CS168 Spring Assignment 8

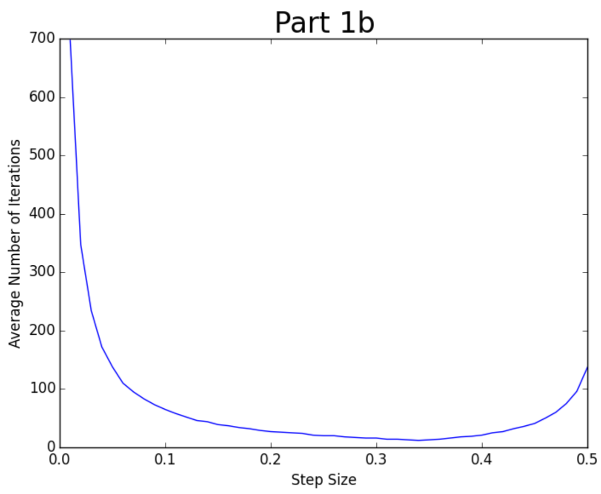
SUNet ID(s): johngold jsalloum
Name(s): John Gold Justin Salloum
Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Part 1: Gradient Descent

(a) We would typically run gradient descent until there is a less than epsilon change between two iterations. However, in this case we can go until the function is equal to 0 (or less than a tiny epsilon), because that is the absolute minimum.

(b) The best learning rate value is 0.35. Increasing the step size leads to more iterations because we will "overshoot" the optimal value.



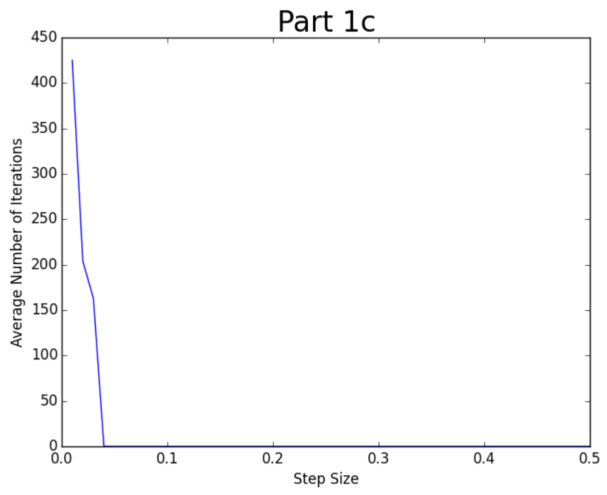
```
def gradient_descent(obj_fct, grad_fct, step_sizes):
    average_iters = []
    for step_size in step_sizes:
        total_iters = 0
        for trial in range(10):
            x = np.random.uniform(-100, 100, 10)
            prev_x = x
            while(obj_fct(x) > 0.0000001):
                new_x = np.zeros_like(x)
                for i in range(len(x)):
                    new_x[i] = x[i] - step_size*grad_fct(x[i], i)
                prev_x = x
                x = new_x
                total_iters += 1

        average_iters.append(total_iters/10)
    return average_iters

def gradient_b(xi, i):
    return (1+i/10.0)*2*xi

def obj_fct_b(x):
    total = 0
    for i in range(len(x)):
        xi = x[i]
        total += (1+i/10.0)*(xi*xi)
    return total
```

(c) We run into a lot of overflow errors, once we hit .04 as the step size, at least one of the numbers in our vector becomes NaN/infinity.

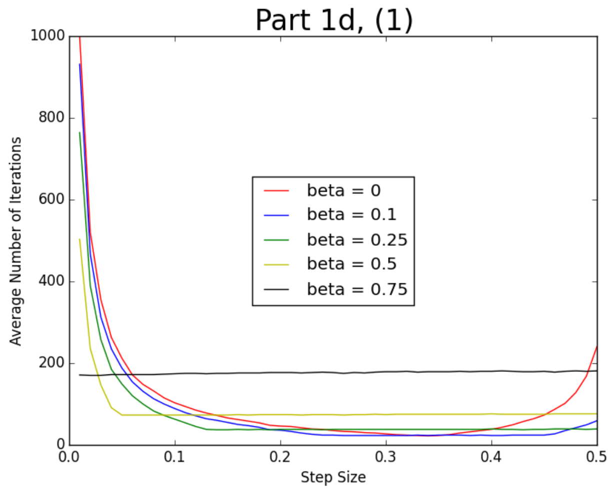


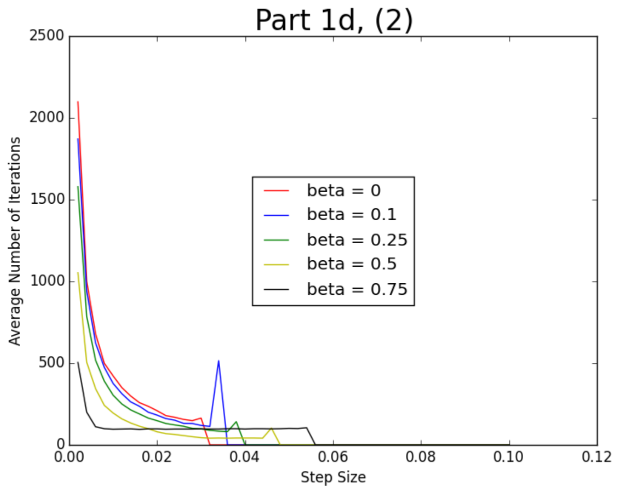
```
def gradient_descent(obj_fct, grad_fct, step_sizes):
    average_iters = []
    for step_size in step_sizes:
        became_nan = False
        total_iters = 0
        for trial in range(10):
            x = np.random.uniform(-100, 100, 10)
            prev_x = x
            while(obj_fct(x) > 0.0000001):
                new_x = np.zeros_like(x)
                for i in range(len(x)):
                    new_x[i] = x[i] - step_size*grad_fct(x[i], i)
                    if(math.isnan(new_x[i])):
                        became_nan = True
                prev_x = x
                x = new_x
                total_iters += 1
        if(became_nan):
            average_iters.append(0)
        else:
            average_iters.append(total_iters/10)
    return average_iters

def gradient_c(xi, i):
    return np.power(2, (i+1)/2.0)*2*xi

def obj_fct_c(x):
    total = 0
    for i in range(len(x)):
        xi = x[i]
        total += np.power(2,(i+1)/2.0)*(xi*xi)
    return total
```

