Math 258A Challenge #2

Ian Gallagher

April 21, 2025

Problem 3: Computation with Newton's Method

Implement Newton's method both with fixed step size ($\lambda^\nu = 1$) and with Armijo step size ($\alpha, \beta = 0.8$) and apply it to the *Rosenbrock function*

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

Starting at $x^0 = (-1.2, 1)$, determine how many iterations it takes to obtain x^ν with $\|\nabla f(x^\nu)\|_2 \leq 10^{-7}$. Also apply the gradient descent method with Armijo step size to the same function and compare the results.
HINT: The Rosenbrock function has a unique minimizer at $(1, 1)$.

Fixed Newton

For the fixed step size, I have implemented the core Newton's method algorithm as follows:

```
def newton_fixed_step(x0, lambda_k, tolerance, max_iter=1e6):
    """Newton's method with a fixed step size."""
    x = x0
    trajectory = [x.copy()]

    iter = 0
    # Compute the initial function value and gradient
    _, grad, hessian = rosenbrock(x)
    while np.linalg.norm(grad) > tolerance and iter < max_iter:
        # Move in the computed Newton direction
        x -= lambda_k * np.linalg.solve(hessian, grad)
        # Update the trajectory
        trajectory.append(x.copy())

        # Compute the next values and iterate
        _, grad, hessian = rosenbrock(x)
        iter += 1
    return np.array(trajectory)
```

Running the above code with the Rosenbrock function, we can see how many iterations it takes to converge to the desired tolerance, as well as plot the trajectory of the optimization process. This is plotted in 1, where we see that there is some overshooting in the steps, but the method converges to the minimum in 6 iterations or so.

