9

Métodos Numéricos

Trabajo práctico N°5

Optimización

Grupo 11: Martin Amagliani
Olivia De Vincenti
Nicolás Fernandez Pelayo

Profesor: Pablo Ignacio Fierens

Fecha de entrega: 20 de noviembre de 2020

Buenos Aires, Argentina

Consigna

Grupo impar

1. Escriba una función que implemente la minimización basada en el método de cuasi-Newton BFGS usando interpolación cuadrática para la búsqueda lineal. La función debe tener el nombre `minimi` y recibir el handle a la función a minimizar, el handle a su gradiente, el punto inicial de la búsqueda(como arreglo de `Numpy`), la tolerancia de finalización y el número máximo de iteraciones, en ese orden. La tolerancia se debe interpretar así: si la norma de la diferencia entre dos puntos consecutivos es menor a la tolerancia, entonces el algoritmo debe terminar.

2. Escriba una función para que, sobre la base de `minimi` busque los parámetros a, c, d, T_2 y T_3 que minimicen:

$$\sum_i \left| y_i - \left(a + c\cos\left(2\pi \frac{t_i}{T_2}\right) + d\cos\left(2\pi \frac{t_i}{T_3}\right)\right)\right|^2 \tag{1}$$

donde t_i, y_i están en la primera y segunda columna del archivo `temp.txt`, respectivamente. Los valores de y_i se corresponden con la medición de temperatura corporal de una persona en cada hora. El nombre de la función en este ítem debe ser `temperatura`, no debe recibir parámetros y debe devolver: un arreglo `Numpy` con los valores ajustados, un arreglo con el error de ajuste.

3. Escriba una función que pruebe el correcto funcionamiento de todas las demás funciones implementadas. La función debe llamarse `test`, sin argumentos.

Todas las funciones deben estar en un archivo denominado `temperamental.py`.

Funciones

Índice de Códigos

1. Ejercicio 1

1.1. minimi

Minimiza la función dada por el método de quasi-Newton (2) BFGS (3) usando interpolación polinomial para la búsqueda lineal.

$$\begin{aligned} \vec{d_k} &= -\mathbf{B}_k \nabla f(\vec{x}_k) \\ \alpha_k &= argmin_\alpha f(\vec{x}_k + \alpha \vec{d_k}) \\ \vec{x}_{k+1} &= \vec{x}_k + \alpha_k \vec{d_k} \end{aligned} \tag{2}$$

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \left[\frac{(\vec{s}_k^T \vec{y}_k + \vec{y}_k^T \mathbf{B}_k \vec{y}_k)(\vec{s}_k \vec{s}_k^T)}{(\vec{s}_k^T \vec{y}_k)^2}\right] - \left[\frac{\mathbf{B}_k \vec{y}_k \vec{s}_k^T + \vec{s}_k \vec{y}_k^T \mathbf{B}_k}{(\vec{s}_k^T \vec{y}_k)^2}\right] \tag{3}$$

Se tomó como parámetro para finalizar la iteración cuando la norma de la diferencia entre la función evaluada en el punto y el anterior es menor a la tolerancia. Además, se consideró que cuando la dirección d es muy chica (en este caso menor al épsilon de máquina) el mínimo se encuentra muy cerca del punto actual y es razonable tomarlo como mínimo.

Código 1: Definición de minimi

```
def minimi(f, vf, x0, tol, max_i):
    x_1 = x0
    B = np.diag(np.full(vf(x0).shape, 0.1))            # Creo el B base
    i = 0
    fin = True

    while i < max_i and fin:
        x_0 = x_1                                       # Actualizo x
        d = -np.dot(B, vf(x_0))                         # Calculo d
        if np.linalg.norm(d) >= np.finfo(float).eps:
            g = lambda alpha: f(x_0 + alpha * d)
            a = argmin(g, 0, 1)                         # Calculo alfa que minimice f
            x_1 = x_0 + a * d                           # Calculo nueva x

            if np.linalg.norm(x_1 - x_0) >= tol and np.linalg.norm(d) >= np.finfo(float).eps:        # Me fijo si terminó
                s = a * d
                y = vf(x_1) - vf(x_0)
                B += ((s.T@y + (y.T@B)@y) * (s@s.T)) / (s.T@y) ** 2 - ((B@y)@s.T + s@(y.T@B)) / (s.T@y)  # Aplico BFGS
                i += 1
                # print("iteración ", i, ": min =", x_1)
            else:
                fin = False                             # Si terminó, salgo del loop
        else:
            fin = False

```

Ya fue evaluado el gradiente en x_0. En cada iteración hay una evaluación de más.

Acá hay algunos productos que se repiten.

```
    print("Se realizaron", i, "iteraciones")

    return x_1
```

1.2. argmin

Realiza la interpolación cuadrática y encuentra el mínimo del polinomio obtenido.

Código 2: Definición de argmin

Repite demasiadas veces las evaluaciones de función. Salvo por la primera vuelta, sólo una evaluación de función por iteración es necesaria usando el truco de duplicar o dividir por dos el ancho del intervalo.

```
def argmin(f, x, h0):
    falta = True
    h = h0
    while falta and h > 1e-10:                  # Se define h = 1e-10 como el menor h soportado
        if f(x) > f(x + h) > f(x + 2*h):         # Si no encuentra un mínimo,
            h = 2*h                             # Duplica el paso
        elif f(x) < f(x + h) < f(x + 2*h):       # Si se pasa del mínimo,
            h = h / 2                           # Toma la mitad del paso
        else:
            falta = False                       # Encontró un mínimo

```

En caso que se salga porque h<= 1e-10: ¿por qué se procede de la forma prescripta en esta función?

```
    ya = f(x)
    yb = f(x + h)
    yc = f(x + 2*h)
    hmin = (4*yb - 3*ya - yc) / (4*yb - 2*ya - 2*yc) * h       # Calcula el mínimo del polinomio
     ↪ interpolador
    return x + hmin
```

2. Ejercicio 2

2.1. temperatura

Busca los valores que minimizan (1) usando `minimi`. También calcula el error de ajuste. Las condiciones iniciales fueron estimadas por tanteo, creemos que cerca de estos valores se encuentra uno de los mínimos locales más bajos.

Código 3: Definición de temperatura

```
def temperatura():
    va = minimi(fun, vfun, np.array([36.19, 0.0001, -0.0001, 0.8, 1.01]), np.finfo(float).eps, 100)
    err = fun(va)

    return va, err
```

¿Por qué estos valores?

2.2. fun

Handle de la función (1).

Código 4: Definición de fun

```
def fun(x):
    return np.sum(funaux(y, t, x[0], x[1], x[2], x[3], x[4]) ** 2)

def funaux(yi, ti, a, c, d, T2, T3):
    return yi - (a + c * np.cos(2 * np.pi * ti / T2) + d * np.cos(2 * np.pi * ti / T3))
```

2.3. vfun

Handle del gradiente de la función (1) calculado a partir de sus derivadas parciales.

Código 5: Definición de vfun

```
def vfun(x):
    grad = np.zeros(5)
    grad[0] = np.sum(fun_a(y, t, x[0], x[1], x[2], x[3], x[4]))
    grad[1] = np.sum(fun_c(y, t, x[0], x[1], x[2], x[3], x[4]))
    grad[2] = np.sum(fun_d(y, t, x[0], x[1], x[2], x[3], x[4]))
    grad[3] = np.sum(fun_T2(y, t, x[0], x[1], x[2], x[3], x[4]))
    grad[4] = np.sum(fun_T3(y, t, x[0], x[1], x[2], x[3], x[4]))
    return grad
```

Obsérvese que, para calcular el gradiente, se llama 5 veces a la función funaux() con los mismos parámetros.

Código 6: Handles de las derivadas parciales de fun

```
###################################################
# Derivadas parciales de fun
# ---------------------------------------------------------------------------------------------------------------
def fun_a(yi, ti, a, c, d, T2, T3):
    return -2 * funaux(yi, ti, a, c, d, T2, T3)
# ---------------------------------------------------------------------------------------------------------------
def fun_c(yi, ti, a, c, d, T2, T3):
    return -2 * np.cos(2 * np.pi * ti/T2) * funaux(yi, ti, a, c, d, T2, T3)
# ---------------------------------------------------------------------------------------------------------------
def fun_d(yi, ti, a, c, d, T2, T3):
    return -2 * np.cos(2 * np.pi * ti/T3) * funaux(yi, ti, a, c, d, T2, T3)
# ---------------------------------------------------------------------------------------------------------------
def fun_T2(yi, ti, a, c, d, T2, T3):
    return -4 * np.pi * c * ti * np.sin(2 * np.pi * ti/T2) * funaux(yi, ti, a, c, d, T2, T3) / T2**2
# ---------------------------------------------------------------------------------------------------------------
def fun_T3(yi, ti, a, c, d, T2, T3):
    return -4 * np.pi * d * ti * np.sin(2 * np.pi * ti/T3) * funaux(yi, ti, a, c, d, T2, T3) / T3**2
###################################################
```

3. Ejercicio 3

3.1. test

Función de prueba. Se evalúan `minimi`, y `temperatura`. Para la primer función, se buscaron funciones con mínimos conocidos para teatear.
Los casos a considerar son:

- Esfera en $\mathbb{R}^3$ corrida
$$f(x, y, z) = x^2 + (y-1)^2 + (z-2)^2$$
Mínimo conocido: (0, 1, 2)

- Función de Beale
$$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$$
Mínimo conocido: (3, 0.5)

- Esfera en $\mathbb{R}^8$
$$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 + x_6^2 + x_7^2 + x_8^2$$
Mínimo conocido: (0, 0, 0, 0, 0, 0, 0, 0)

- Función de cabina
$$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$
Mínimo conocido: (1, 3)

- Función Matyas
$$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$$
Mínimo conocido: (0, 0)

Falta la cita al final del informe.

