MA-0501 Análisis Numérico I
TAREA 3

Luis Fernando Amey Apuy C20470
Javier Hernández Navarro C13674

1. [MATLAB] Considere los siguientes tres algoritmos para obtener la factorización QR de una matriz A:

Data: Matriz $A \in \mathbb{R}^{m\times n}$
Result: Matriz unitaria $Q \in \mathbb{R}^{m\times n}$ y triangular superior $R \in \mathbb{R}^{n\times n}$ tales que $A = QR$
for $j = 1 : n$ **do**
  $\boldsymbol{v}_j = \boldsymbol{a}_j$;
  for $i = 1 : j-1$ **do**
    $r_{ij} = \boldsymbol{q}_i^T \boldsymbol{a}_j$;
    $\boldsymbol{v}_j = \boldsymbol{v}_j - r_{ij}\boldsymbol{q}_i$;
  end
  $r_{jj} = \|\boldsymbol{v}_j\|_2$;
  $\boldsymbol{q}_j = \boldsymbol{v}_j / r_{jj}$;
end

Algoritmo 1: Ortogonalización de Gram-Schmidt (inestable)

Data: Matriz $A \in \mathbb{R}^{m\times n}$
Result: Matriz unitaria $Q \in \mathbb{R}^{m\times n}$ y triangular superior $R \in \mathbb{R}^{n\times n}$ tales que $A = QR$
$V = A$;
for $i = 1 : n$ **do**
  $r_{ii} = \|\boldsymbol{v}_i\|_2$;
  $\boldsymbol{q}_i = \boldsymbol{v}_i / r_{ii}$;
  for $j = i+1 : n$ **do**
    $r_{ij} = \boldsymbol{q}_i^T \boldsymbol{v}_j$;
    $\boldsymbol{v}_j = \boldsymbol{v}_j - r_{ij}\boldsymbol{q}_i$;
  end
end

Algoritmo 2: Ortogonalización de Gram-Schmidt (estable)

Data: Matriz $A \in \mathbb{R}^{m \times n}$
Result: Matriz unitaria $Q \in \mathbb{R}^{m \times m}$ y triangular superior $R \in \mathbb{R}^{m \times n}$ tales que $A = QR$
$R = A$;
$M = I$;
for $j = 1 : n$ **do**
  $\boldsymbol{x} = R_{j:m,j}$ % vector de valores entre filas j y m de la columna j;
  $\boldsymbol{v}_j = \text{sign}(x_1)\|\boldsymbol{x}\|_2 \boldsymbol{e}_1 + \boldsymbol{x}$;
  $\boldsymbol{v}_j = \boldsymbol{v}_j / \|\boldsymbol{v}_j\|_2$;
  $R_{j:m,j:n} = R_{j:m,j:n} - 2\boldsymbol{v}_j \left(\boldsymbol{v}_j^T R_{j:m,j:n}\right)$;
  $M_{j:m,:} = M_{j:m,:} - 2\boldsymbol{v}_j \left(\boldsymbol{v}_j^T M_{j:m,:}\right)$;
end
$Q = M^T$;

Algoritmo 3: Triangularización de Householder

a) (5 puntos) Implemente una función `[Q,R] = qr1(A)` que implemente el Algoritmo 1.

```
function [Q, R] = qr1(A)
    %
    %   Función que calcula la factorización QR de una matriz A
    %   utilizando el método de Gram-Schmidt inestable.
    %
    %   Inputs
    %       A: Matriz de tamaño mxn
    %
    %   Outputs
    %       Q: Matriz unitaria de tamaño mxn
    %       R: Matriz triangular superior de tamaño nxn
    %

    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);

    for j = 1:n
        vj = A(:, j);
        for i = 1:j-1
            R(i, j) = Q(:, i)' * A(:, j);
            vj = vj - R(i, j) * Q(:, i);
        end
        R(j, j) = norm(vj);
        Q(:, j) = vj / R(j, j);
    end
end
```

b) (5 puntos) Implemente una función `[Q,R] = qr2(A)` que implemente el Algoritmo 2.

```
function [Q, R] = qr2(A)
    %
    %   Función que calcula la factorización QR de una matriz A
    %   utilizando el método de Gram-Schmidt estable.
    %
    %   Inputs
    %       A: Matriz de tamaño mxn
    %
    %   Outputs
    %       Q: Matriz unitaria de tamaño mxn
    %       R: Matriz triangular superior de tamaño nxn
    %

    [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);
    V = A;

    for i = 1:n
        R(i, i) = norm(V(:, i));
        Q(:, i) = V(:, i) / R(i, i);
        for j = i+1:n
            R(i, j) = Q(:, i)' * V(:, j);
            V(:, j) = V(:, j) - R(i, j) * Q(:, i);
        end
    end
end
```

c) (5 puntos) Implemente una función `[Q,R] = qr3(A)` que implemente el Algoritmo 3.

```
function [Q, R] = qr3(A)
    %
    %   Función que calcula la factorización QR de una matriz A
    %   utilizando la triangularización de Householder.
    %
    %   Inputs
    %       A: Matriz de tamaño mxn
    %
    %   Outputs
    %       Q: Matriz unitaria de tamaño mxm
    %       R: Matriz triangular superior de tamaño mxn
    %

    [m, n] = size(A);
    R = A;
    Q = eye(m);

    for j = 1:n
        x = R(j:m, j);
        e1 = zeros(length(x), 1);
        e1(1) = 1;
        vj = sign(x(1)) * norm(x) * e1 + x;
        vj = vj / norm(vj);
        R(j:m, j:n) = R(j:m, j:n) - 2 * vj * (vj' * R(j:m, j:n));
        Q(j:m, :) = Q(j:m, :) - 2 * vj * (vj' * Q(j:m, :));
    end

    Q = Q';
end
```

d) (4 puntos) Tome $m = n = 20$. Genere una matriz aleatoria `A = rand(m)`. Calcule las tres factorizaciones QR con las tres funciones `qr1`, `qr2`, `qr3`. Para cada caso, calcule $\|A - QR\|_2$, $\|QQ^T - I\|_2$. ¿Se cumple que $A = QR$? ¿Se cumple que Q es ortogonal? Determine si algún algoritmo es *mejor* que otro.

```
m = 20;
A_random = rand(m);

[Q1, R1] = qr1(A_random);
[Q2, R2] = qr2(A_random);
[Q3, R3] = qr3(A_random);

error_random = zeros(3,2);

error_random(1,1) = norm(A_random - Q1 * R1, 2);
error_random(1,2) = norm(Q1 * Q1' - eye(m), 2);

error_random(2,1) = norm(A_random - Q2 * R2, 2);
error_random(2,2) = norm(Q2 * Q2' - eye(m), 2);

error_random(3,1) = norm(A_random - Q3 * R3, 2);
error_random(3,2) = norm(Q3 * Q3' - eye(m), 2);

format long
disp(error_random)
```

```
1.0e-13 *

0.007520006994808   0.301055625853077
0.007612339524839   0.026872494571861
0.060987006351993   0.012013490420101
```

Al calcular $\|A - QR\|_2$, se puede observar que los valores obtenidos son muy pequeños (en el rango de 10^{-16}). Estos valores tan bajos sugieren que, en todos los casos, la factorización cumple que $A \approx QR$ con una precisión bastante alta, por lo que se puede afirmar que sí se cumple aproximadamente que $A = QR$.

De igual forma, al calcular $\|QQ^T - I\|_2$, se puede observar que los tres métodos producen matrices Q aproximadamente ortogonales con una precisión aproximada de 10^{-13}, por lo que de igual forma se puede decir que sí se cumple que Q es ortogonal.

Ahora bien, basándose en los errores observados, se puede deducir que:

- `qr3` **(Householder)** parece ser el mejor algoritmo en términos de precisión, tanto para la factorización $A = QR$ como para la ortogonalidad de Q. Este método muestra los errores más bajos en ambas métricas, proporcionando estabilidad en los cálculos.
- `qr2` **(Gram-Schmidt estable)** también es una opción adecuada, ya que ofrece buenos resultados de ortogonalidad y precisión, aunque no tan precisos como los de `qr3`.
- `qr1` **(Gram-Schmidt inestable)** es el que presenta mayores errores, especialmente en la ortogonalidad de Q, lo que sugiere que este método es menos fiable numéricamente.

De esta forma, se puede concluir que de los tres algoritmos implementados, el método de Householder (`qr3`) es el más recomendado debido a su menor error en ambas métricas y su mejor estabilidad numérica.

e) (4 puntos) Tome ahora $m = n = 20$. Defina `A = hilb(m)` como la matriz de Hilbert de tamaño $m \times m$. Repita el inciso anterior. Determine si algún algoritmo es *mejor* que otro. Compare con el algoritmo `qr` de `MATLAB`.

```
A_hilbert = hilb(m);

[Q1, R1] = qr1(A_hilbert);
[Q2, R2] = qr2(A_hilbert);
[Q3, R3] = qr3(A_hilbert);

error_hilbert = zeros(4,2);

error_hilbert(1,1)  = norm(A_hilbert - Q1 * R1, 2);
error_hilbert(1,2) = norm(Q1 * Q1' - eye(m), 2);

error_hilbert(1,2) = norm(A_hilbert - Q2 * R2, 2);
error_hilbert(2,2)  = norm(Q2 * Q2' - eye(m), 2);

error_hilbert(3,1) = norm(A_hilbert - Q3 * R3, 2);
error_hilbert(3,2) = norm(Q3 * Q3' - eye(m), 2);

% Comparación con la factorización QR de MATLAB
[Q4, R4] = qr(A_hilbert);

error_hilbert(4,1) = norm(A_hilbert - Q4 * R4, 2);
error_hilbert(4,2) = norm(Q4 * Q4' - eye(m), 2);

format long
disp(error_hilbert)
```

```
   0.000000000000000   0.000000000000000
                   0   0.998956338417536
   0.000000000000000   0.000000000000001
   0.000000000000000   0.000000000000001
```

Basándose en los errores observados con la matriz de Hilbert, se puede deducir que:

- **`qr3` (Householder)** y **`qr` de `MATLAB`** son los métodos más precisos, con errores extremadamente bajos tanto en la factorización como en la ortogonalidad de Q, lo que demuestra su estabilidad y precisión en matrices mal condicionadas.
- **`qr2` (Gram-Schmidt estable)** presenta un error notable en la ortogonalidad de Q, lo que sugiere que este método es menos adecuado para la matriz de Hilbert debido a su sensibilidad numérica.
- **`qr1` (Gram-Schmidt inestable)** muestra errores prácticamente nulos, pero no es confiable en matrices mal condicionadas, como la de Hilbert, debido a la inestabilidad del método.

2. [MATLAB] En esta pregunta se deben usar los comandos de factorización propios de MATLAB (`qr(A)`, `lu(A)`, `svd(A)`, `chol(A)`). Defina `m=12`, `x = ones(m,1)`, `A = hilb(m)`, `b = A*x`. Buscamos resolver el sistema de ecuaciones $Ax = b$ mediante estas factorizaciones. Note que la solución exacta es $x = [1, \ldots, 1]^T$ y denotamos por $\hat{x}$ la solución aproximada numéricamente.

   a) (2 puntos) Calcule el número de condición de A. ¿Qué nos indica este número? Sugerencia: utilice el comando `cond(A)`.

```
m = 12;
x = ones(m, 1); % solución exacta
A = hilb(m);
b = A * x;

fprintf("El número de condición es %d", cond(A))
```

```
El número de condición es 16223826740810282
```

   El número de condición de la matriz A es extremadamente alto (1.62×10^{16}), lo que indica que la matriz está muy mal condicionada. Esto significa que el sistema $Ax = b$ es muy sensible a pequeños errores en los datos de entrada, lo que podría generar grandes errores en la solución $\hat{x}$. En otras palabras, el sistema es numéricamente inestable, y pequeñas perturbaciones pueden afectar significativamente la precisión de la solución.

   b) (3 puntos) Calcule la factorización $PA = LU$ y resuelva el sistema[1]. Determine $\|x - \hat{x}\|_2$.

```
[L, U, P] = lu(A);
y = L \ (P * b);
x_LU = U \ y;

error_LU = norm(x - x_LU, 2);

fprintf("El error por LU es %d", error_LU)
```

```
El error por LU es 5.756644e-01
```

[1] Para resolver sistemas triangulares superiores o inferiores $L\boldsymbol{y} = \boldsymbol{b}$ o $U\boldsymbol{z} = \boldsymbol{b}$, basta ejecutar `y=L\b` o `z=U\b`, pues el comando \ chequea si las matrices son triangulares y realiza sustitución hacia adelante o atrás. De esta forma, realmente no se está invirtiendo ninguna matriz.

c) (3 puntos) Calcule la factorización $A = QR$ y resuelva el sistema. Determine $\|x - \hat{x}\|_2$.

```
[Q, R] = qr(A);
x_QR = R \ (Q' * b);

error_QR = norm(x - x_QR, 2);

fprintf("El error por QR es %d", error_QR)
```

```
El error por QR es 7.587727e-01
```

d) (3 puntos) Calcule la factorización $A = LL^T$ y resuelva el sistema. Determine $\|x - \hat{x}\|_2$.

```
L = chol(A, 'lower');
y = L \ b;
x_Chol = L' \ y;

error_Chol = norm(x - x_Chol, 2);

fprintf("El error por Cholesky es %d", error_Chol)
```

```
El error por Cholesky es 6.777753e-01
```

e) (3 puntos) Calcule la factorización $A = USV^T$ y resuelva el sistema. Determine $\|x - \hat{x}\|_2$.

```
[U, S, V] = svd(A);
x_SVD = V * (S \ (U' * b));
```

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  6.163774e-17.
```

```
error_SVD = norm(x - x_SVD, 2);

fprintf("El error por SVD es %d", error_SVD)
```

```
El error por SVD es 1.379475e+00
```

f) (1 punto) Determine si algún método brinda alguna solución aceptable en términos del error.

A pesar de que todos los métodos tienen errores significativos debido a lo mal condicionada que es la matriz A, el método LU es el que presenta el menor error, lo que lo convierte en la opción menos mala para resolver este sistema.

g) (3 puntos) Utilizando la factorización en valores singulares de A, obtenga la aproximación de rango $\nu = 9$ dada por $A_\nu = \sum_{i=1}^{\nu} \sigma_i u_i v_i^T$. Resuelva el sistema $A_\nu x = b$ y determine $\|x - \hat{x}\|_2$. Justifique sus resultados.

```
rango = 9;
A_nu_inv = V(:, 1:rango) * inv(S(1:rango, 1:rango)) * U(:, 1:rango)';
x_aprox = A_nu_inv * b;

error_approx = norm(x - x_aprox, 2);

fprintf("El error por rango 9 es %d", error_approx)
```

```
El error por rango 9 es 6.547244e-06
```

Este error es considerablemente más pequeño que los errores obtenidos con los métodos anteriores, lo que indica que la aproximación de rango 9 proporciona una solución mucho más precisa. Esto se debe a que la factorización en valores singulares (SVD) permite captar las componentes principales de la matriz A, eliminando las dimensiones menos importantes (las que corresponden a los valores singulares más pequeños), lo que mejora la estabilidad y precisión de la solución.

3. Para graficar una función en dos variables $z = f(x, y)$, se puede generar una malla (matriz) de puntos `xx,yy` en las cuales se evalúa f. De esta forma, `zz = f(xx,yy)` es una matriz del mismo tamaño de `xx` y `yy`.

   a) (2 puntos) [`MATLAB`] Defina los vectores

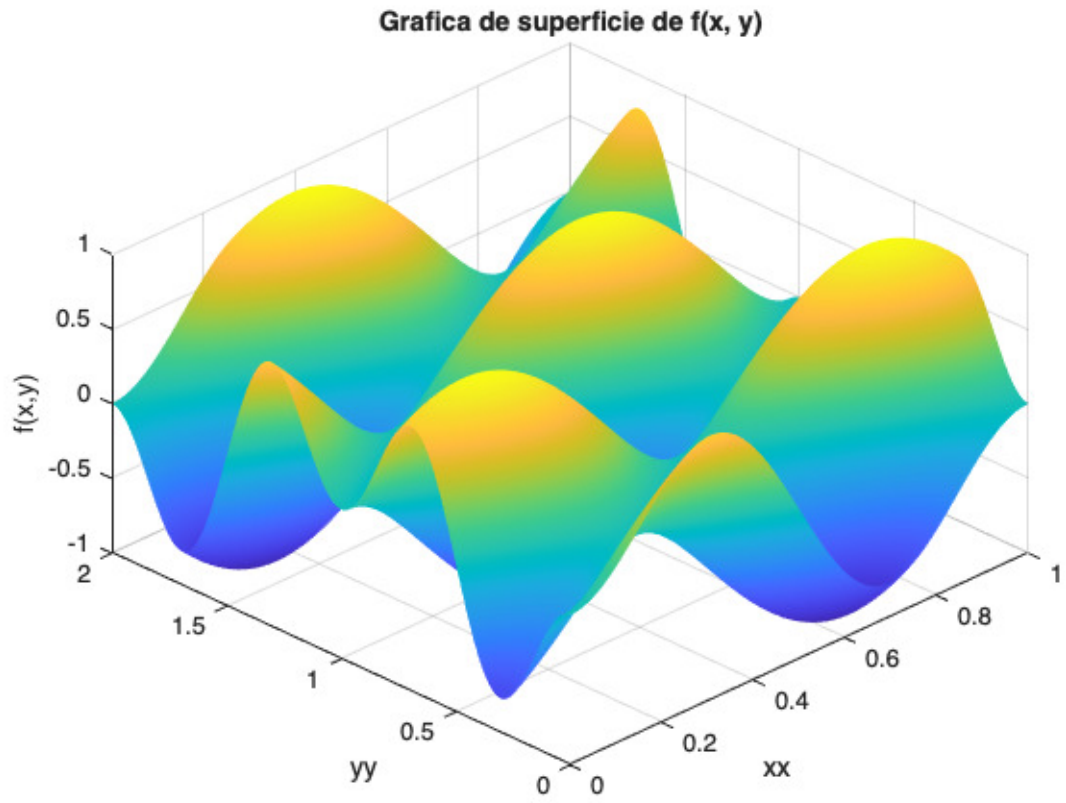
   `x = linspace(0,1,100), y = linspace(0,2,200)`

   La malla de puntos equidistantes en el dominio $D = [0, 1] \times [0, 2]$ se genera mediante el comando `[xx,yy]=meshgrid(x,y)` (note que `xx` y `yy` son matrices). Grafique la función

   $$f(x, y) = \sin(2\pi(x + y)) \sin(\pi(x - y))$$

   en esta malla del dominio D. Sugerencia: utilice el comando `surf(xx,yy,f(xx,yy))` y revise opciones de visualización como `view`, `colorbar`, `shading`.

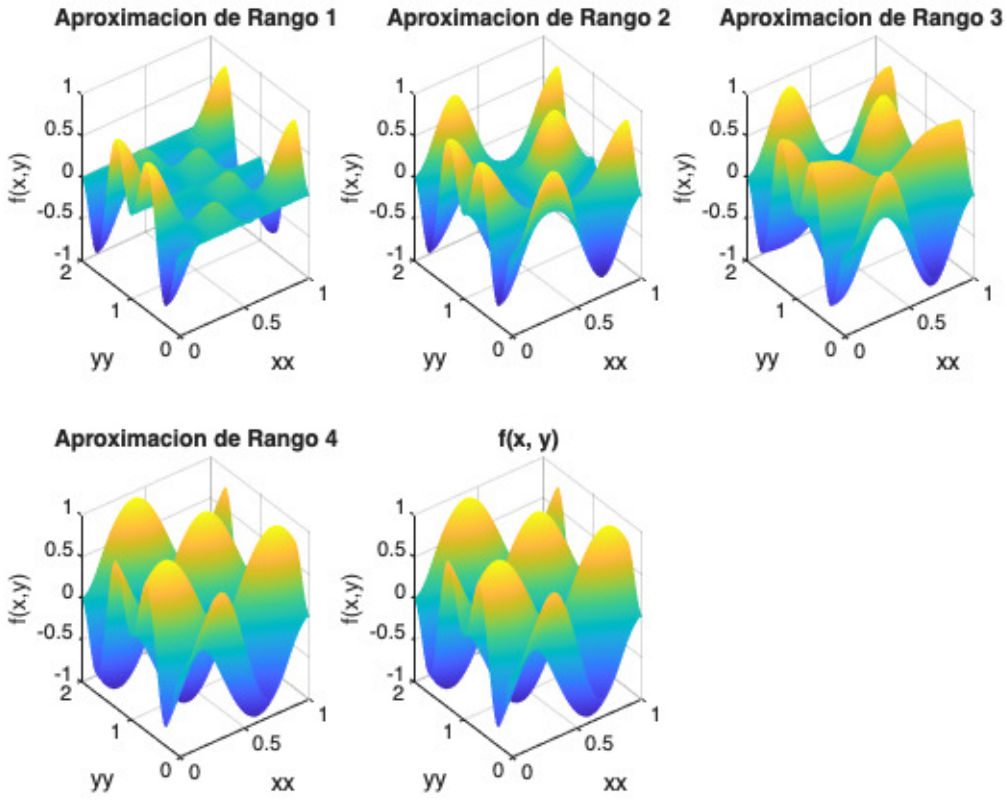
```
x = linspace(0,1,100);
y = linspace(0,2,200);

[xx,yy] = meshgrid(x,y);

f = @(x,y) sin(2.*pi.*(x+y)).*sin(pi.*(x-y));
surf(xx,yy,f(xx,yy))
view([-45 45])
shading interp
xlabel('xx')
ylabel('yy')
zlabel('f(x,y)')
title('Grafica de superficie de f(x, y)')
```



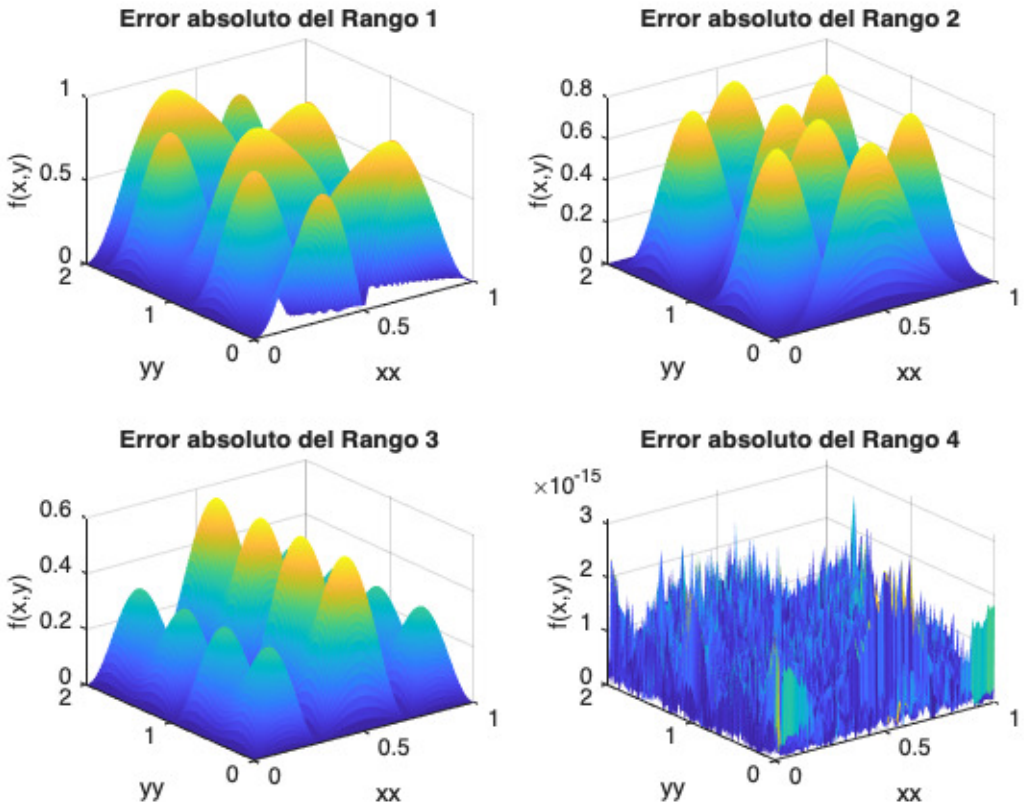
b) (4 puntos) [MATLAB] Defina `zz = f(xx,yy)`, donde f es la función del inciso anterior. Calcule la descomposición en valores singulares de `zz`. Mediante el comando `subplot`, grafique en una misma ventana las mejores aproximaciones de rango 1,2,3 y 4. En una nueva ventana, grafique el error absoluto para cada una de las cuatro aproximaciones. ¿Qué observa?

```
zz = f(xx,yy);
[U,S,V] = svd(zz);
val = diag(S);
aprox = cell(4,1);
aprox{1} = val(1)*U(:,1)*V(:,1)';
for i = 2:4
    aprox{i} = aprox{i-1} + val(i)*U(:,i)*V(:,i)';
end

figure(1)
for i = 1:4
    subplot(2,3,i);
```

```
    surf(xx, yy, aprox{i})
    shading interp
    xlabel('xx')
    ylabel('yy')
    zlabel('f(x,y)')
    title(sprintf('Aproximacion de Rango %d', i))
end

subplot(2,3,5)
surf(xx,yy,f(xx,yy))
shading interp
xlabel('xx')
ylabel('yy')
zlabel('f(x,y)')
title('f(x, y)')
```



```
error = cell(4,1);
for i = 1:4
    error{i} = abs(f(xx, yy) - aprox{i});
    figure(2)
    subplot(2, 2, i)
    surf(xx, yy, error{i})
    shading flat
    xlabel('xx')
    ylabel('yy')
    zlabel('f(x,y)')
    title(sprintf('Error absoluto del Rango %d', i))
end
```



Observamos que la aproximación de rango 4 tiene una precisión de 15 decimales, por lo que para la máquina es bastante precisa.

c) (2 puntos) [`MATLAB`] ¿Cuáles son los valores singulares y el rango de `zz`?

```
disp(val(1:4))
```

```
   35.7919
   35.4410
   35.0901
   35.0901
```

```
rank(zz)
```

```
ans = 4
```

d) (4 puntos) Demuestre (a mano) que la matriz tiene el rango y los valores singulares correspondientes. Sugerencia: recuerde fórmulas para $\sin(x \pm y)$.

Prueba
Expandiendo la función

$$f(x,y) = \sin(2\pi(x+y))\sin(\pi(x-y))$$

$$= (\sin 2\pi x \cos 2\pi y + \sin 2\pi y \cos 2\pi x)(\sin \pi x \cos \pi y - \sin \pi y \cos \pi x)$$

$$= \sin 2\pi x \cos 2\pi y \sin \pi x \cos \pi y - \sin 2\pi x \cos 2\pi y \sin \pi y \cos \pi x +$$

$$\sin 2\pi y \cos 2\pi x \sin \pi x \cos \pi y - \sin 2\pi y \cos 2\pi x \sin \pi y \cos \pi x$$

De acá nos damos cuenta que cada una de estas son una matriz, para sumar todas en una, ya que son vectores independientes. Por lo que nos estamos enfrentando a las aproximaciones de rango, y lo que podemos observar es que ambos vectores x, y, además de independientes, generan un espacio en cada una de estas matrices, para hacer los vectores singulares. Sin enbargo, estos tienen que ser unitarios, y por tanto:

$$\sigma_1 = \|\sin 2\pi x \sin \pi x\| \|\cos 2\pi y \cos \pi y\|$$

$$\sigma_2 = \|\sin 2\pi x \cos \pi x\| \|\cos 2\pi y \sin \pi y\|$$

$$\sigma_3 = \|\cos 2\pi x \sin \pi x\| \|\sin 2\pi y \cos \pi y\|$$

$$\sigma_4 = \|\cos 2\pi x \cos \pi x\| \|\sin 2\pi y \sin \pi y\|$$

```
disp(norm(cos(2.*pi.*x).*cos(pi.*x))*norm(sin(2.*pi.*y).*sin(pi.*y)))
```

```
   35.7919
```

```
disp(norm(sin(2.*pi.*x).*sin(pi.*x))*norm(cos(2.*pi.*y).*cos(pi.*y)))
```

```
35.4410
```

```
disp(norm(sin(2.*pi.*x).*cos(pi.*x))*norm(cos(2.*pi.*y).*sin(pi.*y)))
```

```
35.0901
```

```
disp(norm(cos(2.*pi.*x).*sin(pi.*x))*norm(sin(2.*pi.*y).*cos(pi.*y)))
```

```
35.0901
```

4. Sea

$$p(x) = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1},\ x \in [0,1].$$

Considere el problema de minimizar

$$E(c_0, \ldots, c_{n-1}) := \|f - p\|_2^2 = \int_0^1 |f(x) - p(x)|^2\ dx \to \text{mín}\,.$$

(a) (5 puntos) Obtenga las derivadas parciales $\partial E(c_0, \ldots, c_{n-1})/\partial c_k$ $(k = 0, \ldots, n-1)$. Para obtener el punto crítico, exprese el resultado como un sistema de n ecuaciones con n incógnitas, $M\boldsymbol{c} = \boldsymbol{b}$. ¿Qué problema tiene este sistema?

Para minimizar la integral ocupamos que su derivada sea cero, en otras palabras,

$$\int_0^1 \frac{\partial}{\partial c_k} |f(x) - p(x)|^2\ dx = 0 \implies \int_0^1 2(f(x) - p(x))x^k\ dx = 0$$

$$\implies \int_0^1 f(x)x^k\ dx = \int_0^1 p(x)x^k\ dx$$

$$\implies \int_0^1 f(x)x^k\ dx = \sum_{i=0}^{n-1} c_i \frac{1}{i+k+1}$$

Por lo que este problema se puede ver como una multiplicación de matrices con vectores, $M\boldsymbol{c} = \boldsymbol{b}$, donde

$$M_{ij} = \frac{1}{i+j-1},\ b_i = \int_0^1 f(x)x^i\ dx$$

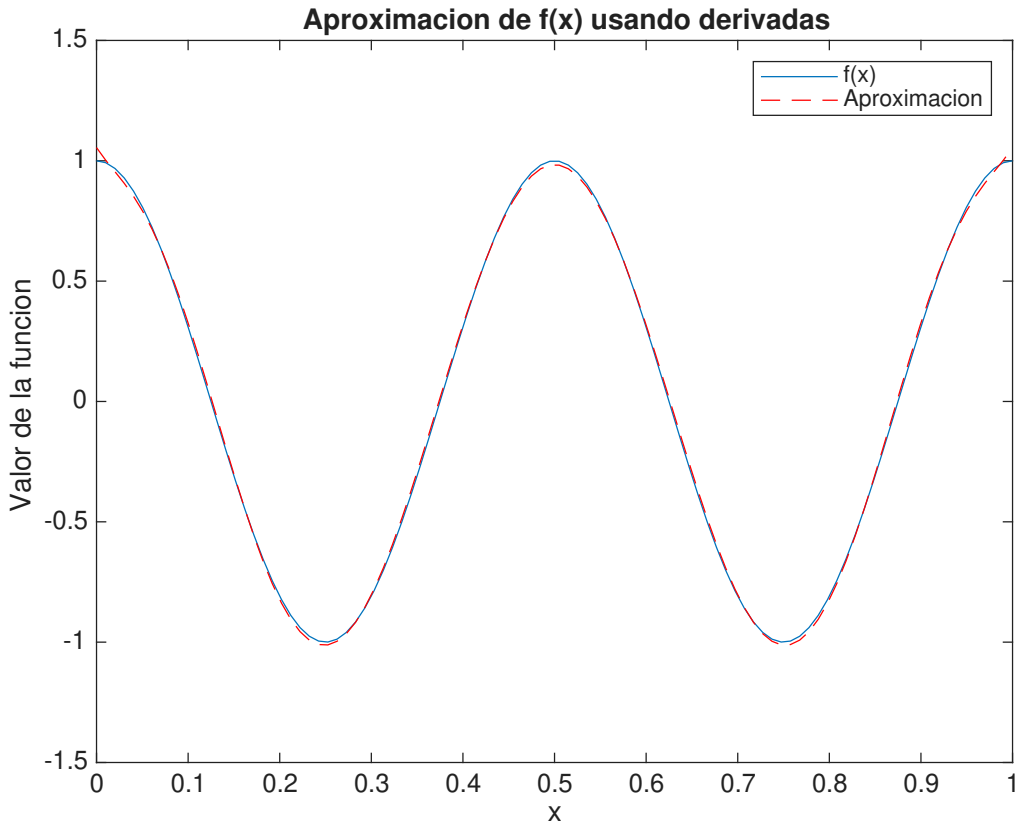
El problema con este sistema es que contiene la matriz de Hilbert, que tiene un número de condición proporcional al tamaño de n. Por lo que para valores altos de n, tiene un número de condición bastante alto, y por tanto la matriz presenta un comportamiento singular.

(b) (5 puntos) [MATLAB] Resuelva el sistema correspondiente para $n = 10$, $f(x) = \cos(4\pi x)$. Dibuje en un mismo gráfico f y p. ¿Cuál es el valor de $\|f - p\|_2^2$? Sugerencia: puede utilizar el comando `integral(f,0,1)`.

```
function c = calculo_aprox(f, n)
    %
    %   Hace un cálculo aproximado de la función aproximada con
    %   la matriz de Hilbert
    %
    %   Inputs
    %       f: función a aproximar
    %       n: tamaño del polinomio interpolador
    %
    %   Outputs
    %       c: vector de constantes de la factorización Mc = b

    M = zeros(n);
    b = zeros(n,1);
    for i = 1:n
        b(i) = integral(@(x) f(x).*x.^(i-1), 0, 1);
        for j = 1:n
            M(i,j) = 1/(i+j-1);
        end
    end
    c = M\b;
end

f = @(x) cos(4.*pi.*x);
c = calculo_aprox(f, 10);
p = @(x) polyval(c(end:-1:1), x);
integral(@(x) (f(x) - p(x)).^2, 0, 1)
```

```
ans =
     1.733766301763702e-04
```

```
xx = linspace(0,1);
figure;
plot(xx, f(xx), '-', 'DisplayName', 'f(x)')
hold on
plot(xx, p(xx), 'r--', 'DisplayName', 'Aproximacion')
hold off
legend;
title('Aproximacion de f(x) usando derivadas');
xlabel('x');
ylabel('Valor de la funcion');
```



(c) (5 puntos) [MATLAB] Para obtener una mejor aproximación, considere la base de polinomios ortogonales de Legendre $\{\phi_j\}_{j\geq 0}$. La proyección de f sobre el espacio de polinomios de grado $n-1$ viene dada por

$$p(x) = \beta_0 + \beta_1\phi_1(x) + \dots \beta_{n-1}\phi_{n-1}(x),$$

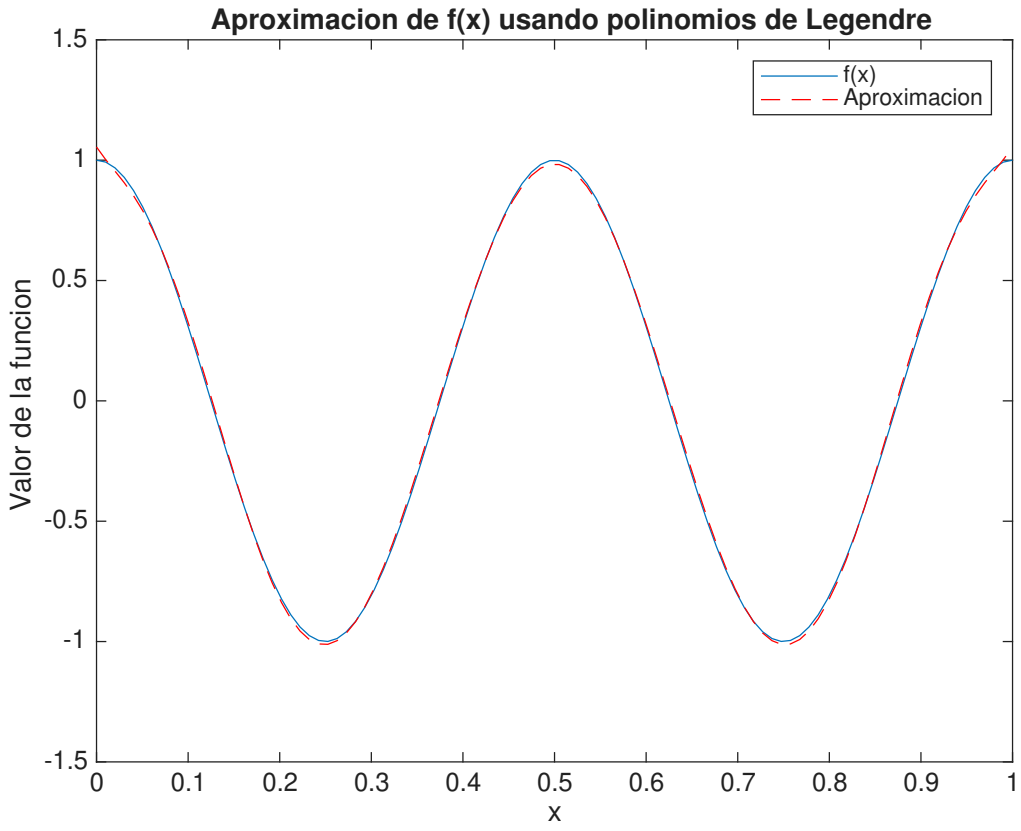
donde

$$\beta_k = \frac{\langle f, \phi_k\rangle}{\langle \phi_k, \phi_k\rangle}.$$

Calcule p para $n = 10$, $f(x) = \cos(4\pi x)$. Compare con los resultados del apartado anterior. Dibuje en la misma gráfica la nueva aproximación y calcule la 2-norma del error.

```
function p = aprox_leg(n, f)
    %
    %   Hace un cálculo aproximado de la función aproximada con
    %   los polinomios de legendre
    %
    %   Inputs
    %       f: función a aproximar
    %       n: tamaño del polinomio interpolador
    %
    %   Outputs
    %       p: polinomio interpolador

    pn = cell(n, 1);
    pn{1} = @(x) 1;
    pn{2} = @(x) x;

    % Construye los polinomios de Legendre
    for k = 2:(n-1)
        pn{k+1} = @(x) ((2*k-1)*x.*pn{k}(x) - (k-1)*pn{k-1}(x)) / (k);
    end

    % Calcula los betas
    betas = zeros(n, 1);
    for k = 1:n
        betas(k) = integral(@(x) f(x) .* pn{k}(2*x-1), 0, 1) / ...
                   integral(@(x) pn{k}(2*x-1).^2, 0, 1,
                   'ArrayValued', true);
    end
```

```
    % suma para obtener el polinomio
    p = @(x) sum(cell2mat(arrayfun(@(k) betas(k) .* pn{k}(2*x-1), ...
        1:n, 'UniformOutput', false)));
end

n = 10;
p = aprox_leg(n, f);

figure;
xx = linspace(0, 1, 100);
plot(xx, f(xx), '-', 'DisplayName', 'f(x)');
hold on;
plot(xx, arrayfun(p, xx), 'r--', 'DisplayName', 'Aproximacion');
hold off;
legend;
title('Aproximación de f(x) usando polinomios de
Legendre');
xlabel('x');
ylabel('Valor de la funcion');
```



```
norma_error = integral(@(x) (f(x) - arrayfun(p, x)).^2, 0, 1);
disp(['Norma del error: ', num2str(norma_error)]);
```

```
Norma del error: 0.00017338
```

5. Si $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^m$, la matriz $A = I + \boldsymbol{u}\boldsymbol{v}^T$ es llamada *perturbación de rango uno de la identidad*.

   (a) (4 puntos) Muestre que si A es invertible, entonces su inversa tiene la forma $A^{-1} = I + \alpha \boldsymbol{u}\boldsymbol{v}^T$ con $\alpha \in \mathbb{R}$. Encuentre el valor de α explícitamente.

   Prueba

   Por hipótesis A es invertible, entonces se cumple que $AA^{-1} = I$. Note que:

$$\begin{aligned} AA^{-1} = I &\implies (I + uv^T)(I + \alpha uv^T) = I \\ &\implies I + \alpha uv^T + uv^T + \alpha(uv^T)(uv^T) = I \\ &\implies I + \alpha uv^T + uv^T + \alpha(v^T u)(uv^T) = I \\ &\implies I + uv^T(\alpha + 1 + \alpha v^T u) = I \\ &\implies \alpha + 1 + \alpha v^T u = 0 \\ &\implies \alpha + \alpha v^T u = -1 \\ &\implies \alpha(1 + v^T u) = -1 \\ &\implies \alpha = -\frac{1}{1 + v^T u} \end{aligned}$$

   De esta forma, se tiene que:

$$A^{-1} = I - \frac{1}{1 + v^T u}\, uv^T$$

   (b) (4 puntos) ¿Para qué valores de $\boldsymbol{u}$ y $\boldsymbol{v}$ es A singular? En este caso, determine el núcleo de A.

   Para encontrar los valores de u y v que hacen que A sea singular, basta con analizar donde se indefine A^{-1}, note que:

$$A^{-1} \text{ se indefine} \iff 1 + v^T u = 0 \iff v^T u = -1$$

   Para encontrar el núcleo, se tiene:

$$\begin{aligned} Ax = 0 &\implies (I + uv^T)x = 0 \\ &\implies x + uv^T x = 0 \\ &\implies x = -uv^T x \end{aligned}$$

Observe que $v^T x$ es un escalar, si denotamos $\lambda = -v^T x$, se concluye que:

$$x = \lambda u$$

Por lo tanto, el núcleo de A viene dado por el conjunto generado por u, i.e.,

$$ker(A) = span(u)$$

6. (6 puntos) Dada $A = U\Sigma V^T \in \mathbb{R}^{m\times n}$ una descomposición en valores singulares de A, defina su pseudoinversa como $A^+ = V\Sigma^+ U^T$, donde Σ^+ se obtiene de Σ al invertir sus entradas no nulas; esto es$(\Sigma^+)_{ii} = 1/\sigma_{ii}$ si $\sigma_{ii} > 0$. Demuestre que si $\boldsymbol{x}$ está en el espacio fila de A y $\boldsymbol{y} = A\boldsymbol{x}$ entonces $\boldsymbol{x} = A^+\boldsymbol{y}$.

Sea entonces el vector $\boldsymbol{x}$ una combinación lineal de las filas, o podemos hacerlo con los vectores singulares derechos. Sea $\boldsymbol{c}$ un vector constante, entonces si r son la cantidad de valores singulares no nulos (por factorización resumida)

$$\boldsymbol{x} = \sum_{i=1}^{r} v_i c_i$$

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^t \implies A\boldsymbol{x} = \sum_{i=1}^{r} \sigma_i u_i v_i^t v_i c_i = \sum_{i=1}^{r} \sigma_i u_i c_i = \boldsymbol{y}$$

Procedemos a multiplicar por $A^+ = \sum_{i=1}^{r} \sigma_i^{-1} v_i u_i^t$

$$\implies A^+\boldsymbol{y} = \sum_{i=1}^{r} \sigma_i \sigma_i^{-1} v_i u_i^t u_i c_i = \sum_{i=1}^{r} v_i c_i = \boldsymbol{x}$$

7. Considere un conjunto de mediciones $\{(x_i, y_i)\}_{i=1}^m$. En este ejercicio deseamos resolver el problema de mínimos cuadrados, donde la función buscada $f(x, \boldsymbol{c})$ no depende linealmente de los coeficientes $\boldsymbol{c} = [c_1, \ldots, c_n]^T \in \mathbb{R}^n$ (en clase analizaremos el caso donde f es un polinomio). De esta forma, deseamos hallar $\boldsymbol{c}$ tal que

$$\sum_{i=1}^{m} |f(x_i, \boldsymbol{c}) - y_i|^2 \to \text{mín}\,.$$

Para x_i fijo y f con derivadas continuas (con respecto a $c_1, \ldots, c_n$), considere la aproximación de primer orden con centro $\boldsymbol{c}^{(k)}$ dada por

$$f(x_i, \boldsymbol{c}) \approx f(x_i, \boldsymbol{c}^{(k)}) + J_f(x_i, \boldsymbol{c}^{(k)})(\boldsymbol{c} - \boldsymbol{c}^{(k)}),$$

donde

$$J_f(x_i, \boldsymbol{c}) = \nabla_{\boldsymbol{c}} f(x_i, \boldsymbol{c}) \in \mathbb{R}^{1\times n}$$

es el Jacobiano de f con derivadas parciales $\dfrac{\partial f}{\partial c_j}$ evaluado en x_i.

De esta forma, deseamos que

$$J_f(x_i, \boldsymbol{c}^{(k)})(\boldsymbol{c} - \boldsymbol{c}^{(k)}) \approx y_i - f(x_i, \boldsymbol{c}^{(k)}) \;\forall\; i = 1, \ldots, m. \tag{1}$$

Defina la matriz $J(\boldsymbol{c}^{(k)}) \in \mathbb{R}^{m\times n}$ donde su fila i viene dada por $J_f(x_i, \boldsymbol{c}^{(k)})$. Defina además

$$\boldsymbol{y} := [y_1, \ldots, y_m]^T \in \mathbb{R}^m,$$
$$\boldsymbol{g}(\boldsymbol{c}) := [f(x_1, \boldsymbol{c}), \ldots, f(x_m, \boldsymbol{c})]^T \in \mathbb{R}^m.$$

Podemos escribir así las ecuaciones dadas en (1) de forma matricial. Si sabemos que $\boldsymbol{c}^{(k)}$ es cercano a $\boldsymbol{c}$, buscamos encontrar $\boldsymbol{c}$ tal que

$$J(\boldsymbol{c}^{(k)})(\boldsymbol{c} - \boldsymbol{c}^{(k)}) \approx \boldsymbol{y} - \boldsymbol{g}(\boldsymbol{c}^{(k)}). \tag{2}$$

Para tal efecto, consideramos el siguiente algoritmo:

- Considere $\boldsymbol{c}^{(0)}$ dado.
- Para $k = 0, 1, 2, \ldots$
  - Obtenga $\Delta\boldsymbol{c}^{(k)}$ tal que $\|J(\boldsymbol{c}^{(k)})\Delta\boldsymbol{c}^{(k)} - (\boldsymbol{y} - \boldsymbol{g}(\boldsymbol{c}^{(k)}))\|_2 \to$ mín (por ejemplo, mediante el uso de las ecuaciones normales).
  - Defina $\boldsymbol{c}^{(k+1)} = \boldsymbol{c}^{(k)} + \Delta\boldsymbol{c}^{(k)}$.

En este ejercicio asumimos que f tiene la forma

$$f(x, \boldsymbol{c}) = c_1 e^{-c_2 x} \sin(c_3 x + c_4), x \in [0, 10],$$

donde $\boldsymbol{c} = [c_1, c_2, c_3, c_4]^T \in \mathbb{R}^4$ es el vector de coeficientes que debemos de encontrar (los cuales corresponden a la amplitud, el decaimiento, el periodo y la fase de f, respectivamente). Para obtener cotas del error, asumiremos que la solución exacta viene dada por el vector

$$\boldsymbol{c}_{\text{exact}} = [1, 1/2, 2, 0]^T.$$

(a) (1 punto) Utilice el comando `load data` para cargar los datos guardados en el archivo `data.dat` suministrado. Este archivo incluye los valores $\{(x_i, y_i)\}_{i=1}^{m}$ para $m = 100$, guardados en dos vectores $\boldsymbol{x}$ y $\boldsymbol{y}$.

