Modelación Numérica I

Iván Vega Gutiérrez

[1]Centro de Investigación en Matemáticas A.C.
Unidad Aguascalientes
E-mail: ivan.vega@cimat.mx

I. Ejercicio 7.2

I.1.

Sean $A \in \mathcal{M}_{6,4}(\mathbb{R})$ y $b \in \mathbb{R}^6$ definidos de la siguiente manera.

$$A = \begin{pmatrix} 1 & 4 & 1 & 0 \\ 2 & 5 & 0 & 1 \\ 3 & 6 & 0 & 0 \\ -1 & 0 & -1 & -4 \\ 0 & -1 & -2 & -5 \\ 0 & 0 & -3 & -6 \end{pmatrix} \qquad y \qquad b_0 = \begin{pmatrix} 2 \\ 4 \\ 3 \\ -2 \\ -4 \\ -3 \end{pmatrix}$$

El objetivo de este ejercicio es hallar una solución x_0 para el problema de mínimos cuadrados con respecto a la matriz A y el vector b_0, es decir que x_0 satisfaga

$$||b_0 - Ax_0|| = \min_{y \in \mathbb{R}^4} ||b - Ay||$$

Lo que es equivalente a resolver la ecuación normal. Dado que estamos trabajando con matrices reales, tenemos que $A^* = A^T$

$$A^T A x = A^T b_0$$

Por lo tanto, nuestro problema se reduce a resolver un sistema lineal. Notemos que $A^T A$ es una matriz cuadrada de orden 4, por lo tanto podemos hallar una solución aplicando algún método directo, en particular el método de Cholesky, el cual ha sido el método más eficiente computacionalmente (hasta el momento) apliquemos el método para obtener nuestra solución x_0. Los códigos (1), (2), (3), (4) y (5) son necesarios para poder implementar el método de Cholesky al problema asociado de mínimos cuadrados.

Código 1. Sustitución hacia adelante

```
import numpy as np
def ForwSub(A,b):
    """
    Esta función calcula la solución de un sistema Ax=b,
    mediante sustitución hacia adelante, donde A es una
    matriz triangular inferior.
```

```
    Argumentos:
    A (matriz): Matriz triangular inferior.
    b (vector): Vctor de términos independientes.
    """
    #Creamos el vector que contendrá los valores de la solución.
    n = A.shape[0]
    x = np.empty(shape=n)
    #Hallamos el valor de cada x_i
    for i in range(n):
        suma = 0
        #Este ciclo utiliza el valor de x_i, para hallar x_i+1
        for j in range(i):
            suma += A[i][j]*x[j]
        x[i] = (b[i] - suma)/ A[i][i]
    return x
```

Código 2. Sustitución hacia atrás

```
import numpy as np
def BackSub(A,b):
    """
    Esta función calcula la solución de un sistema Ax=b,
    mediante sustitución hacia atrás, donde A es una matriz
    triangular superior.

    Argumentos:
    A (matriz): Matriz triangular superior.
    b (vector): Vector de términos independientes.
    """
    #Creamos el vector que contendrá los valores de la solución.
    n =  A.shape[0]
    x = np.empty(shape=n)
    #Hallamos el valor de la variable x_i (empezando por x_n)
    for i in range(n-1, -1, -1):
        suma = 0
        #A partir del valor x_i hallamos el valor de x_i-1
        for j in range(n-1, i, -1):
            suma += A[i][j]*x[j]
        x[i] = (b[i] - suma) / A[i][i]
    return x
```

