Tarea 1

Felipe González Casabianca

Octubre 27 del 2016

Desarrollo

1. Sea para todo $k \in \mathbb{N}$, B_k una matriz definida positiva y **simétrica**, tal que:

$$\|B_k\|\|B_k^{-1}\| < M$$

Veamos que:

$$cos(\theta_k) > \frac{1}{M}$$

Para ver esto, veamos primero que si B es una matriz no singular y x un vector de las dimensiones apropiadas, entonces se tiene que:

$$\|Bx\| \geq \frac{\|x\|}{\|B^{-1}\|} \tag{1}$$

Veamos:

Recuerde que B se puede ver como una transformación lineal, así que su norma es equivalente al mínimo $\alpha \in \mathbb{R}^+$ tal que:

$$\|Bx\| \leq \alpha\|x\|$$

Por lo tanto tenemos que:

$$\|x\| = \|B^{-1}Bx\| \leq \|B^{-1}\|\,\|Bx\|$$

así que:

$$\frac{\|x\|}{\|B^{-1}\|} \leq \|Bx\|$$

como queríamos.

Ahora bien, como B es simétrica definida positiva, tenemos que existe la matriz simétrica $B^{\frac{1}{2}}$ tal que: $B^{\frac{1}{2}} B^{\frac{1}{2}} = B$. Con esto en mente y recordando la caracterización de $cos(\theta_k)$:

$$cos(\theta_k) = \frac{-\nabla f_k^T \, p_k}{\|\nabla f_k\| \, \|p_k\|}$$

tenemos que:

$$\begin{aligned}
cos(\theta_k) &= \frac{-\nabla f_k^T \, p_k}{\|\nabla f_k\| \, \|p_k\|} \\
&= \frac{\left(B_k \, p_k\right)^T p_k}{\|B_k \, p_k\| \, \|p_k\|} \quad (B_k \, p_k = -\nabla f_k \ ) \\
&= \frac{\left(B_k^{\frac{1}{2}} B_k^{\frac{1}{2}} \, p_k\right)^T p_k}{\|B_k \, p_k\| \, \|p_k\|} \\
&= \frac{p_k^T \left(B_k^{\frac{1}{2}}\right)^T B_k^{\frac{1}{2}} \, p_k}{\|B_k \, p_k\| \, \|p_k\|} \quad (B^{\frac{1}{2}} \text{ es simétrica }) \\
&= \frac{\|B_k^{\frac{1}{2}} \, p_k\|^2}{\|B_k \, p_k\| \, \|p_k\|} \\
&\geq \frac{\|B_k^{\frac{1}{2}} \, p_k\|^2}{\|B_k\| \, \|p_k\| \, \|p_k\|} \quad \text{(definición de la norma)} \\
&\geq \frac{\|p_k\|^2}{\|B_k\| \|B_k^{-\frac{1}{2}}\|^2 \, \|p_k\|^2} \quad \text{(por ecuación 1, donde } B_k^{-\frac{1}{2}} \text{ es la inversa de } B_k^{\frac{1}{2}}) \\
&\geq \frac{1}{\|B_k\| \|B_k^{-\frac{1}{2}} \, B_k^{-\frac{1}{2}}\|} \quad \text{(sub-multiplicidad de la norma)} \\
&= \frac{1}{\|B_k\| \|B_k^{-1}\|} \\
&\geq \frac{1}{M}
\end{aligned}$$

con lo que queda demostrado.

2. Recuerde que la función que se quiere minimizar es:

$$f(x) = c^T x - \sum_{j=1}^{m} log(1 - a_j^T x) - \sum_{i=1}^{n} log(1 - x_i^2)$$

Por lo tanto, el gradiente y la matriz Hessiana son respectivamente:

$$\nabla f(x)_k = c_k + \sum_{j=1}^{m} \frac{(a_j)_k}{1 - a_j^T x} + \frac{2x_k}{1 - x_k^2}$$

$$\nabla^2 f(x)_{kl} = \begin{cases} \sum\limits_{j=1}^{m} \frac{(a_j)_k \, (a_j)_l}{(1-a_j^T x)^2} & \text{si } k \neq l \\ \sum\limits_{j=1}^{m} \frac{(a_j)_k^2}{(1-a_j^T x)^2} + \frac{2(1+x_k^2)}{(1-x_k^2)^2} & \text{si } k = l \end{cases}$$