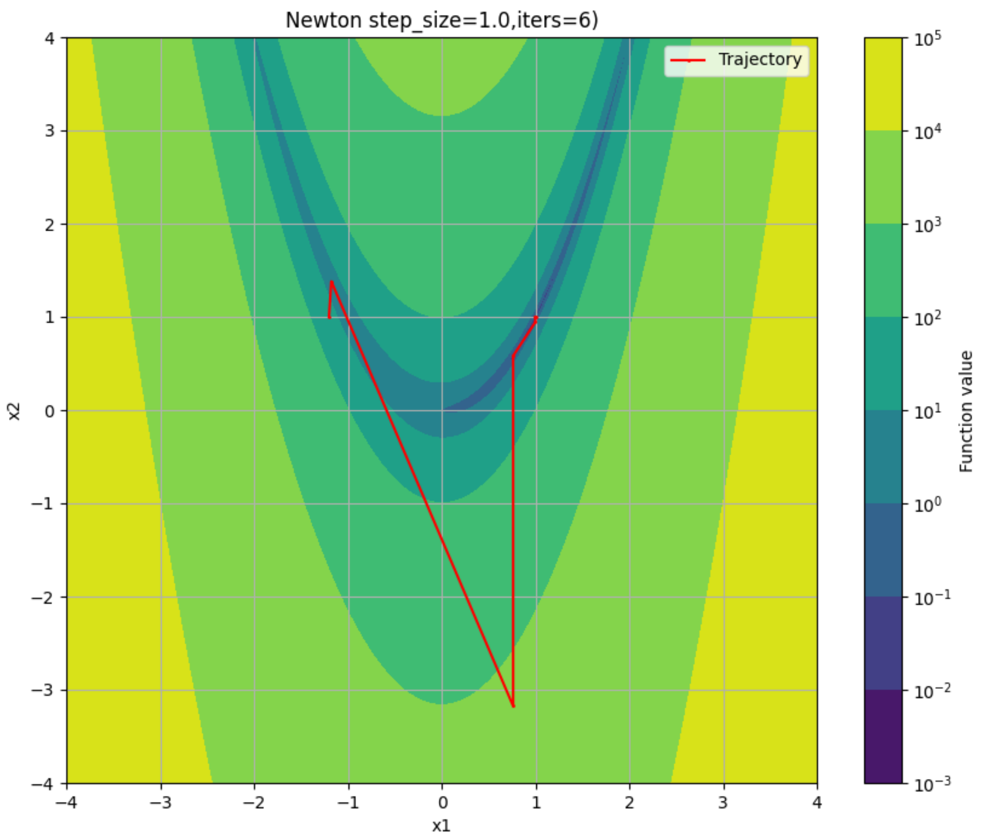

Figure 1: Newton's method with fixed step size converges in 6 iterations

Armijo Newton

Now, we switch to using the Armijo step size rule for Newton's method. The Armijo step size is computed by checking increasing integer values of s until the Armijo condition is satisfied.

```python
def compute_armijo_s(x, a, b):
    """Compute the Armijo step size."""
    s = 1

    f, grad, hessian = rosenbrock(x)
    p = np.linalg.solve(hessian, grad)
    f_next = rosenbrock(x - b**s * p)[0]

    while f_next - f >= - a * b**s * np.dot(grad, p):
        s += 1
        f_next = rosenbrock(x - b**s * p)[0]

    return s


def newton_armijo(x0, a, b, tolerance, max_iter=1e6):
    """Newton's method with a fixed step size."""
    x = x0
    trajectory = [x.copy()]
    iter = 0
    # Compute the initial function value and gradient
    _, grad, hessian = rosenbrock(x)
    while np.linalg.norm(grad) > tolerance and iter < max_iter:
        # Check the armijo condition
        s = compute_armijo_s(x, a, b)

        # Move in the direction of the negative gradient
        x -= b**s * np.linalg.solve(hessian, grad)
        # Update the trajectory
        trajectory.append(x.copy())

        # Compute the next values and iterate
        _, grad, hessian = rosenbrock(x)
        iter += 1
    return np.array(trajectory)
```

Running the code for the Rosenbrock function as before, the output is plotted in 2, where we now see that the trajectory is staying within the minimizing region of the function. The method converges in 74 iterations, so there is a tradeoff here between the number of iterations and the conditioning of the descent method.

