Tarea8

November 1, 2020

1 Tarea 8

Tarea 8 de *Benjamín Rivera* para el curso de **Métodos Numéricos** impartido por *Joaquín Peña Acevedo*. Fecha limite de entrega **1 de Noviembre de 2020**.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as LA
import scipy as sp
import seaborn as sns

from scipy.linalg import solve_triangular # Para backward y forward substitution
from scipy.linalg import svdvals


NOTEBOOK = True
```

1.1 Ejercicio 1

Programar y probar el método de descomposición en valores singulares. Anque algoritmo permite factorizar una matriz rectangular, el programa se va a probar con matrices cuadradas para revisar su número de condición y el uso de la factorización para resolver sistemas de ecuaciones.

1.1.1 Implementar Algoritmo 1

Escriba una función que implemente el Algoritmo 1 para obtener las matrices U, V y el arreglo s de la descomposición en valores singulares de una matriz A.

[2]:
```
def reordenar_matriz(mat, orden):
    if type(mat) == np.matrix:
        return np.concatenate([mat[:, i] for i in orden],
                              axis=1)
    else:
        return [mat[i] for i in orden]


def ordenar(M1: np.matrix, M2: np.matrix, v: list) -> tuple:
    """ Funcion para reordenar las columnas de las
```

```
    matrices como se pide en Algoritmo 1.
    """
    # Buscamos el orden en funcion de s O(nlog n)
    orden = [i for i in range(len(v))]
    orden.sort(key=lambda i: v[i],
               reverse=True)
    # -------------------------------
    # Reordenamos los recibidos O(3n)
    M1 = reordenar_matriz(M1, orden)
    M2 = reordenar_matriz(M2, orden)
    v = reordenar_matriz(v, orden)
    # --------------------------
    # Regresamos los reordenados
    return M1, M2, v


if NOTEBOOK and True:
    """ Parte para probar el uso de ordenar. """
    n = 3
    U = np.matrix(np.random.rand(n, n), dtype=np.float16)
    V = np.matrix(np.random.rand(n, n), dtype=np.float16)
    s = list(np.random.rand(1, n).flatten())
    print(U, end='\n\n')
    print(V, end='\n\n')
    print(s, end='\n\n')

    print('-'*n*12)
    U, V, s = ordenar(U, V, s)
    print(U, end='\n\n')
    print(V, end='\n\n')
    print(s, end='\n\n')
```

```
[[0.00699 0.557   0.211  ]
 [0.8823  0.5234  0.873  ]
 [0.1685  0.4473  0.3882 ]]

[[0.3032 0.7275 0.7104]
 [0.956  0.709  0.4727]
 [0.1611 0.85   0.8247]]

[0.9999067265913241, 0.3135876751977431, 0.585319352113929]

------------------------------------
[[0.00699 0.211   0.557  ]
 [0.8823  0.873   0.5234 ]
 [0.1685  0.3882  0.4473 ]]
```

```
[[0.3032 0.7104 0.7275]
 [0.956  0.4727 0.709 ]
 [0.1611 0.8247 0.85  ]]

[0.9999067265913241, 0.585319352113929, 0.3135876751977431]
```

[3]:
```python
def Algoritmo1(A: np.matrix, m: int, n: int, T=None, N=100, dtype=np.float64):
    """ Descomopsocion en valores singulares.

    Input:
        A := Matriz de mxn con columnas $a_i, i\in [0,n]$
    """
    # En caso de no pasar tolerancia lo ponemos al epsilon del tipo de dato
    if T is None:
        T = np.finfo(dtype).eps ** (1 / 2)

    if not isinstance(A, np.matrix):
        try:
            A = np.matrix(A)
        except:
            raise Exception("=( nshe k pz")

    # Inicializar valores
    V = np.matrix(np.identity(n),
                  dtype=dtype)
    k = 0
    F = 1

    while k < N and F > 0:
        k = k+1
        F = 0
        for i in range(n-1):
            for j in range(i+1, n):
                alpha = dtype(A[:, i].transpose()*A[:, i])
                gamma = dtype(A[:, j].transpose()*A[:, j])
                beta  = dtype(A[:, i].transpose()*A[:, j])

                if alpha*gamma > np.finfo(dtype).eps and abs(beta) >␣
↪T*alpha*gamma:
                    F = 1
                    if beta != 0:
                        nu = (gamma - alpha)/(2*beta)
                        t = 1/(abs(nu) + np.sqrt(1 + nu**2))
                        if nu < 0:
                            t = -t
                        c = 1/np.sqrt(1 + t**2)
```

