions.

```
function optimizer_momentum_decay(f, g, x0, n; alpha = 0.1, beta = 0.9, gamma = 0.75)
    x_history = [x0]
    v = zeros(length(x0))

    while count(f, g) < n
        # Calculate gradient at next predicted design point based on current location/momentum
        g_x = g(x_history[end] + beta * v)

        # Calculate local descent
        d = -g_x / norm(g_x)

        # Calculate momentum
        v = beta * v + alpha * d

        # Calculate next design point
        x_next = x_history[end] + v

        # Add design point to history of design points
        push!(x_history, x_next)

        # Decay the learning rate each step
        alpha *= gamma
    end

    return x_history
end
```

A momentum method was chosen to help dampen the oscillations (zig-zag) being seen in the previous descent methods attempted. The algorithm begins by calculating the gradient at the next predicted design point based on the current location and momentum. Rather than use the gradient at the current design point, like in the regular momentum method, the Nesterov-Momentum method looks ahead to see what the gradient will be at the next predicted position and updates the momentum accordingly. The direction of steepest descent is then calculated by normalizing the gradient and then taking its negative. The updated momentum is then calculated using the current momentum and local descent, as well as the hyper-parameters alpha (learning rate) and beta (momentum decay). The momentum term

increases/decreases in each dimension based on the direction of steepest descent. Finally, the next design point is calculated by adding the calculated momentum to the previous design point. Lastly, the learning rate is reduced by a fixed decay factor. The algorithm repeats until the number of evaluations allowed is reached. Below are the function calls with their chosen hyper-parameters.

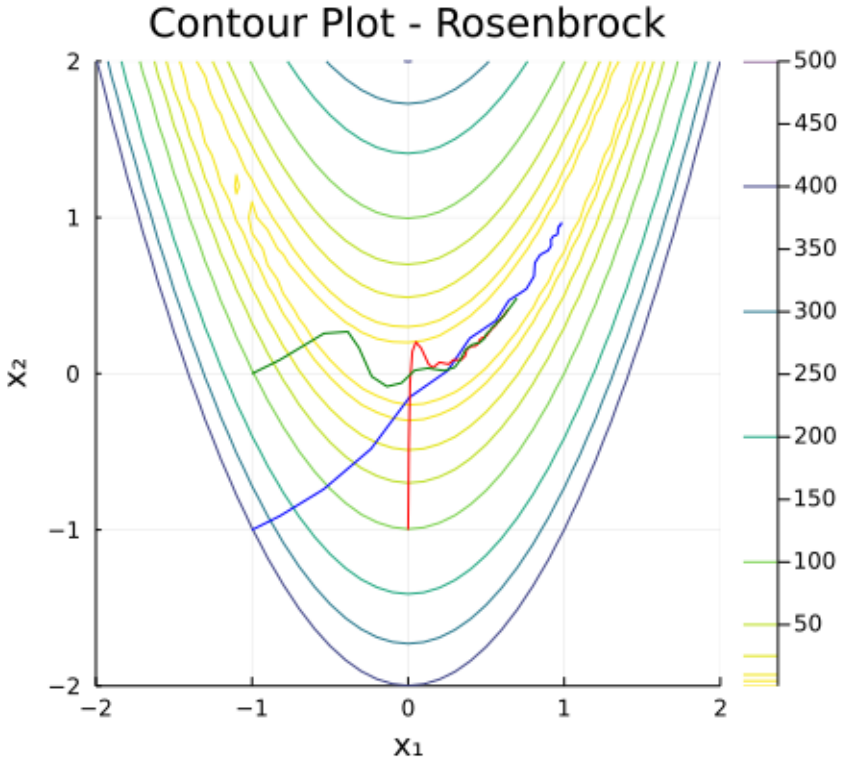
```
function optimize(f, g, x0, n, prob)
    if prob == "simple1"
        x_history = optimizer_momentum_decay(f, g, x0, n; alpha = 0.2, beta = 0.8, gamma = 0.85)
    elseif prob == "simple2"
        x_history = optimizer_momentum_decay(f, g, x0, n; alpha = 1, beta = 0.9, gamma = 0.85)
    elseif prob == "simple3"
        x_history = optimizer_momentum_decay(f, g, x0, n; alpha = 1, beta = 0.9, gamma = 0.85)
    elseif prob == "secret1"
        x_history = optimizer_momentum_decay(f, g, x0, n; alpha = 1, beta = 0.9, gamma = 0.85)
    elseif prob == "secret2"
        x_history = optimizer_momentum_decay(f, g, x0, n; alpha = 1, beta = 0.9, gamma = 0.85)
    end
    x_best = last(x_history)
    return x_best
end
```

Initially, an arbitrarily large step size (learning rate) of 1 was chosen to try to converge quicker. A momentum decay of 0.9 was chosen as it was the recommended value in two online articles I read (https://ruder.io/optimizing-gradient-descent/index.htmlotherrecentoptimizers / https://naokishibuya.medium.com/gradient-descent-optimizers-80d29f22deb5). The decay factor was chosen by trial and error, starting at a larger value then reducing until results were favorable. This was found to be at a decay factor of 0.85. These value worked well for all functions, including the secret functions, except for the Rosenbrock function. The step size and momentum decay were varied until results were favorable. This was found with a learning rate of 0.2 and a momentum decay of 0.8. I assume that a lower step size was needed for Rosenbrock due to how large the gradients are. Too big of a step size would cause the design point to jump around inefficiently.

Part 2: Rosenbrock Contour Plot

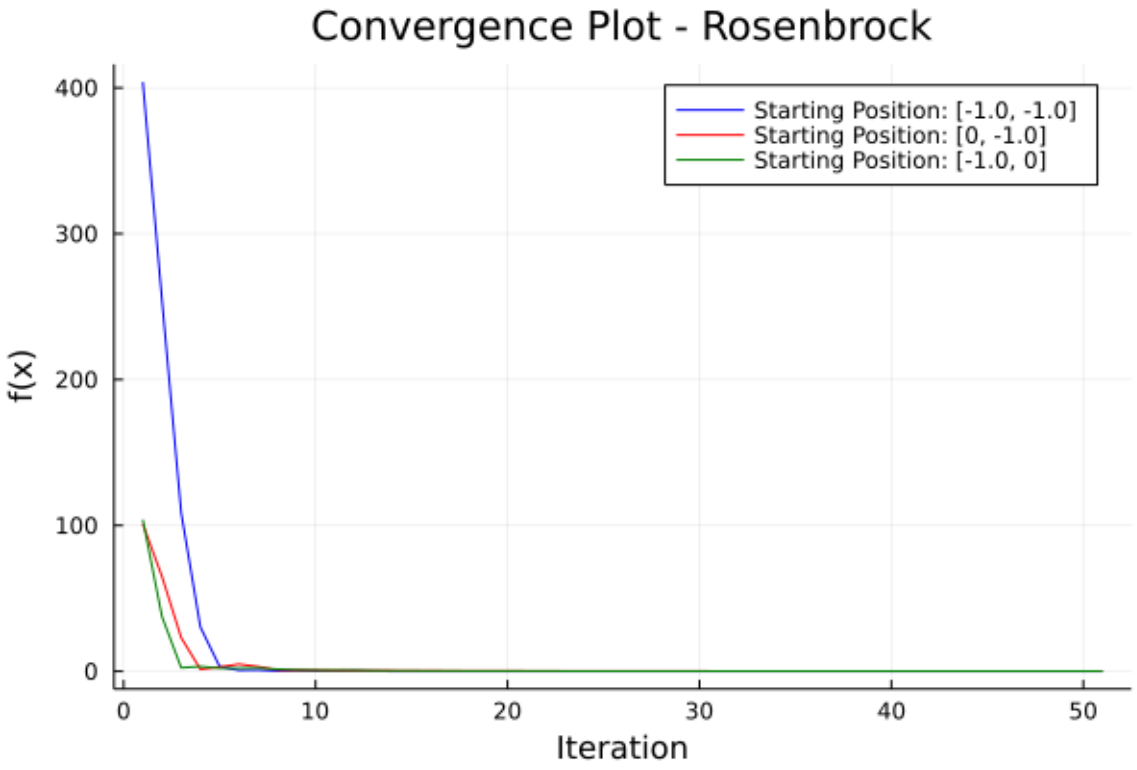
Note: The code from the plotting tutorial video provided was used to initially plot both the contour and convergence plots, then was tweaked to my preferences.