(d) Accelerated gradient descent only slightly reduces the number of iterations in the (1) for small values of beta. For (2), accelerated gradient descent appears to help across the board, before we run into the same overflow errors from earlier.





```
def gradient_descent(obj_fct, grad_fct, step_sizes, beta):
  average_iters = []
  for step_size in step_sizes:
      total_iters = 0
      for trial in range(10):
          x = np.random.uniform(-100, 100, 10)
          prev_x = x
          while(obj_fct(x) > 0.0000001):
              new_x = np.zeros_like(x)
              for i in range(len(x)):
                  new_x[i] = x[i] - step_size*grad_fct(x[i], i) +
                                        beta*(x[i] - prev_x[i])
              prev_x = x
```

```
                x = new_x
                total_iters += 1

        average_iters.append(total_iters/10)
    return average_iters
```

Part 2: QWOP

(a) No deliverables

(b) For the initial QWOP Code: https://www.youtube.com/watch?v=RVYaEZur4xY — [1.4008089863986921, -0.13917877964150951, -0.61061823222546741, -2.0220931657631049, -3.7904971731588049, -0.19534058720704037, -0.91822024478712161, 1.7618198842003558, -1.2496528368408979, -0.039952487207143464, -0.64352792621658517, 0.51021214891451527, -0.83558773989385104, -0.6210923085937603, 0.39821825451552662, 1.1874673963952003, 1.6402647448458372, 1.0195654697854437, 0.48308390551910402, -0.37100324827774905, -1.2605103745759278, -0.34551677380619628, 0.18599606571305721, 0.8416804776569633, -0.17472486402664156, -0.65691610213418272, -1.59637040170116, -1.9805807040803682, 0.77376295509301629, -0.57840861794468645, 0.56032586871990764, 3.8933162934484979, -0.72436185472593295, -0.094577888301985935, 3.1341710933172262, -4.6321775959889679, -0.21116079672889077, 2.5438843618794613, -3.6239892399710318, 0.39395034954146702]

For the second QWOP Code: https://www.youtube.com/watch?v=pXmQglH9NpU [0.91085, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0]

```
def part2():
    plan = np.random.uniform(-1,1, 40)
    split = int(sys.argv[2])

    plan[:split] = start_plan[:split]

    initial_dist = sim(plan)
    cur_dist = initial_dist
    print "Initial distance = ", initial_dist

    prev_plan = copy.deepcopy(plan) + .001
    step_size = .1
    beta = .1
    num_iterations = 1
    best_plan_yet = copy.deepcopy(plan)
    best_dist = initial_dist


    while(np.linalg.norm(prev_plan - plan) > 1e-9):
        new_plan = copy.deepcopy(plan)
        cur_dist = sim(plan)
        for i in range(split,len(plan)):
            #In this case we want gradient ascent, to maximize our objective functi
            new_plan[i] = plan[i] + step_size*sim_analytical_gradient(plan, i, cur_
        new_dist = sim(new_plan)

        prev_plan = plan
        plan = new_plan
        print "new dist = ", new_dist

        num_iterations += 1

        if(new_dist > best_dist):
            best_dist = new_dist
            best_plan_yet = new_plan
            print "new best distance ", best_dist
        if(num_iterations \% 25 == 0):
            print list(best_plan_yet)


The following is from Matlab for particle swarm:

function [rating] = myqwop(plan,doanimation)
rating = -qwop(plan, 0);
end

lb = -1;
ub = 1;
options = optimoptions('particleswarm','ObjectiveLimit',-10);
[x, fval, exitFlag, output] = particleswarm(@myqwop, 40, lb, ub, options);
```

(c) Discussion: We tried a couple of different "hacks" to achieve these numbers, most notably the divide and conquer method. After a couple tries we noticed that the beginning of the vector corresponded to the beginning of the running man. So once we achieved a score of 3, we set the first 10 entires to always equal the movements to get to distance 3. We continued until we reached a score of 5, then 7, updating entires so they stopped changing. Other than that, we just tried different step size values, a little bit of manual annealing as we got towards higher numbers, and different values for beta (momentum). Another method we tried was a very naive MCMC approach that kept on randomly modifying one of the values of ”plan” and comparing the result of sim with the new plan vs. the old one, keeping track of the best plan and best score. This achieved average scores between 6 and 7. Our final best approach was using the particleswarm function in Matlab with size 40, lower bound of -1 and upper bound of 1. After several trials the best result we got was a score of 8.4.