Ahora bien, note que la función original puede no tomar valores si se evalúan los logaritmos en un valor negativo. Recuerde que $x_0 = 0$ y por lo tanto, para que la función pueda ser evaluada, es necesario que:

$$1 - a_j^T \, x_1 > 0 \quad \text{y} \quad 1 - x_1^2 > 0$$

Asumiendo que $B^{-1} = I$ y α constante, tenemos para la primera desigualdad:

$$\begin{aligned} 1 - a_i^T \, x_1 > 0 &\Rightarrow 1 - a_i^T \, (x_0 - \alpha \nabla f(x_0)) > 0 \\ &\Rightarrow 1 - a_i^T \, (0 - \alpha \nabla f(0)) > 0 \\ &\Rightarrow 1 + \alpha \, a_i^T \left(c + \sum_{j=1}^{m} a_j \right) > 0 \\ &\frac{1}{-a_i^T \left(c + \sum\limits_{j=1}^{m} a_j \right)} < \alpha \\ &\Rightarrow \theta_1(a_i) < \alpha \end{aligned}$$

donde:

$$\theta_1(x) = \frac{1}{-x^T \left(c + \sum\limits_{j=1}^{m} a_j \right)}$$

Por otro lado, de la segunda desigualdad tenemos que:

$$
\begin{aligned}
1-(x_1)_i^2>0 &\Rightarrow 1-(x_0-\alpha\nabla f(x_0))_i^2>0\\
&\Rightarrow 1-\alpha^2\big(c+\sum_{j=1}^{m}a_j\big)_i^2>0\\
&\Rightarrow 1-\alpha^2\big(c_i+\sum_{j=1}^{m}(a_j)_i\big)^2>0\\
&\Rightarrow \frac{1}{\big(c_i+\sum\limits_{j=1}^{m}(a_j)_i\big)^2}>\alpha^2\\
&\Rightarrow \frac{1}{\big(c_i+\sum\limits_{j=1}^{m}(a_j)_i\big)^2}>\alpha^2\\
&\Rightarrow \frac{1}{|\big(c_i+\sum\limits_{j=1}^{m}(a_j)_i\big)|}>\alpha\\
&\Rightarrow \theta_2(i)>\alpha
\end{aligned}
$$

donde:

$$\theta_2(i)=\frac{1}{|\big(c_i+\sum\limits_{j=1}^{m}(a_j)_i\big)|}$$

Con lo anterior en mente tome:

$$\alpha_{min}=\text{ínf}\left\{\alpha\in\mathbb{R}^+\mid\alpha>\theta_1(a_j)\right\}\quad\text{y}\quad\alpha_{max}=\sup\left\{\alpha\in\mathbb{R}^+\mid\alpha<\theta_2(j)\right\}$$

Si $\alpha_{max}<\alpha_{min}$ no es posible avanzar en ninguna dirección, pero de lo contrario tenemos que el mayor α que se puede utilizar para el paso constante es:

$$\hat{\alpha}=\frac{\alpha_{max}}{1+\varepsilon}$$

Este será el α seleccionado para los pasos constantes.

Resultados

Ahora bien, procedemos a mostrar los resultados comparativos de los cuatro métodos. Para cada experimento, el parametro de parada fue que:

$$\|\nabla f(x_k)\| < 0{,}001$$

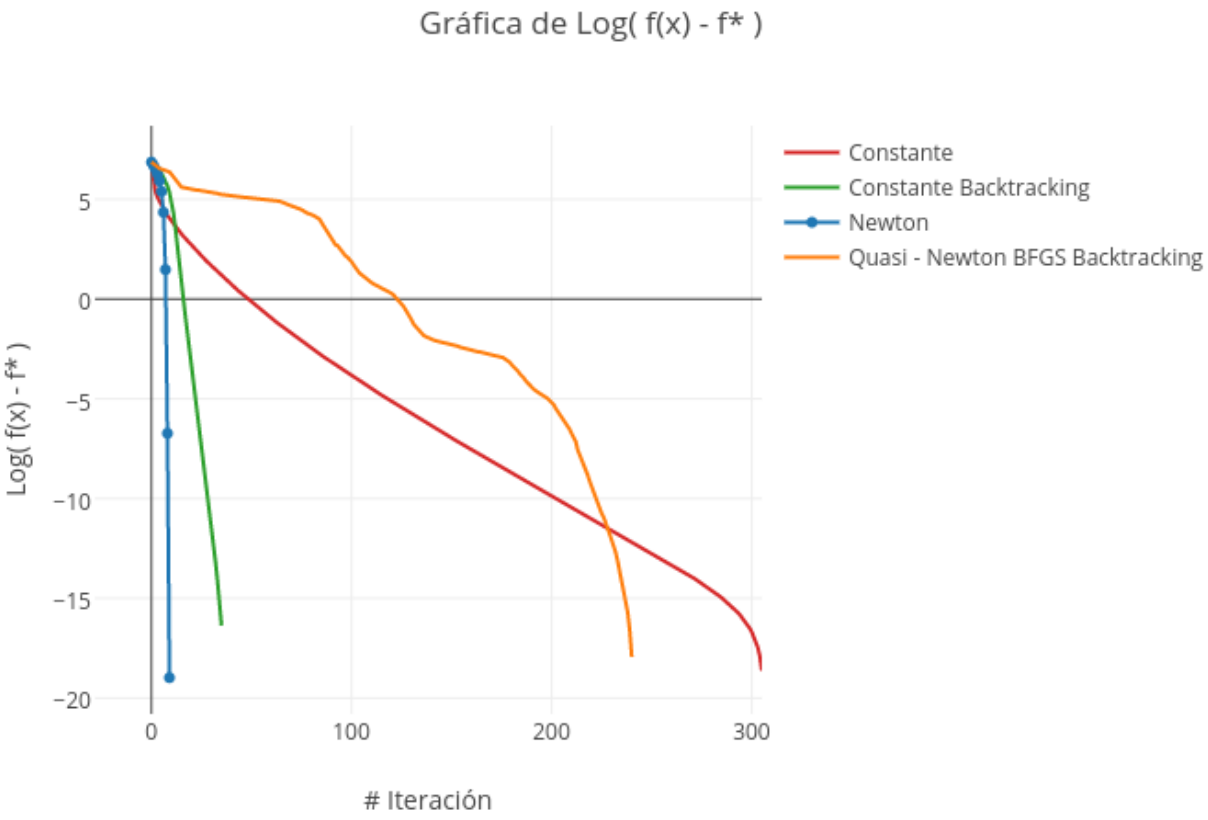

Figura 1: Gráfica comparativa de $Log(f(x) - f*)$ para cada iteración

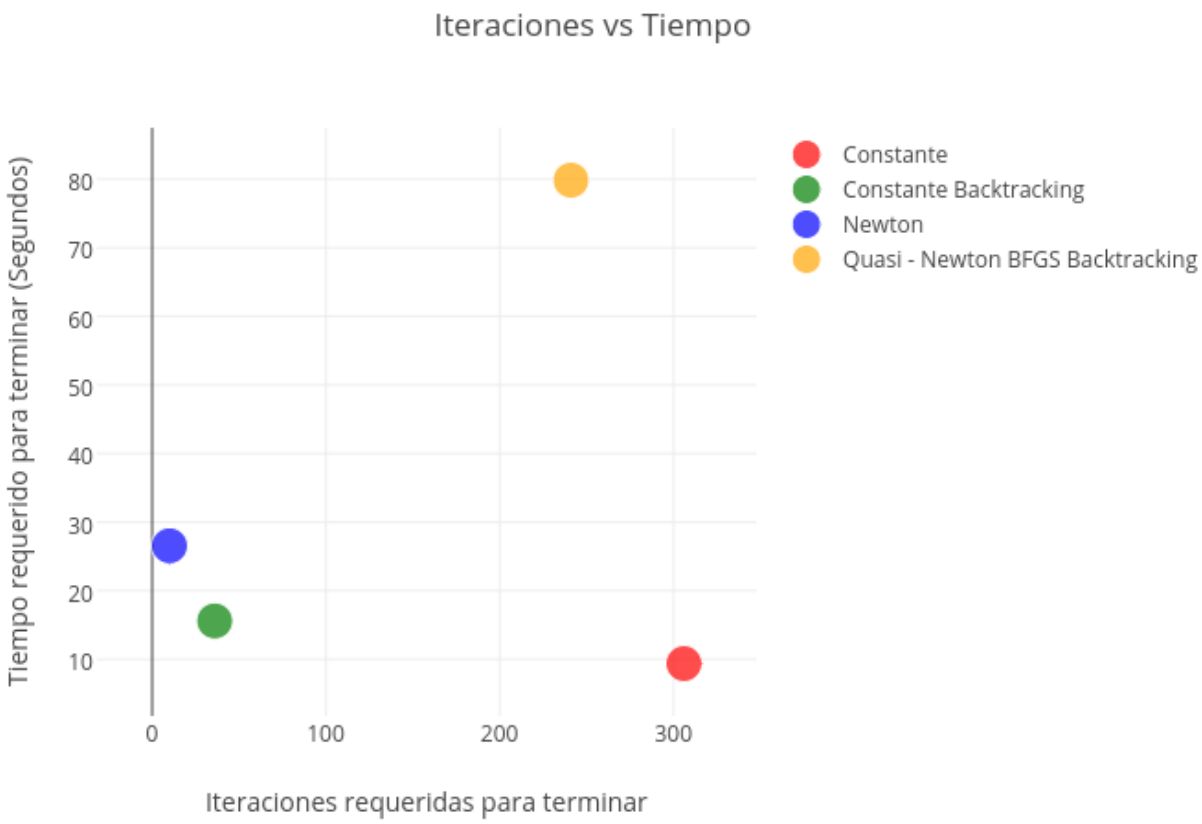

Figura 2: Gráfica comparativa de numero de iteraciones vs el tiempo requerido para terminar

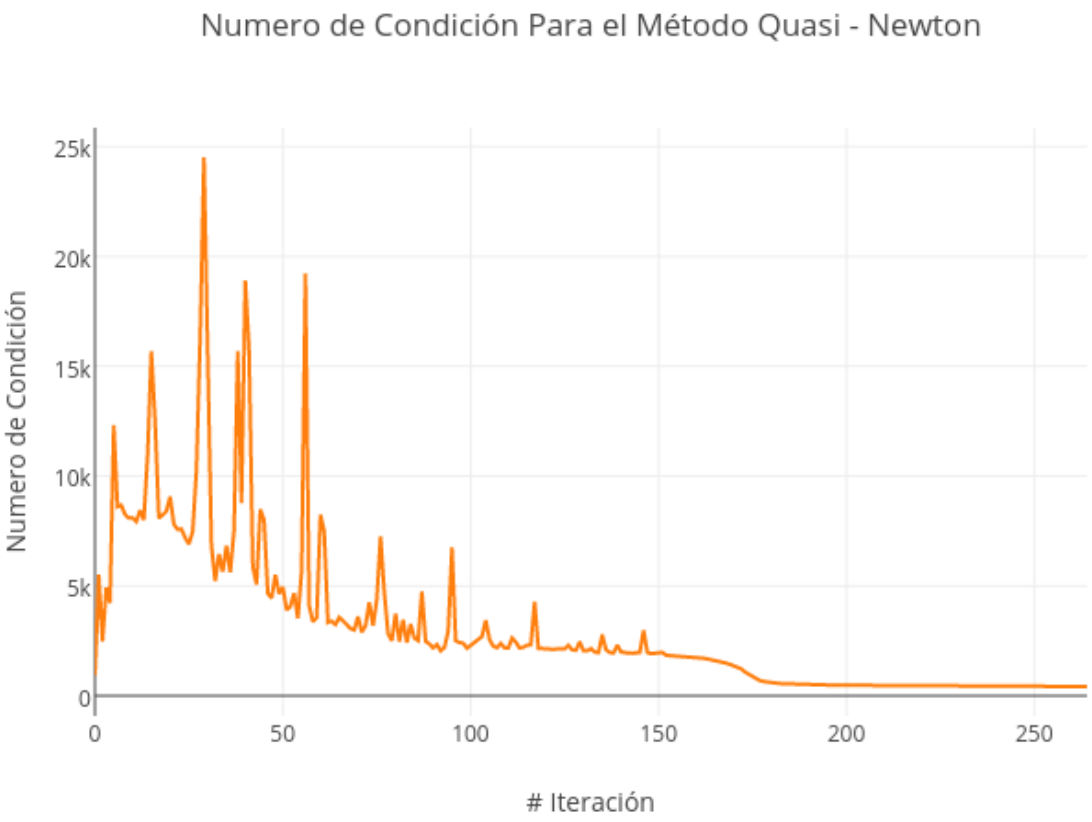

Figura 3: Gráfica de $\kappa(B_k)$ (Número de Condición) para el método Quasi - Newton BFGS con Backtracking

De las anteriores gráficas se puede ver como los mejores métodos son **Newton** y **Constante con Backtracking**.

Ahora bien se quiere resolver el problema:

$$\text{minimizar } f(x) \text{ con } n = 2500,\ m = 1000 \text{ y } \varepsilon = 0{,}000001$$

Pero, recuerde que a pesar de que el método de Newton requiere menos iteraciones, es necesario resolver un sistema lineal en cada iteración, por lo que lo hace un método muy sensible a la dimensión del espacio de salida. Por lo tanto, teniendo en cuenta las dimensiones del problema planteado, se utilizó el método **Constante con Backtracking** para resolverlo llegando a que la solución es:

$$\text{mín } f(x) = -6373{,}18027611$$

y requirió 52 iteraciones y 4.7 minutos en terminar[1].

3. Para este punto, se quiere minimizar la función: $f : \mathbb{R}^2 \to \mathbb{R}$ definida como:

$$h(x) = c^T x \quad \text{con: } c = (1, 1)$$

sujeto a la restricción:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}$$

Note que la forma de la matriz anterior quiere decir que el área factible corresponde al cuadrado de lado 100 unidades centrado en el origen. Además, se puede visualizar la función $z = h(x)$ como el plano xy en $\mathbb{R}^3$ inclinado $\pi/4$ sobre la recta $x - y = 0$, llegando a que la función disminuye a medida que se avanza en dirección $p = (-1, -1)$.

Con lo anterior en mente, se tiene que teóricamente el mínimo de esta función ocurre en la esquina inferior izquierda del cuadrado, en la coordenada $(-100, -100)$.

Ahora bien, siguiendo el método de la barrera logarítmica, se quiere minimzar la función:

$$\hat{h}(x) = c^T x - \sum_{j=1}^{4} log(100 - a_j^T x)$$

con:

$$a_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad a_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad a_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ y } \quad a_4 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Una función muy similar a la del punto anterior. Además, note que las funciones de restricción en este caso son:

$$g_j(x) = a_j^T\, x - 100$$

Con esto en mente, se realizó un descenso del gradiente **Constante con Backtracking** de $\hat{h}$, con $x_0 = [25, 75]$ y parando de iterar cuando

$$\|\nabla \hat{h}\| < 0{,}000001$$

[1]Se ejecutó este mismo escenario pero con el método de Newton y se obtuvo el mismo mínimo en 13 iteraciones pero tomó 31 minutos en completar, como era de esperar

La siguiente gráfica muestra el camino que se siguió para dicho óptimo:

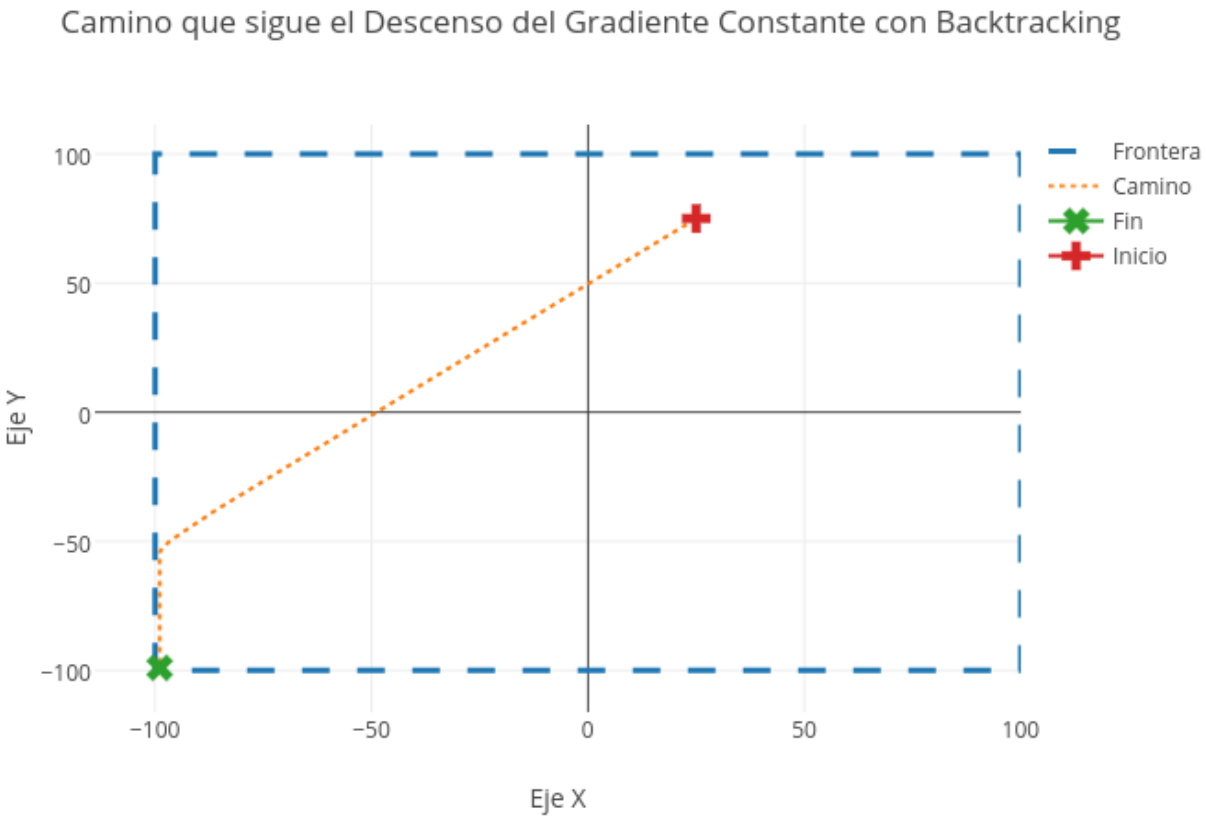

Figura 4: Resultados Experimento final.

En resumen, se realizaron 207 iteraciones, llegando a que el mínimo de la función ocurre en $x = [-99{,}004, 99{,}004]$, un valor coherente con el mínimo teórico.

El Código

Por completitud se incluye el código utilizado para realizar la tarea. Se desarrollo en python 2.7 y se encuentra público en el link:

`https://github.com/minigonche/gradient_descent`

Codigo Fuente

```
#Script for experiments with gradient descent

#-------------------------------------------------------------------------
#------------------------- Script Imports --------------------------------
#-------------------------------------------------------------------------
#To make division easier
from __future__ import division
#For matrix and vector structures
import numpy as np
#For math operations
import math as math
#For system requirements
import sys
#For time measurement
import time

#For Graphing. The methods export the grapgics on plotly, the user only needs
# to enter his/her username and api-key
import plotly.plotly as py
import plotly.graph_objs as go
py.sign_in('minigonche', '8cjqqmkb4o')

#-------------------------------------------------------------------------
#------------------------- Global Variables ------------------------------
#-------------------------------------------------------------------------

n = 500
m = 200
c = np.random.rand(1,n)
#Puts each a_j as a column of the following matrix
A = np.random.rand(n,m)
#Global constant alpha
global_alpha = 0.001
#GLobal epsilon for treshold
global_eps = 0.001
#global difference measure for gradient
global_dif = 0.000001
#Measure how many iterations to print pogress
print_counter = 20


#Final experiment variables
#-----------------------------------------------
c_e = np.random.rand(1,2)
c_e[0,0] = 1
c_e[0,1] = 1

#Puts each a_j as a column of the following matrix
A_e = np.random.rand(2,4)
A_e[0,0] = 1
A_e[0,1] = -1
A_e[0,2] = 0
A_e[0,3] = 0
```

```
A_e[1,0] = 0
A_e[1,1] = 0
A_e[1,2] = 1
A_e[1,3] = -1

cube = 100

global_eps_e = 0.000001


#-------------------------------------------------------------------------
#------------------------- Main Methods ----------------------------------
#-------------------------------------------------------------------------


#NOTE: Vectors are assumed as matrix of dimension 1 x n
#Runs the gradient descent with the given parameters
#Serves as a unified method
def run_gradient_descent(dim,
                         fun,
                         gradient,
                         alpha,
                         B_matrix,
                         eps,
                         inverse = True,
                         initial = None):
    """
        Parameters
        ----------
        dim : int
            The dimension of the vector that the function and gradient receive.
            The domain's dimension of the function we wish to minimize
        fun : function(numpy.vector)
            The function we wish to minimize
        gradient : functon(numpy.vector)
            A functon that receives a numpy.vecotr (real vector: x_k) and
            returns the gradient (as a numpy.vector) of the given function
            evaluated at the real number received as parameter
        alpha : function(numpy.vector, numpy.vector)
            A functon that receives two numpy.vecotors (real vectors:
            x_k and p_k ) and returns the next alpha step
        B_matrix : function(np.matrix, numpy.vector)
            A function that receives a numpy.matrix (the previous matrix) and
            numpy.vecotr (real vector) and returns the next multiplication
            np.matrix
        eps : float
            The epsylon that serves as a stopping criteria for the algorithm
        solve : boolean
            Indicates if the B_matrix method gives the B or the B^-1 matrix
        Initial : np:vector
            The initial vector. If None is received, then the procedure strarts
            at zero.
    """
    #Starts the timer
    start_time = time.time()

    #Initial values
```

```
#The first alpha and B matrix are initialized at None

x = initial
if x is None:
    x = np.zeros((1,dim))

x_last = np.zeros((1,dim))
grad_last = np.zeros((1,dim))
B = None
a = None
p = None

#Treshold
treshold = False

#printing variables
count = 1
global_count = 0

#Graphing variables
x_variables = []
function_values = []


#Becomes true when |f(x_n+1) - f(x_n)| < eps
while(not treshold):
    #Saves the Value
    x_variables.append(x)
    function_values.append(fun(x))

    #Calculates the necesarry advancing parameters
    x_actual = x

    B = B_matrix(B, x_actual,x_last)
    grad = gradient(x_actual)

    #Calcultaes the next value
    if inverse:
        p = (-1)*B.dot(grad.T).T
    else:
        p = (-1)*np.linalg.solve(B, grad.T).T


    #raw_input('espere')

    a = alpha(x_actual, p)
    x = x_actual + a*p
    x_last = x_actual

    #Checks the the treshold
    treshold = np.linalg.norm(grad) < eps
               or np.linalg.norm(grad - grad_last) < global_dif

    if count == print_counter:
        print(np.linalg.norm(grad))
        count = 0
```

```
            count = count + 1
            global_count = global_count +1
            grad_last = grad


        x_final = x
        value_final = fun(x)


        return [x_final,
                value_final,
                x_variables,
                function_values,
                global_count,
                time.time() - start_time]

#end of run_gradient_descent

#Graphing method
def plot_log(function_values, value_final):
    """
        Parameters
        ----------
        function_values : np.array
            An array of the value of the function at the given iteration
        value_final : float
            The mimimum value achieved in the optimization
    """
    #Graphs the plot
    dif = map(lambda y: math.log(y - value_final ),function_values)
    #Draws the initial trace
    trace = go.Scatter(x = range(len(dif)), y =  dif)

    #Export graph
    plot_url = py.plot([trace], auto_open=False)

#end plot_log

#Graphing method
def plot(x_values, y_values):


    #Draws the initial trace
    trace = go.Scatter(x = x_values, y =  y_values)

    #Export graph
    plot_url = py.plot([trace], auto_open=False)

#end plot_log


#-------------------------------------------------------------------------
#------------------------ Experiment Start -------------------------------
#-------------------------------------------------------------------------


#CENTRAL FUNCTION
```

```
#Declares the global function, its gradient and its Hessian
def main_function(x):

    first_term = c.dot(x.T)[0,0]
    second_term = (-1)*sum(map(lambda a: math.log(1 - a.dot(x.T)), A.T))
    third_term = (-1)*sum(map(lambda y: math.log(1 - y**2), x.T))

    return(first_term + second_term + third_term)
#end of main_function

def main_gradient(x):
    first_term = np.array(c)
    #print(first_term.shape)

    #Calculates the common vector in each coordinate
    temp_vec = np.array(map(lambda a_column: 1/(1 - a_column.dot(x.T)), A.T))

    second_term = np.array(map(lambda a_row:  a_row.dot(temp_vec), A)).T

    third_term = np.array(map(lambda y: 2*y/(1 - y**2), x))


    return(first_term + second_term + third_term)
#end of main_gradient

def main_hessian(x):

    #Calculates the common vector in each coordinate
    temp_vec = np.array(map(lambda a_column: 1/(1 - a_column.dot(x.T))**2, A.T)).T


    #There is probably a faster way to do this, but since for this case the
    # Hessian matrix corresponds to a symetric matrix:
    hessian = np.zeros((n,n))
    for i in range(n):
        for j in range(i,n):
            row = np.matrix(A[i,:]*A[j,:])
            value = np.dot(row,temp_vec.T)
            if(i == j):
                value = value + 2*(1+(x[0,i]**2))/((1-(x[0,i]**2))**2)

            hessian[i,j] = value
            hessian[j,i] = value

    return(hessian)
#end of main_hessian

#-----------------------------------
#EXPERIMENT FUNCTION


#Declares the global function, its gradient
def exp_function(x):

    first_term = c_e.dot(x.T)[0,0]
    second_term = (-1)*sum(map(lambda a: math.log(cube - a.dot(x.T)), A_e.T))
```

```
    return(first_term + second_term )
#end of exp_function

def exp_gradient(x):
    first_term = np.array(c_e)

    #Calculates the common vector in each coordinate
    temp_vec = np.array(map(lambda a_column: 1/(cube - a_column.dot(x.T)), A_e.T))
    second_term = np.array(map(lambda a_row:  a_row.dot(temp_vec), A_e)).T

    return(first_term + second_term )
#end of exp_gradient

#-------------------------
#-----Gradient Descent-----
#-------------------------

#First declares the global constant and backtracking method for alpha
#Calculates the max alpha given the restriction of the logarithms
def max_alpha():
    cons = c + np.matrix(sum(A.T))
    theta_1 = max(map(lambda a_column: -1/np.dot(a_column,cons.T) , A.T))
    theta_2 = min(map(lambda v: 1/math.fabs(v), cons.T))
    if(theta_2 < theta_1):
        raise ValueError('No suitable alphas exist')
    return theta_2/(1+0.01)

constant_alpha = max_alpha()

def alpha_constant(x, p):
    return constant_alpha
#end of  alpha_constant

def alpha_global(x, p):
    return global_alpha
#end of  alpha_global

def alpha_backtracking(x, p):
    #For the first iteration
    if(p is None):
        return global_alpha

    a = 1
    flag = True
    while(flag):
        try:
            main_function(x + a*p)
            flag = False
        except ValueError:
            a = a/2

    rho = 4/5
    c = 4/5
    while(main_function(x + a*p) > main_function(x) + c*a*np.dot(main_gradient(x),p.T) ):
        a = rho*a
```

```
    return a
# end of alpha_backtracking

#Alpha backtracking for experiment
def alpha_backtracking_exp(x, p):
    #For the first iteration
    if(p is None):
        return global_alpha_e

    a = 1
    flag = True
    while(flag):
        try:
            exp_function(x + a*p)
            flag = False
        except ValueError:
            a = a/2

    rho = 4/5
    c = 4/5
    while(exp_function(x + a*p) > exp_function(x) + c*a*np.dot(exp_gradient(x),p.T) ):
        a = rho*a

    return a
# end of alpha_backtracking

#-------------------------------------
#---- Constant ------

#Runs the constant example
def run_constant():

    B_const = np.identity(n)
    B_matrix_fun = lambda B, x, x_prev: B_const

    result = run_gradient_descent(dim = n,
                                  fun = main_function,
                                  gradient = main_gradient,
                                  alpha = alpha_constant,
                                  B_matrix = B_matrix_fun,
                                  eps = global_eps )

    return result
#end of run_constant

#-------------------------------------
#---- Constant  Backtracking ------
#Runs the constant eith backtracking example
def run_constant_backtracking():
    B_const = np.identity(n)
    B_matrix_fun = lambda B, x, x_prev: B_const

    result = run_gradient_descent(dim = n,
                                  fun = main_function,
                                  gradient = main_gradient,
                                  alpha = alpha_backtracking,
                                  B_matrix = B_matrix_fun,
```

```
                                    eps = global_eps )

    return result
#end of run_constant_backtracking


#-------------------------------------
#---- Newton ------
#Runs the constant example
def B_matrix_hessian(B, x, x_prev):
    return main_hessian(x)

def run_newton():

    alpha_newton = lambda a,p: 1

    result = run_gradient_descent(dim = n,
                                  fun = main_function,
                                  gradient = main_gradient,
                                  alpha = alpha_newton,
                                  B_matrix = B_matrix_hessian,
                                  eps = global_eps,
                                  inverse = False)

    return result
#end of run_newton


#-------------------------------------
#---- Newton BFGS ------

condition_numbers = []

#Primero se declara la funcion que se encarga de
def BFGS(B, x_actual, x_last):
    if B is None:
        return np.identity(n)


    #Calculates temporal next value
    grad = main_gradient(x_actual)
    p = (-1)*np.linalg.solve(B, grad.T).T
    x_next = x_actual + global_alpha*p

    #x_next = x_actual
    #x_actual = x_last

    s = x_next - x_actual
    y = main_gradient(x_next) - main_gradient(x_actual)
    first_term = B
    second_term = (-1)*np.dot(B.dot(s.T), s.dot(B))/(np.dot(s, B.dot(s.T)))
    third_term = np.dot(y.T,y)/np.dot(y, s.T)

    final_b = first_term + second_term + third_term

    condition_numbers.append(np.linalg.cond(final_b))
```

```
    return final_b


def run_BFGS():

    result = run_gradient_descent(dim = n,
                                  fun = main_function,
                                  gradient = main_gradient,
                                  alpha = alpha_backtracking,
                                  B_matrix = BFGS,
                                  eps = global_eps,
                                  inverse = False)
    return result


#-------------------------------------
#---- Final Experiment -------

#Runs the constant example
def run_experiment():


    x = np.zeros((1,2))
    x[0,0] = 25
    x[0,1] = 75

    B_const = np.identity(2)
    B_matrix_fun = lambda B, x, x_prev: B_const


    result = run_gradient_descent(dim = 2,
                                  fun = exp_function,
                                  gradient = exp_gradient,
                                  alpha = alpha_backtracking_exp,
                                  B_matrix = B_matrix_fun,
                                  eps = global_eps_e,
                                  initial = x)

    return result
#end of run_constant
```