```
                    s = t*c
                else:
                    c = 1
                    s = 0

                # Modificacion de A
                a = A[:, i]
                b = A[:, j]

                A[:, i] = c*a - s*b
                A[:, j] = s*a + c*b

                # Modificacion de V
                a = V[:, i]
                b = V[:, j]

                V[:, i] = c*a - s*b
                V[:, j] = s*a + c*b

    s = []
    for j in range(n):
        s.append(LA.norm(A[:, j].flatten()
                         , 2))

    A, V, s = ordenar(A, V, s)

    U = np.matrix(np.zeros((m, n)),
                  dtype=dtype)
    for j in range(n):
        U[:, j] = A[:, j]/s[j]

    return U, V, s
```

1.1.2 Funcion de solucion

Escriba una función que calcule una aproximación de la solución del sistema $Ax = b$ usando la descomposición en valores singulares de la siguiente manera. La función recibe las matrices U y V , el arreglo s con los valores singulares (calculados con la funcion del inciso anterior), su tamaño n (por matrices cuadradas) y un índice k. La función debe devolver el vector

$$x = \sum_{i=1}^{k} \frac{u_i^T b}{s_i} v_i$$

donde u_i, v_i son los i-esimas columnas de las matrices U, V (correpondientemente).

Nota Dado que la expresion requiere del vector b, que es el vector de terminos dependientes del sistema, este tambien sera pasado a la funcion a pesar de que no se pido en el enunciado. Hay que tener cuidado con sus dimensiones.

[35]:
```
def aproximacion_solucion(U: np.matrix, V: np.matrix, s: list,
                          b: np.matrix, n: int, k: int):
    return sum([ ((b*U[:,i].transpose())
                  /s[i])*V[:, i]
                for i in range(k)])
```

1.1.3 Interfaz 1

Escriba un programa que reciba desde la línea de comandos el nombre de un archivo que contiene una matriz.

Lea el archivo para crear la matriz A y use la función del primer inciso para calcular su descomposición en valores singulares tomando $\tau = \sqrt{\epsilon_m}$

Imprima la siguiente información: 1. Dimensiones m, n 2. El valor del error de la ortogonalidad de U, $||I - U^T U||$ 3. El valor del error de la ortogonalidad de V, $||I - V^T V||$ 4. Crear la matriz S con s en su diagonal e imprimir $||A - USV^T||$. 5. El numero de condicion de la matriz $k_2 = \frac{s_1}{s_n}$

Puede eligir la norma matricial para calcular los errores.

[5]:
```
def load_matrix(file_name: str,
                path = "datos/", ext=".npy",
                dtype=np.float64) -> np.matrix:
    """ Funcion para cargar una matriz de un archivo. """
    if ext == ".npy":
        return np.matrix(np.load(path+file_name+ext,
                                 allow_pickle=True),
                         dtype=dtype)

    else:
        """ Sin fomato especifico, esperamoe esta en texto
        con condificacion estandar 'utf-8' e iran siendo
        ingresados como fuera de esperarse.
        """
        return np.matrix(np.loadtxt(path+file_name+ext),
                         dtype=dtype)
```

[6]:
```
def proceso( file_name: str, path = "datos/", ext=".npy", show=True):
    lon = 100
    dtype = np.float64

    # Cargar matriz
    A = load_matrix(file_name, path, ext,
                    dtype=dtype)
    szA = A.shape
    # Descomposicion
    U, V, s = Algoritmo1(A, szA[0], szA[1])

    I = np.identity(szA[1])
```

```
    # Errores ortogonalidad
    errU = LA.norm( I - U.transpose()*U)
    errV = LA.norm( I - V.transpose()*V)