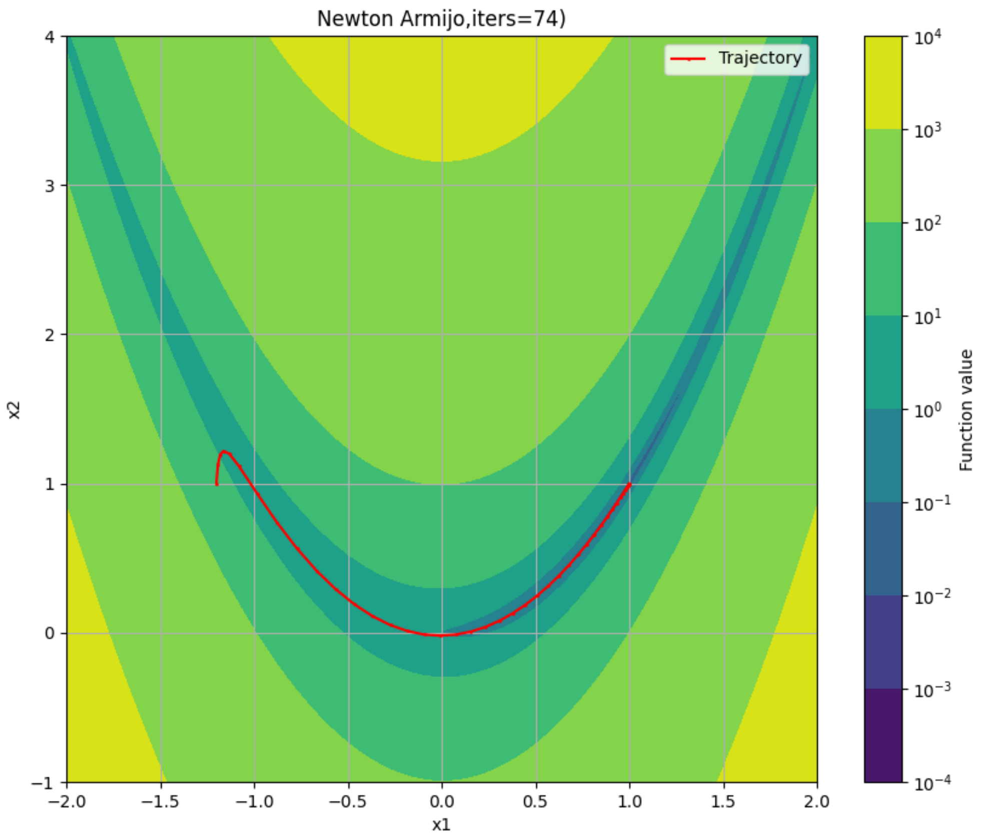

Figure 2: Newton's method with Armijo converges in 74 iterations

Armijo Gradient Descent

Finally, we have the case of gradient descent with Armijo step size. The code for this is a simplified verson of the Newton's method with Armijo step size, as we only need to use the gradient of the function and not the Hessian.

```
def compute_armijo_s(x, a, b):
    """Compute the Armijo step size."""
    s = 1

    f, grad, _ = rosenbrock(x)
    f_next = rosenbrock(x - b**s * grad)[0]

    while f_next - f >= - a * b**s * np.dot(grad, grad):
        s += 1
        f_next = rosenbrock(x - b**s * grad)[0]

    return s


def gradient_descent_armijo(x0, a, b, tolerance, max_iter=100000):
    """Perform gradient descent with a fixed step size."""
    x = x0
    trajectory = [x.copy()]

    iter = 0
    # Compute the initial function value and gradient
    _, grad, _ = rosenbrock(x)
    while np.linalg.norm(grad) > tolerance and iter < max_iter:
        # Check the armijo condition
        s = compute_armijo_s(x, a, b)

        # Move in the direction of the negative gradient
        x -= b**s * grad
        # Update the trajectory
        trajectory.append(x.copy())

        # Compute the next values and iterate
        _, grad, _ = rosenbrock(x)
        iter += 1
    return np.array(trajectory)
```

Running the code for the Rosenbrock function as before, the output is plotted in 3, where we see that the trajectory enters the minimizing region, but makes a large jump towards the absolute minimum of teh function. It then converges very slowly to the minimum over the remaining distances, taking 1131 iterations to converge. This makes sense as the gradient method is not accounting for the low curvature of the function when inside the minimizing basin, and so it is taking very small steps to converge to the minimum.

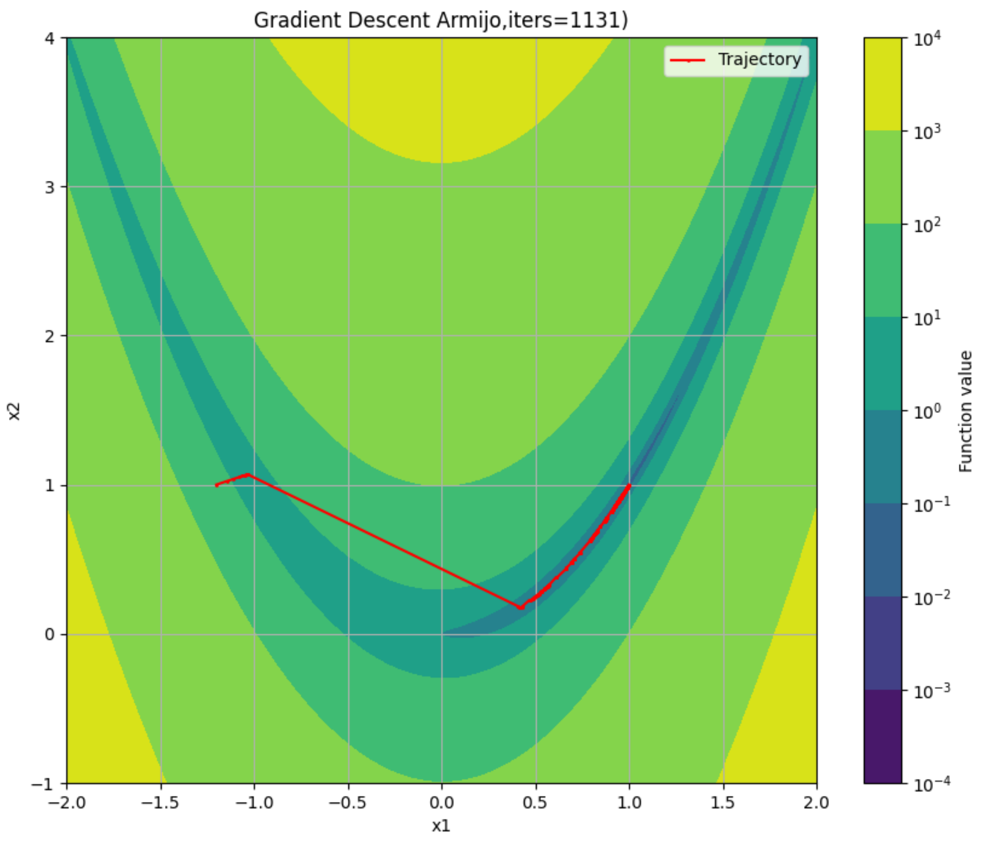

Figure 3: Gradient descent with Armijo converges in 1131 iterations