```
data = load("data/data.mat");
```

(b) (5 puntos) Implemente un programa que ejecute el algoritmo descrito. Para detener la iteración, utilice la condición $\|\Delta \boldsymbol{c}^{(k)}\|_\infty < 10^{-8}$.

```
function fxc = funcion(x, c)
    %
    %   Función que evalúa
    %   f(x, c) = c1 * exp(-c2 * x) * sin(c3 * x + c4)
    %
    %   Inputs
    %       x: vector con las x
    %       c: vector con los coeficientes [c1, c2, c3, c4]
    %
    %   Output
    %       f: valores de f(x, c) evaluados en x

    fxc = c(1) * exp(-c(2) * x) .* sin(c(3) * x + c(4));
end

function Jf = jacobiano_f(x, c)
    %
    %   Función que calcula el Jacobiano de f respecto a los
    %   coeficientes c = [c1, c2, c3, c4].
    %
    %   Inputs
    %       x: vector con las x
    %       c: vector con los coeficientes [c1, c2, c3, c4]
    %
    %   Output
    %       J: matriz Jacobiana evaluada en x y c

    Jf = zeros(length(x), 4);

    Jf(:, 1) = exp(-c(2) * x) .* sin(c(3) * x + c(4));
    Jf(:, 2) = -c(1) * x .* exp(-c(2) * x) .* sin(c(3) * x + c(4));
    Jf(:, 3) = c(1) * exp(-c(2) * x) .* x .* cos(c(3) * x + c(4));
    Jf(:, 4) = c(1) * exp(-c(2) * x) .* cos(c(3) * x + c(4));
end
```

```
function [c, iter, norma_delta_c, num_cond] = minimos_cuadrados(x,
y, c0)
    %
    %   Función que ejecuta el algoritmo para minimizar la diferencia
    %   entre la función f(x, c) y los valores observados y.
    %
    %   Inputs
    %       x: vector con las x
    %       y: vector con las y
    %       c0: vector con los coeficientes [c1, c2, c3, c4]
    %
    %   Outputs
    %       c: vector de coeficientes ajustados
    %       iter: número de iteraciones requeridas
    %       norma_delta_c: valor de la norma de delta c por iteración
    %       num_cond: número de condición por itereción


    tol = 1e-8;
    c = c0;
    iter = 0;
    delta_c = inf;
    num_cond = [];

    while norm(delta_c, inf) > tol

        dif = y - funcion(x, c);
        J = jacobiano_f(x, c);

        delta_c = (J' * J) \ (J' * dif);
        c = c + delta_c;

        iter = iter + 1;
        norma_delta_c(iter) = norm(delta_c, inf);
        num_cond(iter) = cond(J);
    end
end
```

(c) (4 puntos) Corra su programa para $\boldsymbol{c}_0 = [1.1; 0.4; 2.1; 0.2]$. Reporte el número de iteraciones requeridas. Grafique los nodos $\{x_i, y_i\}_{i=1}^m$, la función exacta $f(x, \boldsymbol{c}_{\text{exact}})$ y la función aproximada $f(x, \boldsymbol{c}^{(k)})$. Grafique además $\|\Delta \boldsymbol{c}^{(k)}\|_\infty$ en función de k. ¿Qué puede deducir sobre la velocidad de convergencia?

```
x = data.x;
y = data.y;
c0 = [1.1; 0.4; 2.1; 0.2];

% Ejecución del algoritmo
[c, iter, norma_delta_c, num_cond] = minimos_cuadrados(x, y, c0);

disp("Número de iteraciones:");
```

```
Número de iteraciones:
```

```
disp(iter);
```

```
     7
```

```
disp("Coeficientes ajustados:");
```

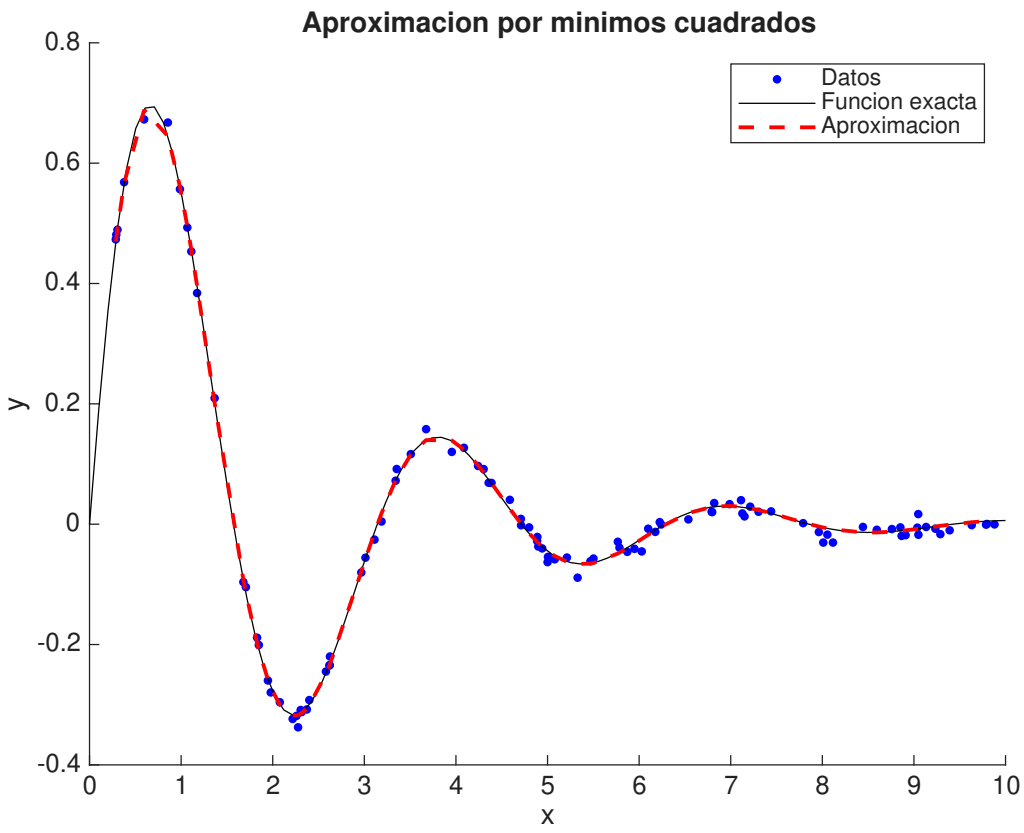
```
Coeficientes ajustados:
```

```
disp(c);
```

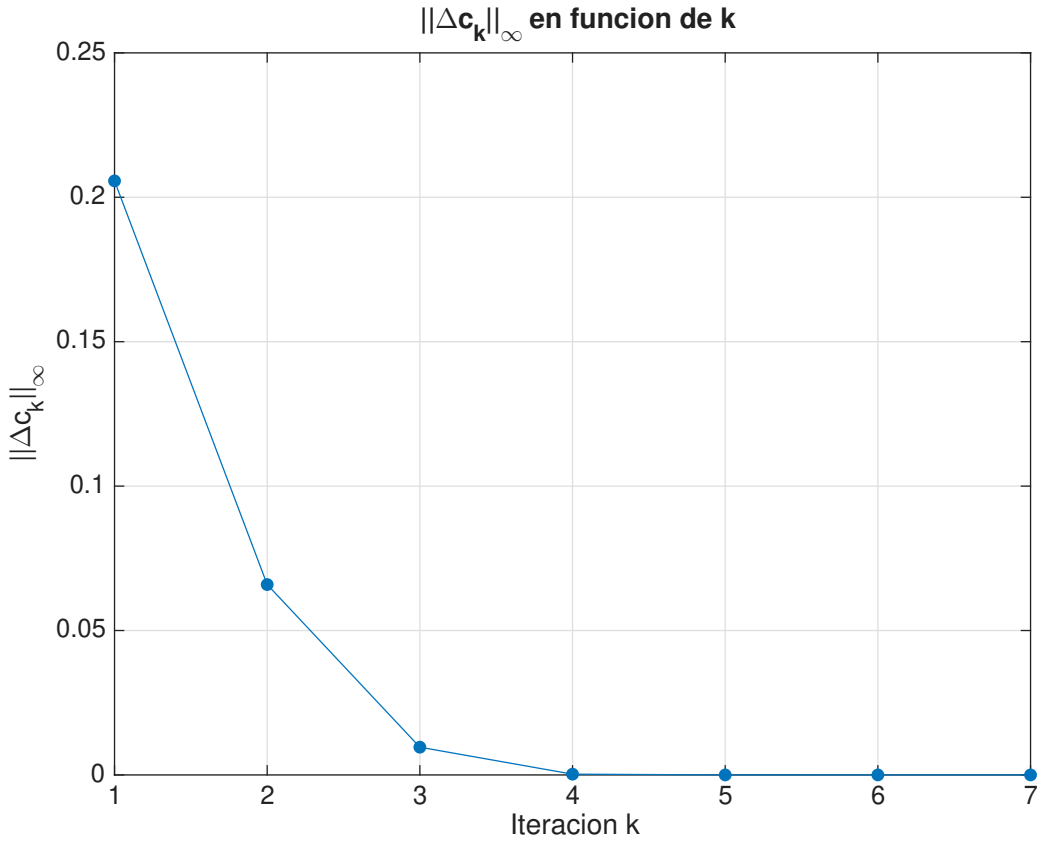
```
   0.996624262952883
   0.497520108541692
   1.998430950161934
   0.001284297892960
```

```
% Función exacta
c_exact = [1; 0.5; 2; 0];
xx = linspace(0, 10);
yy = funcion(xx, c_exact);

% Gráficas
figure;
hold on;
plot(x, y, 'b.', 'DisplayName', 'Datos', 'MarkerSize', 10);
plot(xx, yy, 'k-', 'DisplayName', 'Funcion exacta');
plot(sort(x), funcion(sort(x), c), 'r--', 'DisplayName',
'Aproximacion', 'LineWidth', 1.5);
```

```
legend;
xlabel('x');
ylabel('y');
title('Aproximacion por minimos cuadrados');
hold off;
```



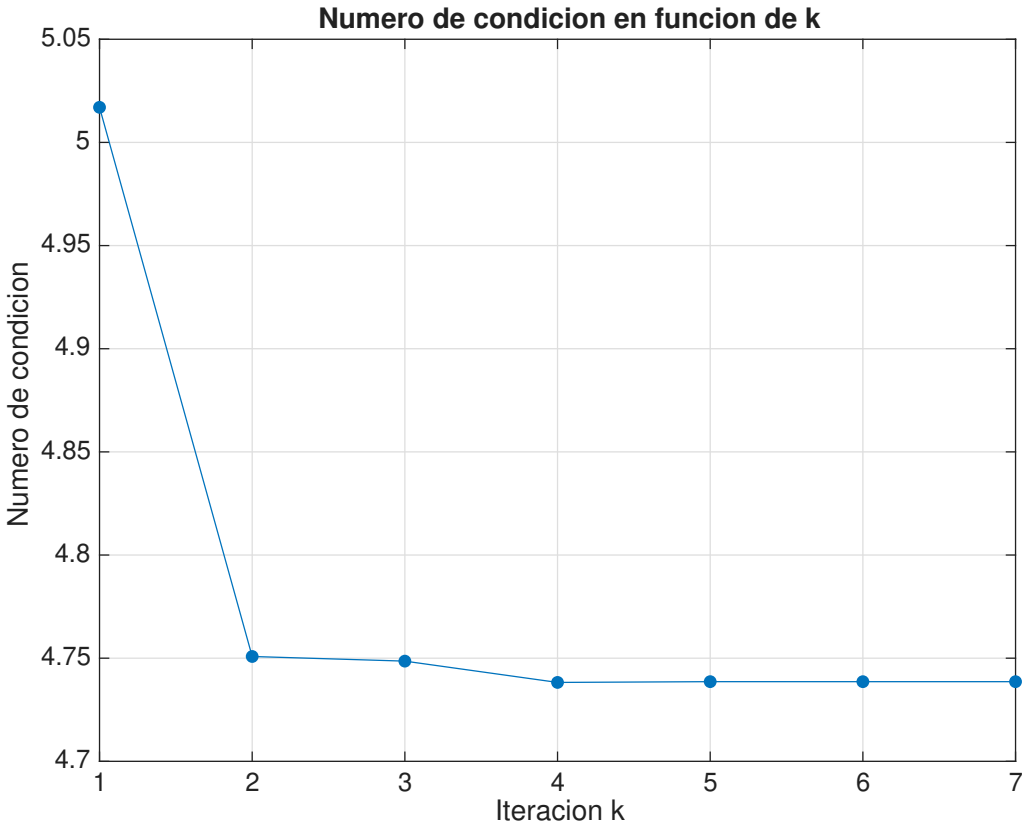

```
figure;
plot(1:iter, norma_delta_c, '.-', 'MarkerSize', 15);
xlabel('Iteracion k');
ylabel('||\Deltac_k||_\infty');
title('||\Deltac_k||_\infty en funcion de k');
grid on;
```



El comportamiento de la norma $\|\Delta \boldsymbol{c}^{(k)}\|_\infty$ en función de las iteraciones muestra que el algoritmo converge rápidamente. Esto es típico de los métodos de optimización basados en el gradiente, especialmente cuando la función es bien comportada y el punto inicial c_0 está cerca de la solución exacta. La rapidez de convergencia puede variar dependiendo de la condición del problema, pero en este caso, la convergencia fue alcanzada en solo 7 iteraciones.

(d) (2 puntos) Para el valor inicial del inciso anterior, grafique el número de condición $\kappa(J(\boldsymbol{c}^{(k)}))$ en función de k. ¿Qué nos indica este gráfico? Sugerencia: utilice la función `cond(J)`.

```
figure;
plot(1:iter, num_cond, '.-', 'MarkerSize', 15);
xlabel('Iteracion k');
ylabel('Numero de condicion');
title('Numero de condicion en funcion de k');
grid on;
```



El gráfico del número de condición $\kappa(J(\boldsymbol{c}^{(k)}))$ en función de las iteraciones k muestra la estabilidad del algoritmo. Un número de condición bajo y constante indica que el sistema está bien condicionado, lo que sugiere que el algoritmo es estable y preciso. En cambio, picos o un aumento en el número de condición pueden señalar que el sistema se vuelve mal condicionado, lo que podría afectar la precisión del algoritmo. Este gráfico ayuda a evaluar la robustez del proceso de optimización.

(e) (2 puntos) Ahora considere $\boldsymbol{c}_0 = [1, 1, 1, 1]^T$. ¿Qué ocurre en este caso?

```
c0 = [1; 1; 1; 1];
[c, iter] = minimos_cuadrados(x, y, c0);
```

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  1.843432e-25.
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  5.007698e-30.
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  4.766386e-30.
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  3.109715e-29.
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  4.668944e-26.
Warning: Matrix is singular to working precision.
```

```
disp("Número de iteraciones:");
```

```
Número de iteraciones:
```

```
disp(iter);
```

```
     9
```

```
disp("Coeficientes ajustados:");
```

```
Coeficientes ajustados:
```

```
disp(c);
```

```
   NaN
   NaN
   NaN
   NaN
```

En este caso, al usar el valor inicial $c0 = [1, 1, 1, 1]^T$, el algoritmo de mínimos cuadrados produce advertencias sobre la singularidad o mala escala de la matriz, lo que indica que el sistema está cerca de ser singular. Esto provoca que los coeficientes ajustados sean valores no numéricos (NaN), lo cual es una consecuencia de la mala condición de la matriz. El sistema es mal condicionado debido a la elección de los valores iniciales, lo que impide que el algoritmo encuentre una solución válida.

8. Considere un conjunto de $m+1$ puntos $\{(x_i, y_i)\}_{i=0}^{m}$, donde

$$0 = x_0 < x_1 < \ldots < x_{m-1} < x_m = 1$$

es una partición del intervalo $[0,1]$ (no necesariamente con puntos equidistantes). En particular asumiremos que $y_0 = y_m$ pues consideraremos funciones periódicas. Deseamos construir una función $s_2 : \mathbb{R} \to \mathbb{R}$ tal que:

- s_2 es periódica con periodo 1; esto es, $s_2(x+1) = s_2(x)\ \forall\ x \in \mathbb{R}$.
- $s_2 \in C^1(\mathbb{R})$; esto es, s_2 tiene derivada continua.
- s_2 interpola los valores del conjunto de puntos; esto es, $s_2(x_i) = y_i\ \forall\ i = 0, 1, \ldots, m$.
- la restricción de s_2 a cada subintervalo $[x_i, x_{i+1}]$ $(i = 0, 1, \ldots, m-1)$ es un polinomio cuadrático.

Es claro que basta construir s_2 en el intervalo $[0,1]$ al ser periódica.

(a) (7 puntos) Escriba un sistema de ecuaciones que permita determinar $s_2 : [0,1] \to \mathbb{R}$ (Sugerencia: puede seguir la construcción de los splines cúbicos naturales del Capítulo 7; en este caso defina $\sigma_i = s_2'(x_i),\ i = 0, 1, \ldots, m$ y genere un sistema para encontrar los σ_i. Otra idea es escribir la restricción de $s_2(x)$ en cada subintervalo $[x_i, x_{i+1}]$ de la forma $s_2(x) = \alpha_i + \sigma_i(x - x_i) + \beta_i(x - x_i)^2$. Recuerde además que s_2 debe ser periódica).