Código 3. Factorización de Cholesky

```
import numpy as np
import math

def Cholesky(A):
    """
    Esta función regresa una matriz triangular inferior B
    tal que la matriz A se puede expresar como el producto
    de B y la transpuesta de B (factorización de Cholesky)

    Argumentos:
```

```
    A (matriz) : A es una matriz simétrica

    Excepciones:
    División entre cero:     si A[j][j] = 0
    Raiz cuadrada de un numero negativo:     si A[J][J] - suma < 0
    Resultado erroneo:  si A no es simetrica
    """
    #Primero, creamos la matriz B
    n = A.shape[0]
    B = np.zeros((n,n))
    #Verificamos si la matriz es simetrica
    if not np.array_equal(A, np.transpose(A)):
        print("Error de ejecución: La matriz no es simétrica")
        return
    #Hallamos los valores de la j-esima columna de B
    for j in range(n):
        suma = 0
        #El siguiente ciclo es para hallar los valores de la diagonal
        for k in range(j):
            suma += B[j][k]*B[j][k]
        #Verificamos que no se calculen raíces cuadradas negativas
        if A[j][j] - suma < 0:
            print("Error de ejecución: Raíz cuadrada negativa")
            return
        else:
            #Asignamos el valor de B en la diagonal
            B[j][j] = math.sqrt(A[j][j] - suma)
        #Hallamos los valores de la iésima fila de B
        for i in range(j+1,n):
            suma = 0
            for k in range(j):
                suma += B[j][k]*B[i][k]
            #Verificamso que no existan divisiones entre cero
            if B[j][j] == 0:
                print("Error de ejecución: División entre cero")
                return
            else:
                #Asigamos el valor de B en la i-esima fila
                B[i][j] = (A[i][j] - suma)/B[j][j]
    return B
```

Código 4. Solución al sistema por Cholesky

```
import numpy as np
def SolCholesky(A,b):
    """
    Esta función da la solución al sistema Ax=b mediante la
    factorización de Colesky.

    Argumentos:
    A: Es una matriz cuadrada simétrica
    b: Vector de valores independientes

    Salida
```

```
    x: Vector que contiene la solución del sistema
    """
    # Hallamaos la factorización de A = BBt
    B = Cholesky(A)
    Bt = np.transpose(B)
    # Se resuelven los sistemas By = b  y  Btx=y
    x = BackSub(Bt, ForwSub(B,b))
    return x
```

Código 5. Solución de mínimos cuadrados por Cholesky

```
import numpy as np
def MinimosCuadradosCholesky(A, b):
    """
    Esta función devuelve la solución al problema de mínimos
    cuadrados, es decir, encuentra un x tal que Ax sea lo más
    cercano a b.
    """
    # Resolvemos la ecuación normal At*A(x) = At*b utilizando la
    # factorización de Cholesky
    At = np.transpose(A)
    A = np.dot(At, A)
    b = np.dot(At, b)
    x = SolCholesky(A,b)
    return x
```

Ahora definamos un vector $b \in \mathbb{R}^6$ como una pequeña variación de b_0 definido por

$$b = b_0 + 0{,}01r,$$

donde $r \in \mathbb{R}^6$ es un vector con elementos aleatorios. Calculemos la solución del problema de mínimos cuadrados con respecto a la matriz A y el vector b. Además, definamos y calculemos los errores relativos

$$\frac{||x - x_0||_2}{||x_0||_2} \qquad y \qquad \frac{||b - b_0||_2}{||b_0||_2}.$$

Por otro lado, el coeficiente de amplificación C_b es una medida sobre el error relativo de la solución real con respecto al error relativo.

$$C_b = ||A^\dagger||_2 \frac{||b_0||}{||x_0||}$$

Donde $A^\dagger$ es la matriz pseudo-inversa de Moore-Penrose de A, y se define como

$$A^\dagger = (A^T A)^{-1} A^T$$

En general, si se cumple que Ker $A = \{0\}$, se satisface que

$$\frac{||x - x_0||_2}{||x_0||_2} \leq C_b \frac{||b - b_0||_2}{||b_0||_2}.$$

El código (6) proporciona el coeficiente de amplificación de la solución.

Código 6. Constante de amplificación

```
import numpy as np
def Cb(A,b,x):
    """
    Esta función calcula la constante de amplificación de la solución
    x con respecto al error relativo de b
    """
    #Definimos la matriz pseudo-inversa de Moore Penrose (At*A)^-1*At
    At =  np.transpose(A)
    MP = np.dot(np.linalg.inv(np.dot(At, A)), At)
    #Calculamos
    cb= np.linalg.norm(MP)*((np.linalg.norm(b))/(np.linalg.norm(x)))
    return cb
```

I.2.

Para este ejercicio se pretende hacer los mismo cálculos que para el anterior: calcular la solución de mínimos cuadrados, calcular la solución tomando en cuenta la variación en el vector b, calcular los errores relativos y el coeficiente de amplificación, pero tomando en cuenta el siguiente vector.

$$b = \begin{pmatrix} 3 \\ 0 \\ -2 \\ -3 \\ 0 \\ 2 \end{pmatrix}$$

I.3.

Ahora, tratemos de entender mejor que sucede con el coeficiente de amplificación con variaciones, para ello consideremos que i son pasos de tamaño $\frac{1}{100}$ calculemos el coeficiente de amplificación $C_b(i)$ asociado al vector $b_2 = ib_0 + (1-i)b_1$

I.4. Solución

A continuación se muestra el código (7) que se implementó para hallar los resultados que se requerían.

Código 7. Ejercicio 7.2

```
import numpy as np
import matplotlib.pyplot as plt

#Problema 7.1

# Inciso 1

#Definimos la matriz A y el vector b0
A = np.arange(1,7).reshape((3,2), order= 'F')
I = np.array([ [ 1, 0], [0, 1], [0, 0]])
A = np.concatenate((np.concatenate((A,I), axis = 1),
                    np.concatenate( (-1*I, -1*A), axis=1)), axis=0)
```

```
b0 = np.array([2, 4, 3, -2, -4, -3])
# Hallamos la solución x asociada al problema de minimos cuadrados con
# respecto a la matriz A y el vector b
x0 = MinimosCuadradosCholesky(A,b0)

# Definimos al vector b como una pequeña variación de b
e = 0.01
b = b0 + e*np.random.rand(len(b0))
# Hallamos la solución x asociada al problema de minimos cuadrados con
# respecto a la matriz A y el vector b
x = MinimosCuadradosCholesky(A,b)

# Calculamos los errores relativos
error1 = np.linalg.norm(x-x0)/np.linalg.norm(x0)
error2 = np.linalg.norm(b-b0)/np.linalg.norm(b0)

# Calculamos el coeficiente de amplificación
cb = Cb(A,b,x)

# Impresión de resultados
print("Inciso 1. \n")
print("Las soluciones al problema de mínimos cuadrados son:")
print("x0 = {}".format(x0))
print("x = {}".format(x))
print("Los errores relativos son: ")
print(error1)
print(error2)
print("El coeficiente de amplificación es: ")
print("Cb = {}".format(cb))

# Inciso 2

# Definimos el vector b1
b1 = np.array([3, 0, -2, -3, 0, 2])
# Hallamos la solución x asociada al problema de minimos cuadrados con
# respecto a la matriz A y el vector b1
x1 = MinimosCuadradosCholesky(A,b1)

# Definimos al vector b como una pequeña variación de b
e = 0.01
b = b1 + e*np.random.rand(len(b1))
# Hallamos la solución x asociada al problema de minimos cuadrados con
# respecto a la matriz A y el vector b
x = MinimosCuadradosCholesky(A,b)

# Calculamos los errores relativos
error1 = np.linalg.norm(x-x1)/np.linalg.norm(x1)
error2 = np.linalg.norm(b-b1)/np.linalg.norm(b1)

# Calculamos el coeficiente de amplificación
cb = Cb(A,b,x)

# Impresión de resultados
print("\n \n")
```

```
print("Incico 2. \n")
print("Las soluciones al problema de mínimos cuadrados son:")
print("x1 = {}".format(x1))
print("x = {}".format(x))
print("Los errores relativos son: ")
print(error1)
print(error2)
print("El coeficiente de amplificación es: ")
print("Cb = {}".format(cb))

# Inciso 3.

CB = []
y = []
for i in range(1, 101):
    k = i/100
    y.append(i)
    b2 = k*b0 + (1-k)*b1
    x2 = MinimosCuadradosCholesky(A,b2)
    CB.append(Cb(A,b2,x2))
print("\n \n")
print("Inciso 3. \n")
plt.plot(y,CB)
plt.title("Coeficiente de amplificación")
plt.xlabel("i")
plt.ylabel("Cb(i)")
plt.show()
```

Los resultados al implementar el código son los siguientes.

```
Inciso 1.

Las soluciones al problema de mínimos cuadrados son:
x0 = [-1.  1. -1.  1.]
x = [-0.99741237  0.99953825 -0.99765213  0.99792199]
Los errores relativos son:
0.0020457067247021043
0.0014261968166998742
El coeficiente de amplificación es:
Cb = 5.918463946391684

Incico 2.

Las soluciones al problema de mínimos cuadrados son:
x1 = [0. 0. 0. 0.]
x = [ 3.23060490e-03 -7.08529941e-05  3.15449288e-03 -2.34606474e-03]
Los errores relativos son:
inf
0.0029344625768198255
El coeficiente de amplificación es:
Cb = 1556.3376052877222
```

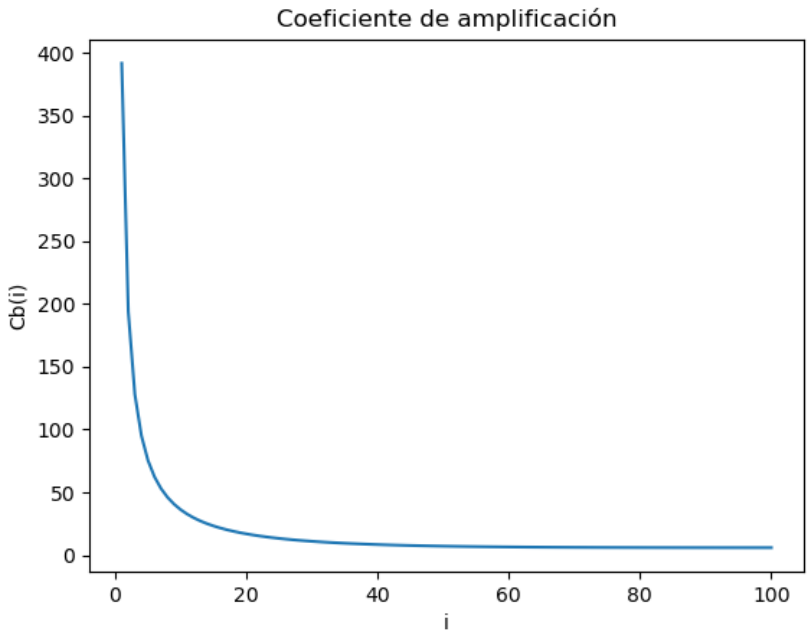

Figura 1. Coficiente de amplificación

De lo anterior, podemos observar que en el inciso 1, la solución x_0 al sistema es exacta, y al hacer la pequeña variación en b la solución x no cambia mucho, se mantiene una continuidad en las soluciones. Sin embargo, para el inciso 2, x_1 no es una buena solución al sistema, y hay una diferencia considerable entre las soluciones x_1 y x, esto se puede relacionar directamente con el coeficiente de amplificación, mientras que en el inciso 1 es pequeño, en el inciso 2 es muy grande.

Por otro lado, en la figura (1) se observa que a medida que que las iteraciones van aumentando, el coeficiente disminuye.

II. Ejercicio 7.4

Para este ejercicio, definamos la matriz A como una matriz de 300 filas y 100 columnas de rango 100 y el vector b como un vector unitario de 300 elementos aleatorios, el código (8) crea la matriz A.

II.1.

Como hemos visto, el problema de mínimos cuadrados es equivalente a resolver el sistema

$$A^T Ax = A^T b$$

El objetivo de este ejercicio es comparar computacionalmente los siguientes métodos:

- Método de Cholesky
- Método de Factorización QR
- Método SVD

II.2.

Para este ejercicio, definamos la matriz A y el vector b como:

$$A = P \begin{pmatrix} 0{,}00001 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{0{,}00001} \end{pmatrix} P^{-1} \qquad \text{donde} \qquad P = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

y

$$b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

El objetivo es comparar las soluciones del problema utilizando minimos cuadrados mediante la factorización de Cholesky y QR. El código (9) contiene las comparaciones requeridas.