    # Error de la factorizacion
    S = np.diag(s)
    errS = LA.norm( A - U*S*V.transpose())

    # Condicion de matrix
    k2 = s[1]/s[-1]

    if show:
        print('-'*lon)
        print(f"Dimensiones\n\t m={szA[0]}, n={szA[1]}")
        print(f"Errores ortogonaldad")
        print(f"\t U -> {errU}")
        print(f"\t V -> {errV}")
        print(f"Error factorizacion")
        print(f"\t s -> {s}")
        print(f"\t S -> {errS}")
        print(f"Condicion de matriz\n\t {k2}")
        print('-'*lon)

    return U, V, s
```

[7]:
```
if NOTEBOOK:
    U, V, s = proceso('matA5')
    S = np.diag(s)
    A = U*S*V.transpose()
    print(A)
    print(svdvals(A))
```

```
--------------------------------------------------------------------------------
--------------------
Dimensiones
         m=5, n=5
Errores ortogonaldad
         U -> 2.122411848634482e-08
         V -> 1.2914954894440789
Error factorizacion
         s -> [5.668182669132584, 5.539783461443808, 3.580430538421286,
0.9564239531731384, 0.45613123053785987]
         S -> 6.20438087415781
Condicion de matriz
         12.14515273359252
--------------------------------------------------------------------------------
--------------------
```

```
[[ 0.24507182 -0.90270864  1.0648785   0.83548115  0.41287544]
 [ 0.71778638 -1.33941771  0.55354223  0.4567546  -1.03591622]
 [-1.74882695 -1.18155014  1.0253082  -2.96266235  1.00376056]
 [ 1.45478487  0.01716455 -1.22855353 -2.85284197 -0.34251647]
 [ 0.12012564  0.00428951 -0.54711507 -1.57187169  0.50304521]]
[4.62546674 3.18677891 2.15964637 0.85253219 0.20911506]
```

Aqui note que no esta funcionando mi Algoritmo 1, creo que el problema esta en el calculo de la norma. Cuando este resultado es comparado contra las funciones implementadas en nmumpy, los valores difieren por mucho.

1.1.4 Pruebas

Genere el vector x que tiene sus entradas iguales a 1 y calcule el vector $b = Ax$.

Use la función del inciso 2 para calcular la solución x_1 del sistema de ecuaciones usando $k = n$. Imprima las primeras y últimas entradas del vector x_1 y reporte el error $||x-x_1||$ y $||Ax_1-b||$.

[44]:
```
if NOTEBOOK:
    n = 5
    file_name = 'matA' + str(n)

    A = load_matrix(file_name)
    U, V, s = proceso(file_name)

    x = np.matrix([1 for n in range(n)]).transpose()
    b = A*x

    print(b)
    sol = aproximacion_solucion(U, V, s, b, n, n)
    print(sol)
    print(LA.norm(x - sol))
    print(LA.norm(A*sol - b))

    eps = np.finfo(float).eps
    ks = ([k for k in range(len(s)-1)
          if s[k] > eps**(2/3) and s[k+1] <= eps**(2/3)])
    sol2 = aproximacion_solucion(U, V, s, b, n, ks)
    print(sol2)
    print(LA.norm(x - sol))
    print(LA.norm(A*sol - b))
```

```
--------------------------------------------------------------------------------
--------------------
Dimensiones
         m=5, n=5
Errores ortogonaldad
         U -> 2.122411848634482e-08
         V -> 1.2914954894440789
```

```
Error factorizacion
         s -> [5.668182669132584, 5.539783461443808, 3.580430538421286,
0.9564239531731384, 0.45613123053785987]
         S -> 6.20438087415781
Condicion de matriz
         12.14515273359252
--------------------------------------------------------------------------------
--------------------
[[ 1.9834636 ]
 [-3.8776106 ]
 [-3.999597  ]
 [-6.15011408]
 [-2.49258567]]
[[ 1.01183074]
 [-1.97809812]
 [-2.04032744]
 [-3.13737771]
 [-1.27155084]]
6.355362840222379
40.38798399875771
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-15962d939156> in <module>
     18     ks = ([k for k in range(len(s)-1)
     19           if s[k] > eps**(2/3) and s[k+1] <= eps**(2/3)])
---> 20     sol2 = aproximacion_solucion(U, V, s, b, n, ks)
     21     print(sol2)
     22     print(LA.norm(x - sol))

<ipython-input-35-098f5df2ac65> in aproximacion_solucion(U, V, s, b, n, k)
      3     return sum([ ((b*U[:,i].transpose())
      4                   /s[i])*V[:, i]
----> 5                 for i in range(k)])


TypeError: 'list' object cannot be interpreted as an integer
```

[ ]: