Análisis Númerico: Parcial 3

4 de marzo de 2025

Universidad Nacional de Colombia

Jorge Mauricio Ruiz Vera

Andrés David Cadena Simons acadenas@unal.edu.co
Sandra Natalia Florez Garcia sflorezga@unal.edu.co

Problema 1:

Dada la función

$$f(x) = x - \frac{1}{x} - 2, \quad f : \mathbb{R}^+ \to \mathbb{R},$$

se construye el siguiente algoritmo para aproximar la raíz $r = 1$:

$$x_{n+1} = 2 - \frac{1}{x_n}.$$

1. Verificar que si $x_0 > 1$ entonces la sucesión $\{x_n\}$ es monótona decreciente y acotada inferiormente por 1. Concluir que $x_n \to 1$ aunque esta iteración no esta en las hipótesis del teorema del punto fijo. ¿Qué hipótesis no se cumple?

Solución:

Dada la sucesión:

$$x_{n+1} = 2 - \frac{1}{x_n},$$

queremos demostrar que si $x_0 > 1$, entonces $\{x_n\}$ es monótona decreciente y acotada inferiormente por 1.
Demostramos por inducción que $x_n \geq 1$ para todo n:

- El caso base se tiene por hipótesis, ya que $x_0 > 1$.
- Ahora suponga que se tiene para n, es decir.
  Supongamos que $x_n \geq 1$. Veamos si $x_{n+1} \geq 1$:

  $$x_{n+1} = 2 - \frac{1}{x_n}.$$

  Como $x_n \geq 1$, se tiene $\frac{1}{x_n} \leq 1$, lo que implica que:

  $$x_{n+1} = 2 - \frac{1}{x_n} \geq 2 - 1 = 1.$$

  Por lo tanto, la sucesión es acotada inferiormente por 1.

Ahora afirmamos que $x_{n+1} \leq x_n$, ya que:

$$2 \leq x_n + \frac{1}{x_n}.$$

Para todo $x_n > 0$, en particular para todo $x_n > 1$ (los nuestros) ahora si reordenamos la desigualdad:

$$x_{n+1} = 2 - \frac{1}{x_n} \leq x_n$$

Problema 1 continúa en la siguiente página...

Por lo tanto, la sucesión es monótona decreciente.
Ahora, para ver la convergencia note que $\{x_n\}$ es monótona decreciente y acotada inferiormente por 1, por el teorema de monotonía, converge a un límite L. Tomando el límite en la ecuación de recurrencia:

$$L = 2 - \frac{1}{L}.$$

Multiplicamos por L:

$$\begin{aligned} L^2 - 2L + 1 &= 0, \\ (L-1)^2 = 0 \quad &\Rightarrow L = 1. \end{aligned}$$

Por lo tanto, $x_n \to 1$.
Ahora el teorema del punto fijo requiere que la función de iteración sea contractiva en un intervalo alrededor del punto fijo. La derivada de la función de iteración:

$$\varphi(x) = 2 - \frac{1}{x}$$

es

$$\varphi'(x) = \frac{1}{x^2}.$$

Evaluando en $x = 1$:

$$\varphi'(1) = 1.$$

Como $|\varphi'(1)| = 1$, la condición de contractividad ($|\varphi'(x)| < 1$ en un entorno del punto fijo) no se cumple, por lo que no se puede garantizar la convergencia por el teorema del punto fijo.

2. Dar un algoritmo para aproximar la raíz de f que converja cuadráticamente.

Solución:

Ahora veamos un algoritmo de convergencia cuadrática que nos permita aproximar la raíz.
Primero note que la raíz tiene multiplicidad 2, pues

$$\begin{aligned} x + \frac{1}{x} - 2 &= 0 \\ x^2 - 2x + 1 &= 0 \\ (x-1)^2 &= 0 \end{aligned}$$