Código 8. Matriz real de mxn y rango r

```
def MatRank(m,n,r):
    """
    Esta función regresa una matriz real de tamaño mxn y rango r

    Argumentos:
    m(entero positivo) el número de filas que queremos
    n(entero positivo) el número de columnas que queremos
    r(entero positivo) el rango de la matriz A que queremos

    Salida:
    A (matriz de mxn) Matriz mxn de rango r
    """
    # Hallamos el valor mas pequeño entre m y n
    if m <= n:
        minimo = m
        maximo = n
    else:
        minimo = n
        maximo = m
    # Verificamos que sea posible contruir la matriz A, es decir
    # que el rango no sea mayor a min(n,m)
    if r > minimo:
        print("Error: El rango no puede ser mas grande que {}".format(minimo))
    #Construimos la matriz deseada
    else:
        # Creamos una matriz de min(n,m)x min(n,m) con valores aleatorios
        A = np.random.rand(minimo,minimo)
        A = A + np.linalg.norm(A,np.inf)
        # Rellenamos la matriz A con los datos faltantes para que
        # tengamos una matriz de tamaño m por n.
        if m >= n:
            # Rellenamos con unos las filas faltantes
            A = np.concatenate((A, np.ones((maximo-minimo,minimo))),
            axis = 0)
            # Hacemos que los vectores(columna) r,r+1,. . .,minimo-1 sean
            # linealmente dependientes del vector(columna) r-1,
            # esto para que el rango de A sea r
```

```
            for k in range(r, minimo):
                A[:, k] = np.random.rand()*A[:, r-1]
        else:
            # Rellanamos con unos las columnas faltantes
            A = np.concatenate((A, np.ones((minimo, maximo-minimo))),
            axis = 1)
            # Hacemos que los vectores(fila) r, r+1,. . ., minimo-1 sean
            # linealmente dependientes del vector(fila) r-1, esto para que
            # el rango de A sea r
            for k in range(r, minimo):
                A[k, :] = np.random.rand()*A[r-1, :]
    return A
```

El código (9) se muestra a continuación

Código 9. Ejercicio 7.4

```
from timeit import default_timer

# Ejercicio 7.4

# Inicso 1.

# Definimos una matriz A de 300 filas y 100 columnas con rango 100
# y un b un vector aleatorio de 300 elementos
A = MatRank(300, 100, 100)
b = np.random.rand(300,1)

# Inciso a)

# Solución del problema de mínimos cuadrados tomando la matriz A
# y el vector b mediante factorización de Cholesky
inicio_chol = default_timer()
x1 = MinimosCuadradosCholesky(A,b)
fin_chol = default_timer()
tiempo_chol = fin_chol - inicio_chol
distmin_chol = np.linalg.norm(np.dot(A,x1) -b)

# Inciso b)

# Solución del problema de mínimos cuadrados tomando la matriz A
# y el vector b mediante factorización QR (método Householder)
inicio_qr = default_timer()
x2 = MinimosCuadradosHouseholder(A,b)
fin_qr = default_timer()
tiempo_qr = fin_qr - inicio_qr
distmin_qr = np.linalg.norm(np.dot(A,x2) - b)

# Inciso c)

# Solución del problema de mínimos cuadrados tomando la matriz A
# y el vector b mediante el método SVD
inicio_svd = default_timer()
x3 = MinimosCuadradosSVD(A,b)
fin_svd = default_timer()
```

```
tiempo_svd = fin_svd - inicio_svd
distmin_svd = np.linalg.norm(np.dot(A,x3) - b)

# Impresión de resultados

print("Iniciso 1. \n")
# Cholesky
print("Inciso (a)")
print("Para el método de Cholesky se obtuvieron los siguientes resultados:")
print("Tiempo de ejecución: {}".format(tiempo_chol))
print("Distancia entre la solución y el vector b: {}".format(distmin_chol))
print("\n")
# QR
print("Inciso (b)")
print("Para el método QR se obtuvieron los siguientes resultados:")
print("Tiempo de ejecución {}".format(tiempo_qr))
print("Distancia entre la solución y el vector b: {}".format(distmin_qr))
print("\n")
# SVD
print("Inciso (c)")
print("Para el método SVD se obtuvieron los siguientes resultados:")
print("Tiempo de ejecución {}".format(tiempo_svd))
print("Distancia entre la solución y el vector b: {}".format(distmin_svd))
print("\n")

# Inciso 2)

# Inicializamos los datos
e = 1e-5
P = np.array([[1, 1, 0],
              [0, 1, -1],
              [1, 0, -1]
              ])
D = np.array([[e, 0, 0],
              [0, 1, 0],
              [0, 0, 1/e]
              ])
A = np.dot(np.dot(P, D), np.linalg.inv(P))
b = [1, 1, 1]

# Resolvemos por el método de Cholesky
inicio_chol = default_timer()
x1 = MinimosCuadradosCholesky(A,b)
fin_chol = default_timer()
tiempo_chol = fin_chol - inicio_chol
distmin_chol = np.linalg.norm(np.dot(A,x1) -b)

# Resolvermos por el método QR
inicio_qr = default_timer()
x2 = MinimosCuadradosHouseholder(A,b)
fin_qr = default_timer()
tiempo_qr = fin_qr - inicio_qr
distmin_qr = np.linalg.norm(np.dot(A,x2) - b)
```

```
# Impresión de resultados
print("Inciso 2.\n")

# Cholesky
print("Cholesky")
print("Para el método de Cholesky se obtuvieron los siguientes resultados:")
print("Tiempo de ejecución: {}".format(tiempo_chol))
print("Distancia entre la solución y el vector b: {}".format(distmin_chol))
print("Solución x = {}".format(x1))
print("\n")
# QR
print("QR")
print("Para el método QR se obtuvieron los siguientes resultados:")
print("Tiempo de ejecución {}".format(tiempo_qr))
print("Distancia entre la solución y el vector b: {}".format(distmin_qr))
print("Solución x = {}".format(x2))
```

