Linearity II
Gradient Descent Homework

David Abrahams

October 5, 2015

1 The contour plot

I generated a contour plot for $f(x, y) = \sin(x)\sin(x + 3y)$ with the following python functions:

```
def function(x, y):
    return sin(x) * sin(x + 3 * y)

def contour(func, x_min, x_max, y_min, y_max):

    x = linspace(x_min, x_max)
    y = linspace(y_min, y_max)
    X, Y = meshgrid(x, y)

    Z = func(X, Y)

    CS = plt.contour(X, Y, Z)

def grad_f(x, y):
    d_dx = sin(2*x + 3*y)
    d_dy = 3 * sin(x) * cos(x + 3 * y)
    return d_dx, d_dy
```

This produced the graph in Figure 1. There are a series of hills and valleys where the sin functions peak and valley. Running the following code to show the `quiver`:

```
def grad_f(x, y):
    d_dx = sin(2*x + 3*y)
    d_dy = 3 * sin(x) * cos(x + 3 * y)
    return d_dx, d_dy

def quiver(grad_func, x_min, x_max, y_min, y_max):
    x = linspace(x_min, x_max, num=20)
    y = linspace(y_min, y_max, num=20)
    X, Y = meshgrid(x, y)
```

```
    U, V = grad_func(X, Y)
    q = plt.quiver(X, Y, U, V)
```

produces Figure 2. The gradient points toward each of the hills, and away from the valleys.

To find a local maximum, I implemented a gradient descent algorithm. Starting from $(1, 1)$, it moves in the direction of the gradient.

```
def descent(grad_func, x_0, y_0, lambda_val, n_iterations):

    res = vstack((array([x_0, y_0]), zeros((n_iterations, 2))))

    for i in range(1, n_iterations + 1):
        prev_x, prev_y = res[i - 1, :]
        gradient_x, gradient_y = grad_func(prev_x, prev_y)
        next_x = lambda_val * gradient_x + prev_x
        next_y = lambda_val * gradient_y + prev_y
        res[i, :] = [next_x, next_y]

    x = res[:, 0]
    y = res[:, 1]
    q = plt.quiver(x[:-1], y[:-1], x[1:]-x[:-1], y[1:]-y[:-1],
                   scale_units='xy', angles='xy', scale=1)
    return res
```

Using the parameters `lambda_val=0.25, n_iterations=10` produces the graph in Figure 3. Using `lambda_val=0.1` produces Figure 4.

Using `lambda_val=0.25` fails to find the maximum because it bounces around it. `lambda_val=0.10` moves too slowly. A better solution is to dynamically change lambda using `fmin` in order to reach the highest possible point at each step iteration.

```
def accurate_descent(func, grad_func, x_0, y_0, n_iterations):

    res = vstack((array([x_0, y_0]), zeros((n_iterations, 2))))

    for i in range(1, n_iterations + 1):
        prev_x, prev_y = res[i - 1, :]
        gradient_x, gradient_y = grad_func(prev_x, prev_y)
        lambda_val = optimize_lambda(func, prev_x, prev_y, gradient_x,
                                     gradient_y)
        next_x = lambda_val * gradient_x + prev_x
        next_y = lambda_val * gradient_y + prev_y
        res[i, :] = [next_x, next_y]

    x = res[:, 0]
    y = res[:, 1]
    plt.quiver(x[:-1], y[:-1], x[1:]-x[:-1], y[1:]-y[:-1], scale_units='xy',
               angles='xy', scale=1)
```

```
    return res

def optimize_lambda(func, x_0, y_0, grad_x, grad_y):
    anon_func = lambda x: -f_x_lamba_f(func, x_0, y_0, grad_x, grad_y, x)
    return fmin(anon_func, 0)

def f_x_lamba_f(func, x_0, y_0, grad_x, grad_y, lambda_val):
    next_x = x_0 + grad_x * lambda_val
    next_y = y_0 + grad_y * lambda_val
    return func(next_x, next_y)
```

Doing 10 iterations produces the graph in Figure 5. This is clearly the most efficient algorithm, as it reaches the maximum relatively quickly.