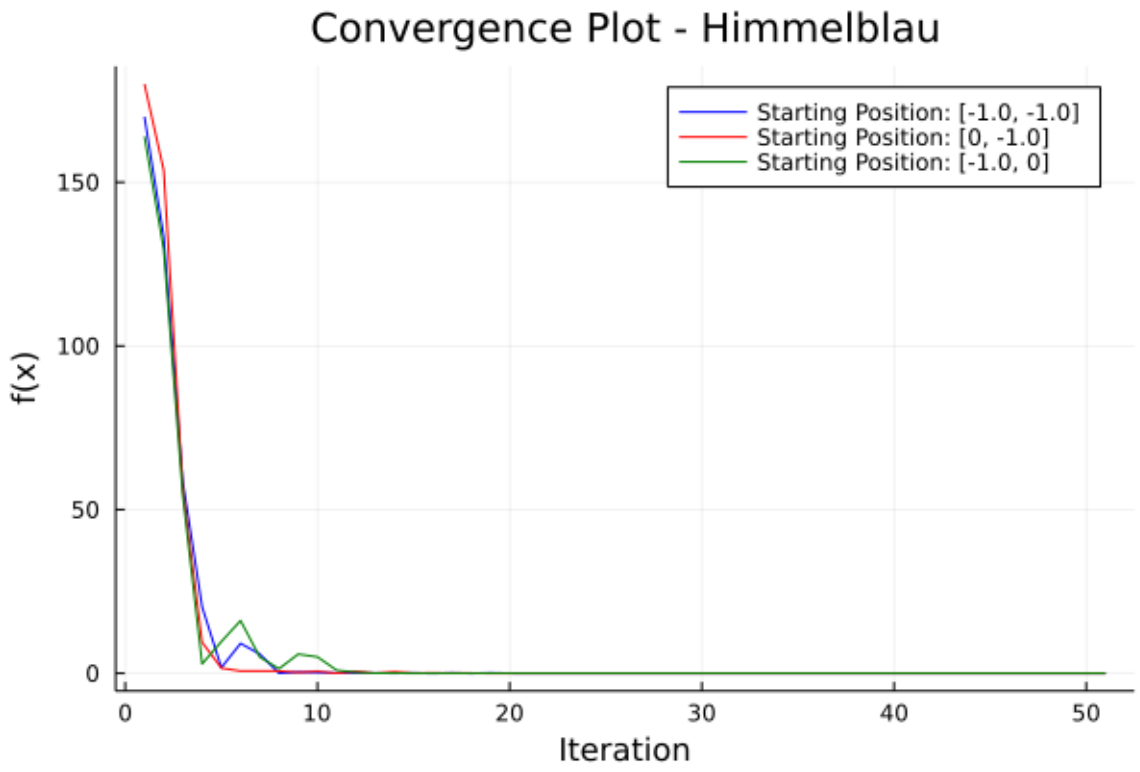
A contour plot for the Rosenbrock function can be seen below. The path taken by my algorithm for 3 different starting positions can be seen as well. I allowed the algorithm to iterate for 50 steps, regardless of evaluations used, to better view how the algorithm progressed over a longer period. Path 1 (blue) starts at (-1, -1), path 2 (red) starts at (0, -1), and path 3 starts at (-1, 0). As seen, all three paths take large steps toward the minimum initially, then smaller steps as they approach the absolute minimum. The algorithm does well to reduce oscillations.