La mayoría de estas funciones se obtuvieron de una página web que las recomienda para testear algoritmos de optimización. Sabemos que no es recomendable utilizar fuentes no oficiales, pero debido a nuestro conocimientos adquiridos en Matemática III creemos que las funciones elegidas son adecuadas. Se consideró que los resultados coinciden cuando el módulo de la diferencia entre cada componente es menor que 10^{-9}.

¿?

Código 7: Definición de test

```python
def test():
    tb = np.array([
        np.array(["Esfera corrida en R3", f1, vf1, np.array([0, 0, 0]), np.finfo(float).eps, 100,
                                                                                    np.array([0.0, 1.0, 2.0])   ]),
        np.array(["Función de Beale", fbeale, vfbeale, np.array([0, 0]), np.finfo(float).eps, 1000,
                                                                                    np.array([3.0, 0.5])        ]),
        np.array(["Esfera en R^8", fesfr8, vfesfr8, np.array([1, 1, 1, 1, 1, 1, 1, 1]), np.finfo(float).eps, 100,
                                                              np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])]),
        np.array(["Función de cabina", fcab, vfcab, np.array([0, 0]), np.finfo(float).eps, 100,
                                                                                    np.array([1.0, 3.0])        ]),
        np.array(["Función de Matyas", fmat, vfmat, np.array([1, 2]), np.finfo(float).eps, 200,
                                                                                    np.array([0.0, 0.0])        ])])

    for i in range(len(tb)):
        print("\n" + tb[i][0])                                    # Print nombre de la función
        m = minimi(tb[i][1], tb[i][2], tb[i][3], tb[i][4], tb[i][5])    # minimi de la función i de tb
        print(m)
        err = False
        for j in range(len(tb[i][6])):                            # Compruebo que cada punto minimo encontrado sea
            if np.abs(m[j] - tb[i][6][j]) > 1e-9:                 # aproximadamente el conocido
                print("ERROR: Los valores obtenidos no coinciden con los conocidos")
                print("Mínimo conocido:", tb[i][6])
                err = True
                break
        if not err:
            print("Caso exitoso: El mínimo obtenido coincide con el conocido")
    print(temperatura())                                          # Función temperatura ejercicio 2
    return
```

Código 8: Handles de las funciones de prueba

```
#################################################
# Función Esfera en R^8 utilizada para testear el algoritmo
# ----------------------------------------------------------------------------------------
def fesfr8(x):
    return x[0]**2 + x[1]**2 + x[2]**2 + x[3]**2 + x[4]**2 + x[5]**2 + x[6]**2 + x[7]**2
#################################################
# Gradiente de la función Esfera
# ----------------------------------------------------------------------------------------
def vfesfr8(x):
   return np.array([2*x[0], 2*x[1], 2*x[2], 2*x[3], 2*x[4], 2*x[5], 2*x[6], 2*x[7]])
#################################################

#################################################
# Función Beale utilizada para testear el algoritmo
# ----------------------------------------------------------------------------------------
def fbeale(x):
    return (1.5 - x[0] + x[0]*x[1])**2 + (2.25 - x[0] + x[0]*x[1]**2)**2 + (2.625 - x[0] + x[0]*x[1]**3)**2
#################################################
# Gradiente de la función Beale
# ----------------------------------------------------------------------------------------
def vfbeale(x):
   return np.array([0.25*(x[1]-1)*((8*x[1]**5 + 8*x[1]**4 + 16*x[1]**3 - 8*x[1] - 24)*x[0] + 21*x[1]**2 + 39*x[1] + 51), 0.25*x[0]*(24*x[0]*x[1]**5 + 16*x[0]*x[1]**3 + (63 - 24*x[0])*x[1]**2 + (36-8*x[0])*x[1] - 8*x[0] + 12)])
#################################################

#################################################
# Función de cabina utilizada para testear el algoritmo
# ----------------------------------------------------------------------------------------
def fcab(x):
    return (x[0] + 2*x[1] - 7)**2 + (2*x[0] + x[1] - 5)**2
#################################################
# Gradiente de la función de cabina
# ----------------------------------------------------------------------------------------
def vfcab(x):
   return np.array([10*x[0] + 8*x[1] - 34, 10*x[1] + 8*x[0] - 38])
#################################################

#################################################
# Función Matyas utilizada para testear el algoritmo
# ----------------------------------------------------------------------------------------
def fmat(x):
    return 0.26*(x[0]**2 + x[1]**2) - 0.48*x[0]*x[1]
#################################################
# Gradiente de la función de cabina
# ----------------------------------------------------------------------------------------
def vfmat(x):
   return np.array([(13*x[0] - 12*x[1])/25, (13*x[1] - 12*x[0])/25])
#################################################
```