Los resultados que se obtineen son los siguientes

```
Iniciso 1.

Inciso (a)
Para el método de Cholesky se obtuvieron los siguientes resultados:
Tiempo de ejecución: 0.012876221000624355
Distancia entre la solución y el vector b: 151.43632597838914


Inciso (b)
Para el método QR se obtuvieron los siguientes resultados:
Tiempo de ejecución 0.009045227001479361
Distancia entre la solución y el vector b: 151.436327014116


Inciso (c)
Para el método SVD se obtuvieron los siguientes resultados:
Tiempo de ejecución 0.007371800998953404
Distancia entre la solución y el vector b: 151.43632701409692


Inciso 2.

Cholesky
Para el método de Cholesky se obtuvieron los siguientes resultados:
Tiempo de ejecución: 0.0001758739999786485
Distancia entre la solución y el vector b: 0.577303074582103
Solución x = [4.75382214 0.66665431 4.08718117]

QR
Para el método QR se obtuvieron los siguientes resultados:
Tiempo de ejecución 0.0035421769971435424
Distancia entre la solución y el vector b: 7.66312669381364e-07
```

```
Solución x = [5.00005337e+04  5.00005263e-01  5.00000337e+04]
```

A modo de conclusión del inciso 1, podemos decir que los tres métodos convergen a la solución y los resultados son casi idénticos, sin embargo el método SVD es el más rápido, de ahí le sigue el método QR y por último el método Cholesky.

Por otro lado, con respecto al inciso 2, podemos observar que el método de Cholesky es más rápido, sin embargo el método QR da una mejor solución.

III. Códigos

Código 10. Matriz de Householder

```python
def House(v):
    """
    Esta función regresa la matriz de Householder
    H de n por n asociada al vector v de dimensión n
    """
    n = len(v)
    norm = np.linalg.norm(v)
    I = np.identity(n)
    vt = np.transpose(v)
    H = I - (2/norm**2)*(np.dot(vt,v))
    return H
```

