ME596 Homework 6/7

Erin Schmidt

Problem Statement

We must construct an objective function for the following problem, handling constraints using the exterior penalty function method.

$$\min \; f(x) = (x_1 - 100)^2 + (x_2 - 50)^2$$

$$\text{Subject to} \quad g(x) = 170 - x_1 - x_2 \leq 0$$

Steepest Descent

In [11]:
```
#Steepest descent algorithm
#-Erin Schmidt

import numpy as np
import math as m

class eq_int:
    def func(alpha, x, norm_del_f, rp, count, n=0): #the objective function
        count += 1
        x[0] += alpha*norm_del_f[0]
        x[1] += alpha*norm_del_f[1]
        f = st_dec.func(x, rp, n)[0]
        return f, count

    def mini(au, al, x, norm_del_f, rp, count): #evaluates f at the minimum (or optimum) stati
        alpha = (au + al)*0.5
        (f, count) = eq_int.func(alpha, x, norm_del_f, rp, count)
        return f, alpha, count

    def search(al, x, norm_del_f, rp, delta=0.01, epsilon=1E-4, count=0):
        (f, count) = eq_int.func(al, x, norm_del_f, rp, count)
        fl = f #function value at lower bound

        while True:
            aa = delta
            (f, count) = eq_int.func(aa, x, norm_del_f, rp, count)
            fa = f
            if fa > fl:
                delta = delta * 0.1
            else:
                break

        while True:
            au = aa + delta
```

```
            (f, count) = eq_int.func(au, x, norm_del_f, rp, count)
            fu = f
            if fa > fu:
                al = aa
                aa = au
                fl = fa
                fa = fu
            else:
                break

        while True:
            if (au - al) > epsilon: #compares interval size to convergence criteria
                delta = delta * 0.1
                aa = al #intermediate alpha
                fa = fl #intermediate alpha function value
                while True:
                    au = aa + delta
                    (f, count) = eq_int.func(au, x, norm_del_f, rp, count)
                    fu = f
                    if fa > fu:
                        al = aa
                        aa = au
                        fl = fa
                        fa = fu
                        continue
                    else:
                        break
                continue
            else:
                (f, alpha, count) = eq_int.mini(au, al, x, norm_del_f, rp, count)
                return f, alpha, count

class st_dec:
    def func(x, rp, n=0):
        g = max([0, 170 - x[0] -x[1]])
        f = (x[0] - 100)**2 + (x[1] - 50)**2 + rp*g**2 #pseudo-objective value at x
        del_f = [2*(x[0] - 100) + rp*(-2*g), 2*(x[1] -50) + rp*(-2*g)] #gradient value at x
        del_f = np.array(del_f)
        f_val = (x[0] - 100)**2 + (x[1] - 50)**2 #actual function value
        n += 1
        return f, del_f, n, f_val

    def steepest(x, rp):
        n = 0 #iteration counter
        alpha = .01 #initial step size
        (f, del_f, n, f_val) = st_dec.func(x, rp, n)
        while True:
            x_old = x
            alpha_old = alpha
            norm_del_f = -del_f/m.sqrt(del_f[0]**2+del_f[1]**2) #normalize grad vector
            alpha = eq_int.search(alpha, x, norm_del_f, rp)[1]
            x[0] += alpha*norm_del_f[0] #next step x-values
            x[1] += alpha*norm_del_f[1]
            (f, del_f, n, f_val) = st_dec.func(x, rp, n)
```

```
            # Convergence criteria
            #if abs((alpha - alpha_old)/alpha) < 1E-12: #convergence criteria
             #   return f, n, x, alpha, f_val

            #a=np.array(x).reshape((2,1))
            #b=np.array(x_old).reshape((2,1))
            #if np.linalg.norm(b - a) < 1E-6:
             #   return f, n, x, alpha, f_val

            if n > 100000:
                return f, n, x, alpha, f_val


#starting design
x = [0,0]
rp = 100
(f, n, x, alpha, f_val) = st_dec.steepest(x, rp)
print('iterations = ', n)
print('x* = ', x)
print('f(x*) = ', f_val)
```

```
iterations =  100001
x* =  [109.60332810927328, 59.603328109273427]
f(x*) =  184.447821549
```

Discussion

For the sake of comparison I experimented with an alternate approach, without using an equal interval search to optimize the step-size:

In [13]:
```
# Another, simpler approach, without dynamic step sizes
import numpy as np

# Initial Guess
x_old = np.array([0, 0]).reshape((2,1))
x_new = np.array([10,100]).reshape((2,1))

# Parameters
gamma = .01 # Step size
epsilon = 1E-3 # Convergence parameter

#Objective function
def func(x, rp=200):
    g = max([0, 170 - x[0] - x[1]])
    f = (x[0] - 100)**2 + (x[1] - 50)**2 + rp*g**2 #pseudo-objective value at x
    del_f = [2*(x[0] - 100) + rp*(-2*g), 2*(x[1] -50) + rp*(-2*g)] #gradient value at x
    del_f = np.array(del_f).reshape((2,1))
    f_val = (x[0] - 100)**2 + (x[1] - 50)**2 #actual function value
    return f, del_f, f_val, g

# Steepest Descent
i = 0
while np.linalg.norm(x_old - x_new) > epsilon:
    x_old = x_new
```

```
        x_new = x_old - gamma * func(x_old)[1]
        i = i + 1

    print('iterations = ', i)
    print('x* = ', x_new.T[0])
    print('f(x*) = ', func(x_new)[2][0])
```

```
iterations =  16318
x* =  [ 109.9747016   59.9747016]
f(x*) =  198.98934383
```

This version of the steepest descent scheme runs much faster than the one using the dynamic step-size optimization (perhaps 2 orders of magnitude less physical time for the same number of iterations). Various values for the initial guess seems to find a similar optimum point, however the number of iterations required to find it can vary substantially. This applies also to the step size and the convergence parameter. Making the converge parameter smaller, will increase the number of digits of precision between the returned values of x_0 and the 'true' optima. The value of the penalty function coefficient does have a strong effect on the final optimum x^*. Too large values of `rp` can cause the algorithm to fail to converge. Too small values of `rp` and the penalty function fails to be optimized.

Regardless both versions of the steepest descent algorithm come close to the 'canonical' value of $f(x^*) = 200$ at $x^* = [110, 60]$.