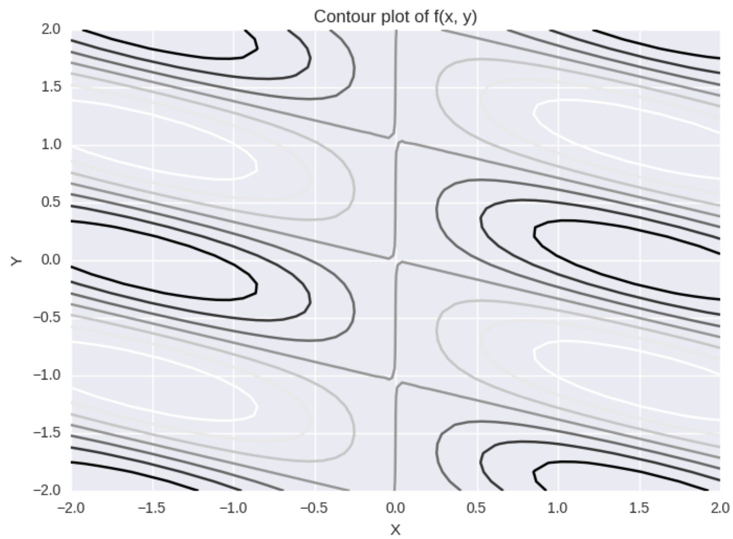

Figure 1: The contour plot

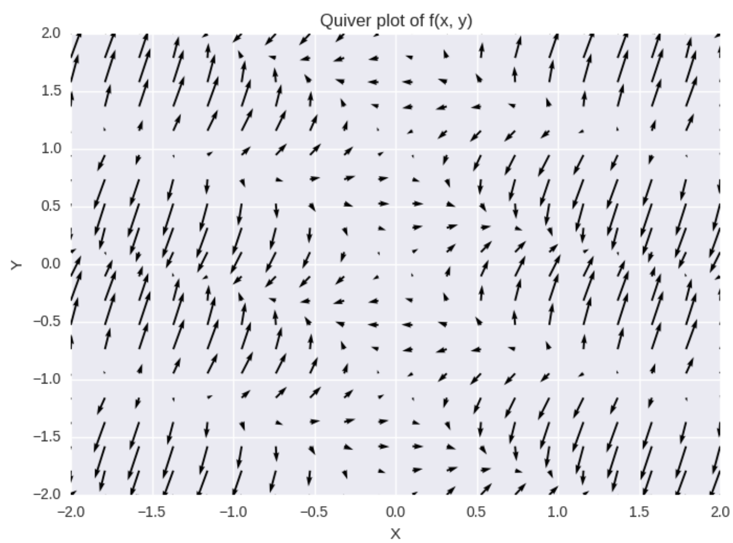

Figure 2: The quiver plot

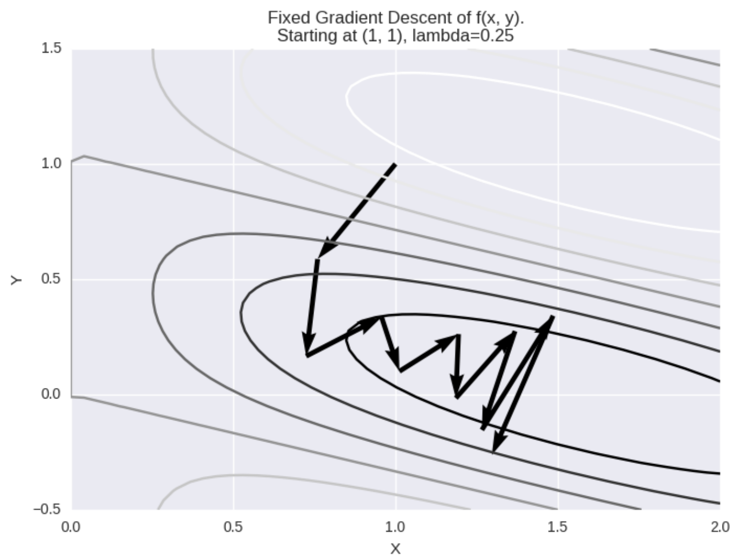

Figure 3: Gradient descent: `lambda_val=0.25, n_iterations=10`

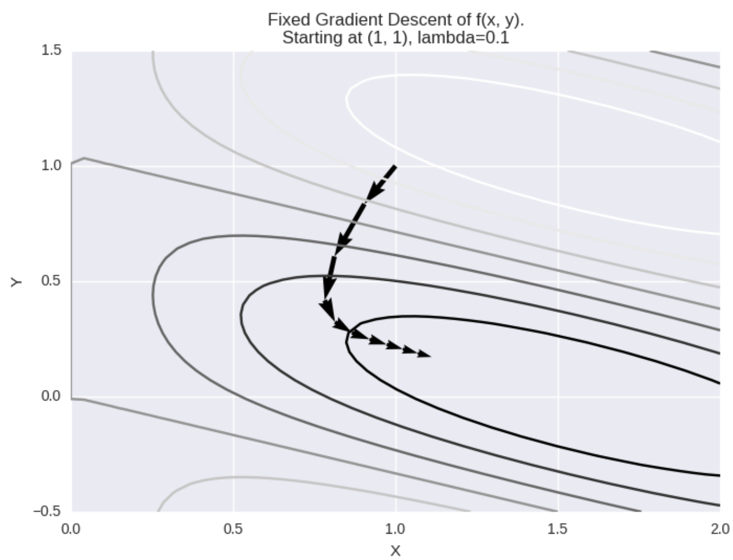

Figure 4: Gradient descent: `lambda_val=0.1, n_iterations=10`

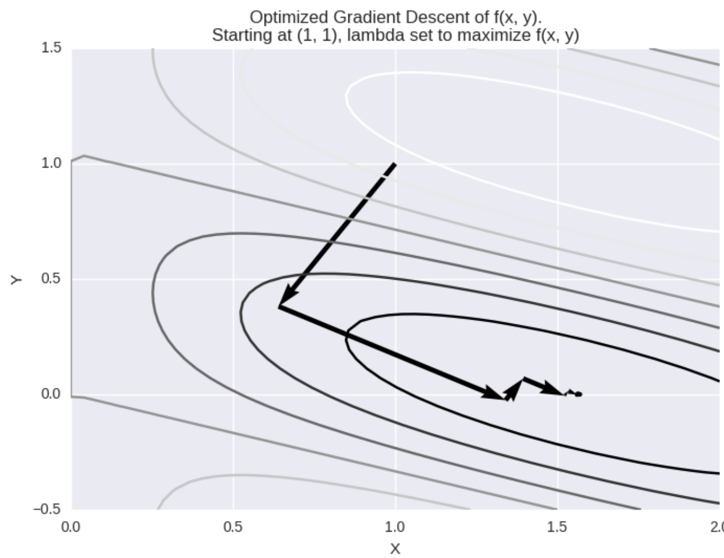

Figure 5: Gradient descent, with `lambda` being set to the value that gets to the highest point at each step.