Sigamos el lado de $s_2(x) = \alpha_i + \sigma_i(x - x_i) + \beta_i(x - x_i)^2$. Sabemos que

$$s_2(x_i) = y_i \implies \alpha_i = y_i$$

Por otro lado queremos que $s_2(x_i^+) = s_2(x_i^-) \implies s_2'(x_i^+) = s_2'(x_i^-)$, por lo que tenemos que realizar los dos lados, pero antes podemos averiguar quien es β_i. Si $h_i = x_i - x_{i-1}$

$$s_2'(x) = \sigma_i + \beta_i(x - x_i) \implies \sigma_{i+1} = \sigma_i + \beta_i(x_{i+1} - x_i)$$
$$\implies \beta_i = \frac{\sigma_{i+1} - \sigma_i}{2h_{i+1}}$$

Por lo que podemos sustituir en la ecuación principal $s_2(x_i^+) = s_2(x_i^-)$

$$y_{i-1} + \sigma_{i-1}h_i + \frac{\sigma_i - \sigma_{i-1}}{2h_i}h_i^2 = y_i \implies y_i - y_{i-1} = \frac{\sigma_i + \sigma_{i-1}}{2}h_i$$

Con esta última igualdad podemos definir la matriz de sigmas, con la condición de periodicidad en la derivada, $\sigma_0 = \sigma_m$

$$X\sigma = \Delta y$$

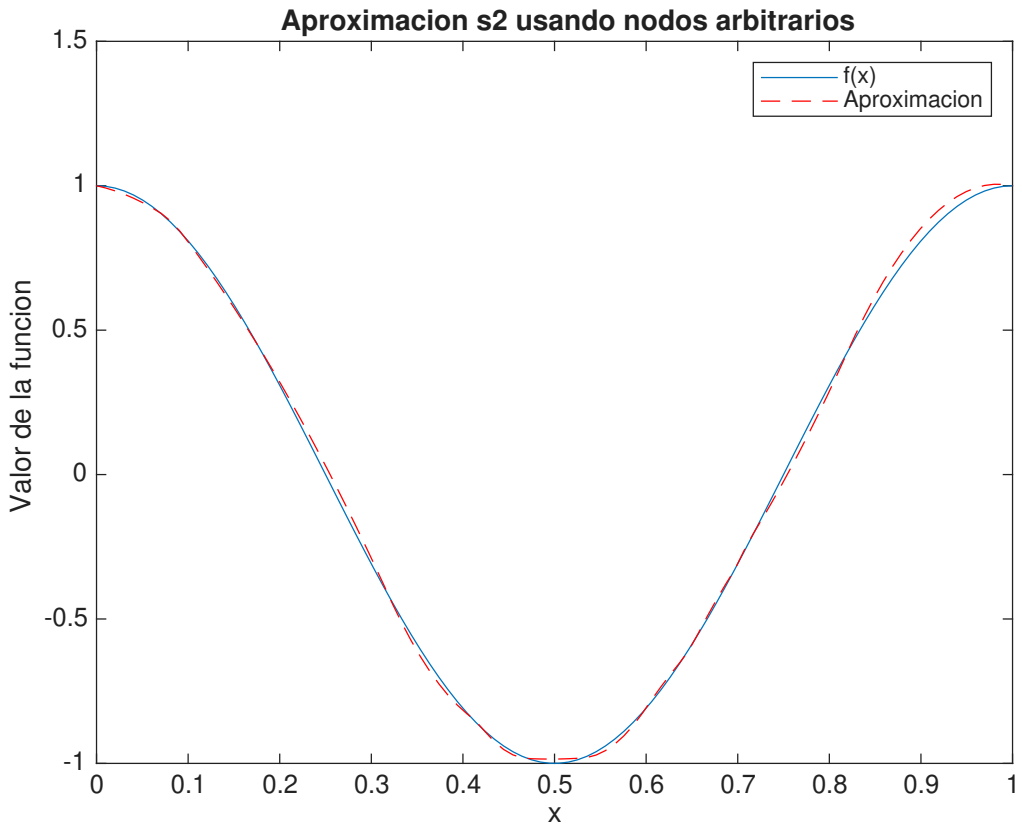
donde

$$X = \begin{pmatrix} \frac{x_1 - x_0}{2} & \frac{x_1 - x_0}{2} & 0 & \ldots & 0 & 0 \\ 0 & \frac{x_2 - x_1}{2} & \frac{x_2 - x_{1,}}{2} & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & \frac{x_m - x_{m-1}}{2} & \frac{x_m - x_{m-1}}{2} \\ 1 & 0 & 0 & \ldots & 0 & -1 \end{pmatrix}, \Delta y = \begin{pmatrix} y_1 - y_0 \\ y_2 - y_1 \\ \vdots \\ y_m - y_{m-1} \\ 0 \end{pmatrix}$$

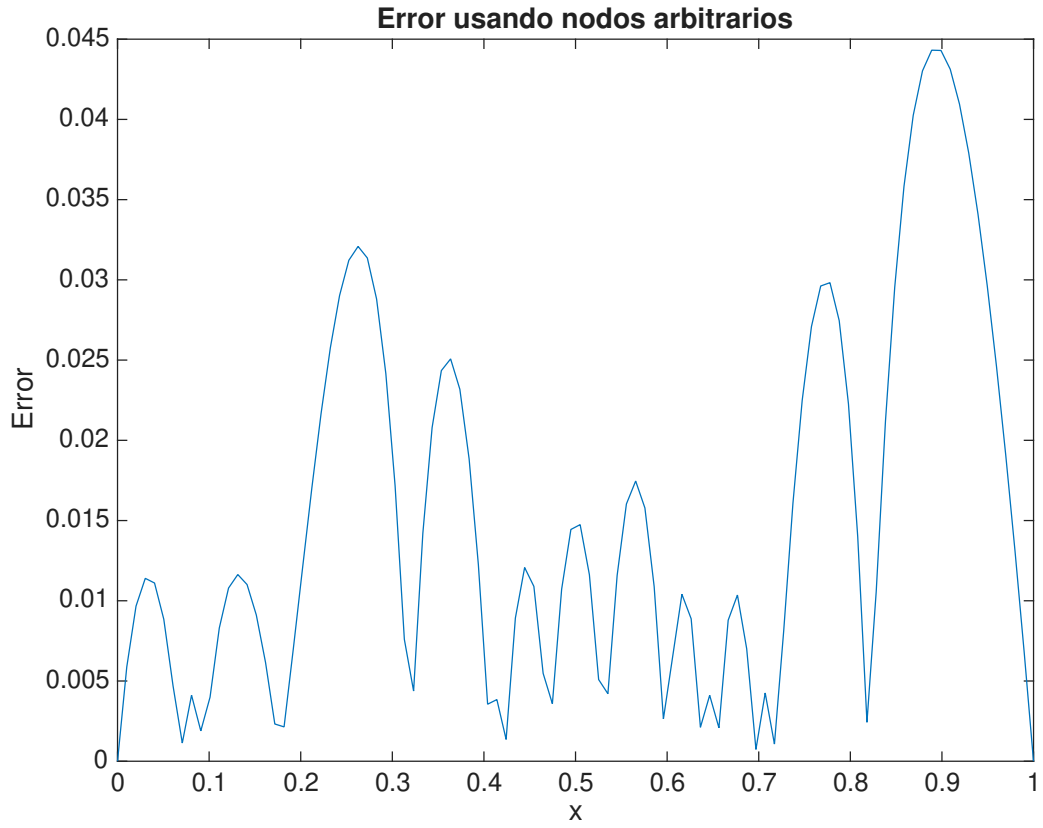
(b) [MATLAB] (4 puntos) Fije $m = 15$ y tome $f(x) := cos(2\pi x)$. Defina $\boldsymbol{x} = [x_0, \ldots, x_m]^T$ utilizando valores aleatorios con la función `rand` y calcule $y_i = f(x_i)\ \forall\ i = 0, 1, \ldots, m$. Construya s_2 para este conjunto de datos. Grafique f, s_2 y los puntos $\{(x_i, y_i)\}_{i=0}^m$ en un mismo gráfico. Grafique además el error absoluto $|f(x) - s_2(x)|$ y estime $\|f - s_2\|_{L^\infty([0,1])}$.

```
function sigmas = calculo_s2(xn, yn)
    m = length(xn);
    x = zeros(m);
    y = zeros(m,1);
    for i = 1:(m-1)
        x(i,i) = (xn(i+1)-xn(i))/2;
        x(i,i+1) = x(i,i);
        y(i) = yn(i+1) - yn(i);
    end
    x(end, 1) = 1;
    x(end, end) = -1;
    y(end) = 0;
    sigmas = x\y;
end

m = 15;
xn = sort(rand(m+1,1));
xn(1) = 0;
xn(end) = 1;
f = @(x) cos(2.*pi.*x);
yn = f(xn);

sigmas = calculo_s2(xn, yn);

function s2 = s2(sigmas, x, xn, yn)
    nodes = find(xn>x);
    if x == 1
        lower = length(xn)-1;
    else
        lower = nodes(1)-1;
    end
    s1 = sigmas(lower);
    s2 = sigmas(lower+1);
    y1 = yn(lower);
    x1 = xn(lower);
    x2 = xn(lower + 1);
```

```
    s2 = y1 + s1.*(x - x1) + ((s2 - s1)/(2 * (x2 - x1)))*(x-x1)^2;
end

xx = linspace(0,1);
yy = arrayfun(@(x) s2(sigmas, x, xn, yn), xx);
figure;
plot(xx, f(xx))
hold on;
plot(xx, yy, 'r--')
hold off;
xlabel('x')
ylabel('y')
title('Aproximacion por nodos arbitrarios')
```



```
figure;
plot(xx, abs(f(xx) - yy))
xlabel('x')
ylabel('y')
title('Error usando nodos arbitrarios')
```