Código 11. Factorización de Householder

```python
def Householder(A):
    """
    Esta función realiza la factorización A=QR mediante el
    algoritmo de Householder.  Donde Q es una matriz ortogonal
    y R es una matriz triangular superior cuya diagonal son enteros
    positivos

    Argumentos:
    A(matriz de mxn): Matriz a factorizar con mas filas que columnas

    Salida
    (Q,R): Tupla que contiene las matrices Q y R respectivamente
    """
    # Hallamos el número de filas m y columnas n
    (m, n) = np.shape(A)
    I = np.identity(n)
    # Paso 1
    # Verificamos si en la primera columna a1 existen ceros por
    # debajo del primer elemento
    if  np.array_equiv(A[1: , 0], np.zeros((m-1, 1))):
        Hk = I
    else:
        a1 = A[:, 0]
        norm1 = np.linalg.norm(a1)
        e1 = np.zeros((m,1))
        e1[0] = norm1
```

```
        Hk = House(a1 + e1)
    #  Inicializamos las matrices Ak y Q
    Ak = A
    Q = Hk
    Ak = np.dot(Hk, Ak)
    #Paso k
    for k in range(1, n):
        Ceros = np.zeros((m-k-1, 1))
        # Verificamos si en la columna ak solo hay ceros por debajo
        # del elemento akk de ser cierto la la k-ésima matriz de
        # Householder será la matriz identidad
        if np.array_equiv(A[k+1:, k], Ceros):
            Hk = I
        # Si no se cumple lo anterior, construimos la k-ésima matriz
        # de Householder
        else:
            # ak es el vector de tamaño (m-(k-1)) de las ultimas m+1-k
            # entradas del vector k-ésimo de Ak
            ak =  A[k:,k]
            # Construimos la k-ésima matriz de HouseHolder Hk la cual
            # esta compuesta por cuatro submatrices
            #Matriz superior izquierda
            Ik = np.identity(k)
            # Matriz superior derecha
            CerosUp = np.zeros((k ,m-k))
            # Matriz inferior izquierda
            CerosDown = np.zeros((m-k,k))
            # Hallamos la matriz Householder Hw =H(ak + norm(ak)ek) que
            # será la matriz inferior derecha de Hk
            ek = np.zeros((m-k,1))
            ek[0] = np.linalg.norm(ak)
            Hw = House(ak + ek)
            # Concatenamos las cuatro matrices para formar Hk
            HkU = np.concatenate((Ik, CerosUp), axis = 1)
            HkD = np.concatenate((CerosDown, Hw), axis = 1)
            Hk = np.concatenate((HkU, HkD), axis = 0)
        # Actualizamos la matriz Ak que eventualmente será R
        Ak = np.dot(Hk, Ak)
        # Actualizamos la matriz Q donde Q = H1 H2 ... Hm
        Q = np.dot( Q, Hk)
    R = Ak
    return Q, R
```

Código 12. Mínimos cuadrados por Householder

```
def MinimosCuadradosHouseholder(A, b):
    """
    Esta función devuelve la solución al problema de mínimos
    cuadrados, es decir, encuentra un x tal que Ax sea lo más
    cercano a b utilizando el método de Householder
    """
    # Hallamos la factorización A = QR
    # Q,R = Householder(A)
    Q, R = np.linalg.qr(A
```

```
    m,n = np.shape(A)
    # Como Q es ortogonal Q^-1 = Q^t
    Qinv = np.transpose(Q)
    b = np.dot(Qinv, b)
    b = b[0:n]
    R = R[0:n,:]
    # Resolvemos el sistema Rx = Q^-1 b
    x = BackSub(R, b)
    return x
```

Código 13. Facotrización por SVD

```
import math

def FactSVD(A):
    """
    Esta función calcula las matrices U, D y V tales que
    A = UDV^T mediante la descomposición de valores singulares

    Argumentos:
    A: Matriz de m por n que se desea factorizar

    Salida
    (U, D, V): Matrices de la factorizacón
    """
    m, n = np.shape(A)
    B = np.dot(np.transpose(A), A)
    # Hallamos los valores y vectores propios de B
    valores, V = np.linalg.eig(B)
    # Creamos la matriz D
    D = np.zeros((m, n))
    k = np.minimum(m, n) - np.isclose(valores, 0).sum()
    # Rellenamos la matriz D
    for i in range(k):
        D[i, i] = np.sqrt(valores[i])
    # Creamos la matriz U
    U = np.zeros((m, n))
    for i in range(k):
        U[:, i] = 1/(D[i, i] * np.dot(A, V[:, i]))
    return (U, D, V)
```

Código 14. Mínimos cuadrados por SVD

```
def MinimosCuadradosSVD(A, b):
    """
    Esta función devuelve la solución al problema de mínimos
    cuadrados, es decir, encuentra un x tal que Ax sea lo más
    cercano a b mediante el método SVD
    """
    A = np.array(A)
    b = np.array(b)
    U,s,V = np.linalg.svd(A)
    n, m = np.shape(A)
    r = 1
```

```
    while r < m and abs(s[r]) >= 1e-14:
        r = r+1
    y = U.transpose().dot(b)
    for i in range(r):
        y[i] = y[i]/s[i]
    y = y.reshape((n,))
    y = np.concatenate((y[0:r],np.zeros((m-r,))),axis=0)
    x = V.transpose().dot(y)
    return(x)
```