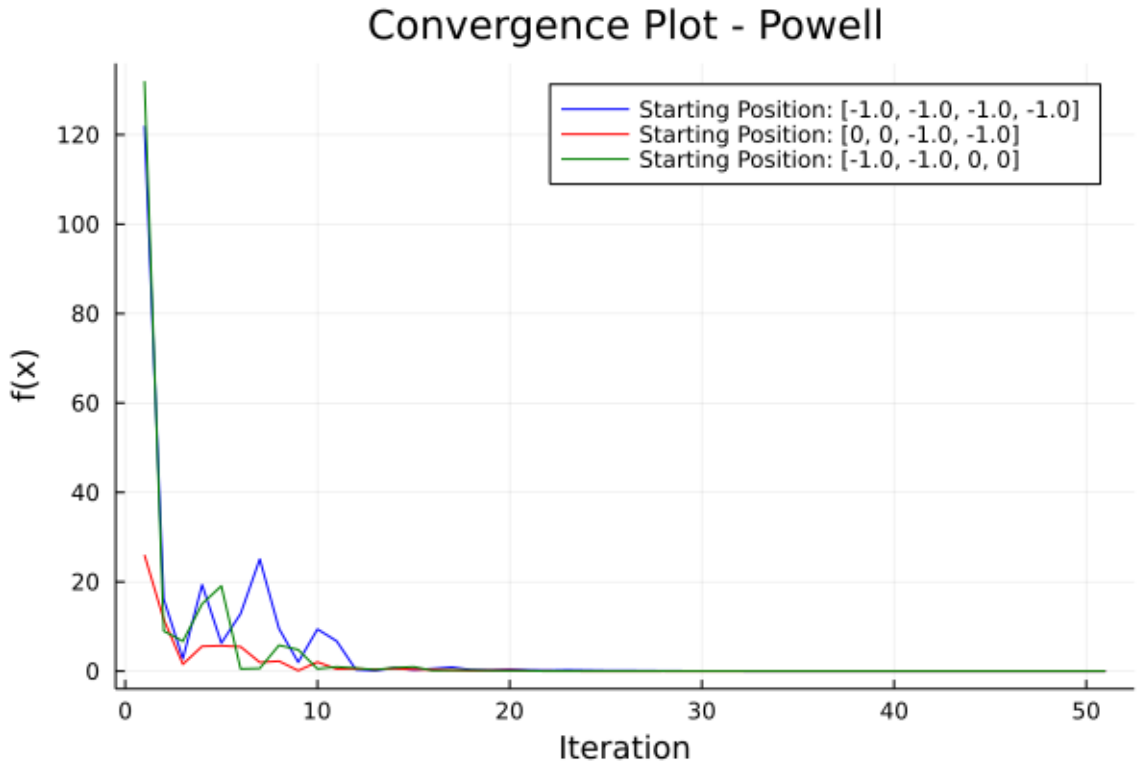


Part 3: Convergence Plots

Same as with the contour plot, the algorithm was allowed to run for 50 steps. All 3 function appear to convergence to the region of the global minimum rather quickly (within first 20 iterations), regardless of the starting position. Himmelblau and Powell experience much larger oscillations initially, which I assume is due to the larger starting step size of 1 comapared to 0.2 for the Rosenbrock function. I predict these oscillations would significantly decrease if I reduced the initial step size. 3 starting positions are used for each convergence plot and are listed in the plot's legend.




Convergence Plot - Himmelblau
Starting Position: [-1.0, -1.0]
Starting Position: [0, -1.0]
Starting Position: [-1.0, 0]
f(x)
150
100
50
0
0
10
20
30
40
50
Iteration


Convergence Plot - Powell
Starting Position: [-1.0, -1.0, -1.0, -1.0]
Starting Position: [0, 0, -1.0, -1.0]
Starting Position: [-1.0, -1.0, 0, 0]
f(x)
120
100
80
60
40
20
0
0
10
20
30
40
50
Iteration