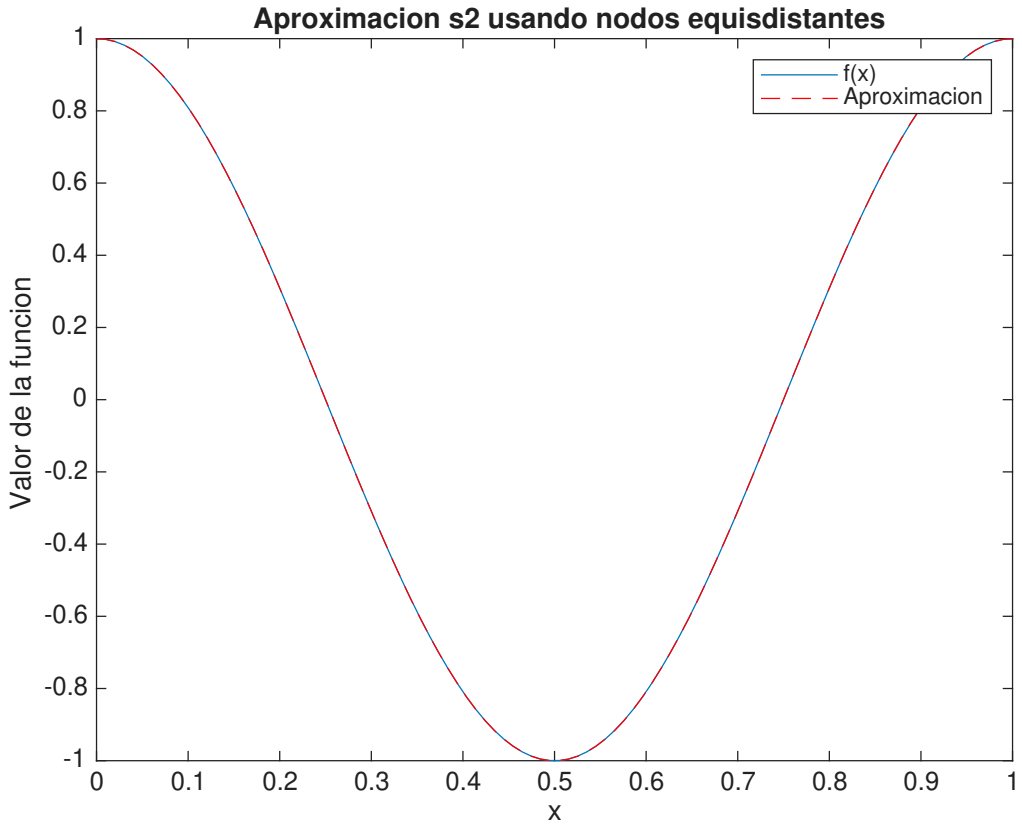



```
disp(max(abs(f(xx) - yy)))
```

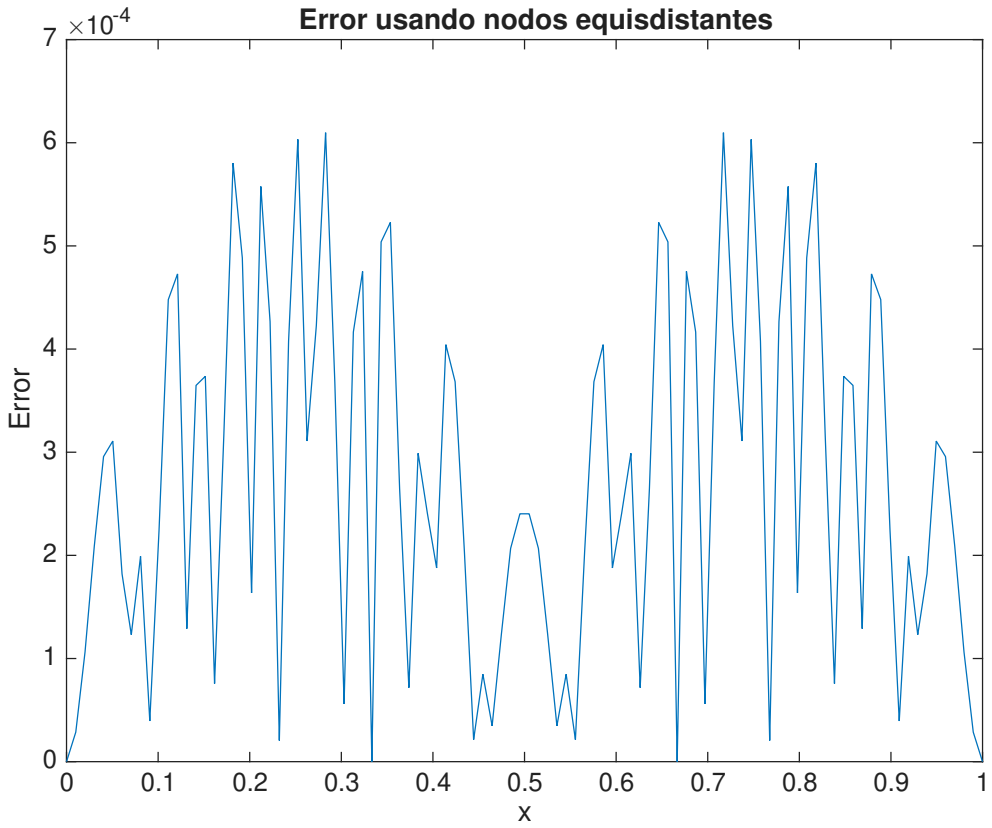
```
0.0437
```

(c) [MATLAB] (3 puntos) Repita el ejercicio anterior para $\boldsymbol{x}$ = `linspace(0,1,m+1)'`. Compare con los resultados del inciso anterior y comente sus resultados.

```
xn = linspace(0,1,m+1);
yn = f(xn);
sigmas = calculo_s2(xn, yn);
yy = arrayfun(@(x) s2(sigmas, x, xn, yn), xx);
figure;
plot(xx, f(xx))
hold on;
plot(xx, yy, 'r--')
hold off;
xlabel('x')
ylabel('y')
title('Aproximacion por nodos equisdistantes')
```



```
figure;
plot(xx, abs(f(xx) - yy))
xlabel('x')
ylabel('y')
title('Error usando nodos equisdistantes')
```




```
disp(max(abs(f(xx) - yy)))
```

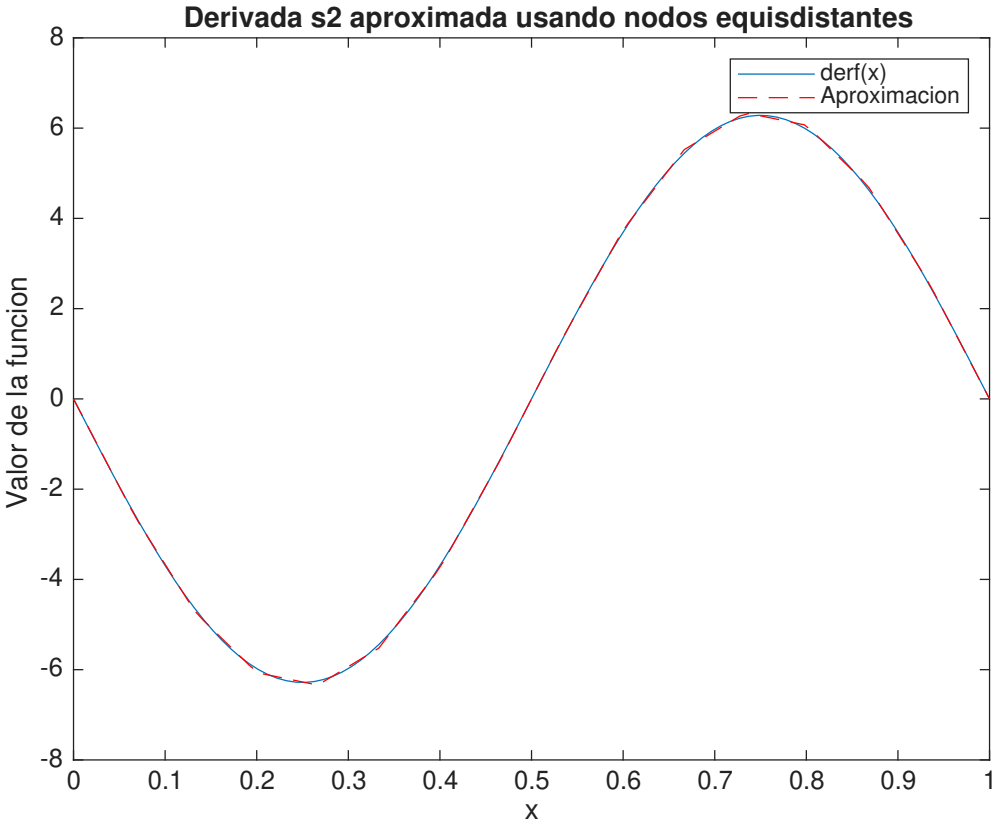
```
6.0980e-04
```

Podemos ver que con nodos equisdistantes, el sistema de ecuaciones que define las derivadas σ_i en cada punto es más estable y fácil de resolver. Esto ocurre porque los elementos de la matriz del sistema se distribuyen de manera más uniforme, lo que facilita obtener las inversas de una forma más precisa y confiable para resolver el sistema. En el caso de los nodos aleatorios, la matriz puede llegar a estar mal condicionada o inclusive puede llegar singular, lo cual puede ocasionar errores numéricos al calcular las derivadas, afectando la precisión del spline.

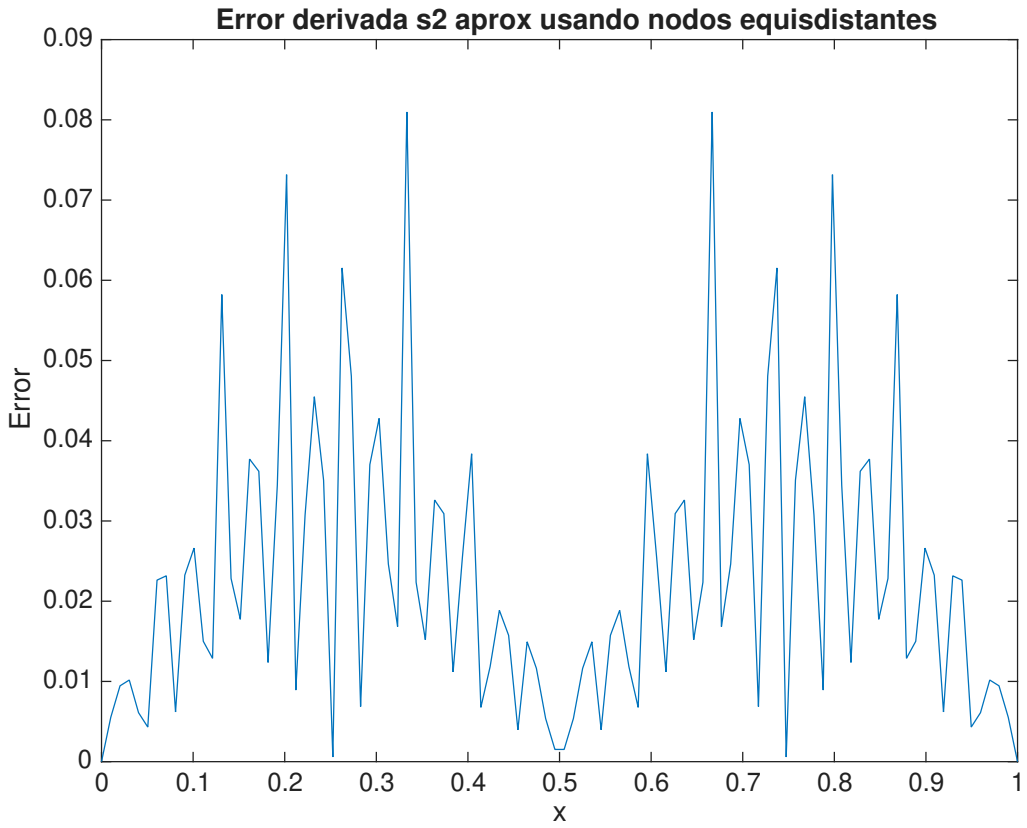
(d) [MATLAB] (6 puntos) Uno puede aproximar $f'(x)$ mediante $s_2'(x)$. Utilizando s_2 tal como se calculó en el inciso anterior, escriba un programa que calcule $s_2'(x)$. Grafique f' y s_2' en el mismo gráfico. Grafique además el error absoluto $|f'(x) - s_2'(x)|$ y estime $\|f' - s_2'\|_{L^\infty([0,1])}$. Note que s_2' es diferenciable a trozos. ¿Cómo podría aproximar f' por una función s de manera tal que s'' sea continua?

Podríamos aproximar f' con un spline cúbico natural, puesto que su segunda derivada es continua, por sus condiciones en los extremos, y además, la derivada de s_2 es continua y diferenciable de forma continua.

```
function s2 = der_s2(sigmas, x, xn)
    nodes = find(xn>x);
    if x == 1
        lower = length(xn)-1;
    else
        lower = nodes(1)-1;
    end
    s1 = sigmas(lower);
    s2 = sigmas(lower+1);
    x1 = xn(lower);
    x2 = xn(lower + 1);
    s2 = s1 + ((s2 - s1)/((x2 - x1)))*(x-x1);
end

derf = @(x) -2.*pi.*sin(2.*pi.*x);
yy = arrayfun(@(x) der_s2(sigmas, x, xn), xx);
figure;
plot(xx, derf(xx))
hold on;
plot(xx, yy, 'r--')
hold off;
xlabel('x')
ylabel('y')
title('Derivada s2 aproximacion usando nodos equisdistantes')
```


Derivada s2 aproximada usando nodos equisdistantes
derf(x)
Aproximacion
Valor de la funcion
x
8
6
4
2
0
-2
-4
-6
-8
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

```
figure;
plot(xx, abs(derf(xx) - yy))
xlabel('x')
ylabel('y')
title('Error derivada s2 aprox usando nodos equisdistantes')
```




```
disp(max(abs(derf(xx) - yy)))
```

```
0.0810
```