Entonces, para obtener un algoritmo que converja cuadráticamente a la raíz $x = 1$,

podemos usar el **método de Newton modificado**:

$$x_{n+1} = x_n - \frac{f(x_n)f'(x_n)}{(f'(x_n))^2 - f(x_n)f''(x_n)}.$$

Calculamos $f(x) = x + \frac{1}{x} - 2$ y sus derivadas:

$$f'(x) = 1 - \frac{1}{x^2},$$
$$f''(x) = \frac{2}{x^3}.$$

Aplicamos la fórmula de Newton:

$$\begin{aligned} x_{n+1} &= x_n - \frac{\left(x_n + \frac{1}{x_n} - 2\right)\left(1 - \frac{1}{x_n^2}\right)}{\left(1 - \frac{1}{x_n^2}\right)^2 - \left(x_n - \frac{1}{x_n} - 2\right)\left(\frac{2}{x_n^3}\right)}, \\ &= x_n - \frac{(x_n^2 + 1 - 2x_n)x_n^2(x_n^2 - 1)}{(x_n^4 - 2x_n^2 + 1) - 2(x_n^2 + 1 - 2x_n)}. \end{aligned}$$

Este método tiene **convergencia cuadrática**, pues cumple que su raíz es de multiplicidad 2, $f \in C^2$ para $x > 0$, en particular para una vecindad de 1 y los únicos problemas de divergencia se encuentran directamente en la raíz 1 y 0, por lo que no tendremos que preocuparnos por fallos en las iteraciones, así podemos estar seguros de la convergencia cuadrática del método, lo que significa que el error disminuye aproximadamente como el cuadrado del error anterior en cada iteración.

Problema 2:

Sea

$$f(x) = (x - r_1)(x - r_2)\dots(x - r_d), \quad \text{donde} \quad r_1 < r_2 < \dots < r_d$$

Solución:

(a) Probar que si $x_0 > r_d$ la sucesión de Newton-Raphson converge a r_d.

Sea $f(x)$ la función definida por

$$f(x) = (x - r_1)(x - r_2)\cdots(x - r_d),$$

donde $r_1, r_2, \dots, r_d$ son raíces reales distintas y están ordenadas de manera creciente, es decir, $r_1 < r_2 < \dots < r_d$. Queremos analizar el comportamiento de la sucesión generada por el método de Newton-Raphson cuando el valor inicial x_0 es mayor que r_d y demostrar que converge a r_d.

Para aplicar el método de Newton-Raphson, primero derivamos la función $f(x)$. La derivada se obtiene como la suma de términos en los cuales se excluye un factor en cada producto:

$$f'(x) = \sum_{i=1}^{d} \prod_{j \neq i} (x - r_j).$$

El método de Newton-Raphson define la sucesión:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Sustituyendo las expresiones de $f(x)$ y $f'(x)$, se obtiene

$$x_{n+1} = x_n - \frac{(x_n - r_1)(x_n - r_2)\cdots(x_n - r_d)}{\sum_{i=1}^{d} \prod_{j \neq i} (x_n - r_j)}.$$

Ahora analizamos el comportamiento de esta sucesión cuando $x_0 > r_d$. En este caso, para todo i, se cumple que $x_0 - r_i > 0$, por lo que el producto $f(x_0)$ es positivo. Además, dado que cada término en la suma de $f'(x_0)$ también es positivo, se concluye que $f'(x_0) > 0$, lo que implica que el cociente $\frac{f(x_0)}{f'(x_0)} > 0$. Por lo tanto, la actualización en Newton-Raphson cumple que:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} < x_0.$$

Esto indica que la sucesión $\{x_n\}$ es estrictamente decreciente. Ahora probamos por inducción que $x_n > r_d$ para todo n. Para $n = 0$, se cumple por hipótesis que $x_0 > r_d$. Supongamos que $x_n > r_d$ y probemos que $x_{n+1} > r_d$.

Definimos la función auxiliar:

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

Problema 2 continúa en la siguiente página...

Se tiene que $g(r_d) = r_d$, ya que $f(r_d) = 0$. Además, podemos demostrar que $g(x)$ es una función creciente para $x > r_d$. Para ello, derivamos $g(x)$:

$$g'(x) = 1 - \frac{d}{dx}\left(\frac{f(x)}{f'(x)}\right).$$

Aplicando la regla del cociente,

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{[f'(x)]^2} = \frac{f(x)f''(x)}{[f'(x)]^2}.$$

Dado que para $x > r_d$, se cumple que $f(x) > 0$ y $f''(x) > 0$, se concluye que $g'(x) > 0$, lo que significa que $g(x)$ es estrictamente creciente en $x > r_d$. Como $x_n > r_d$, entonces $x_{n+1} = g(x_n) > g(r_d) = r_d$, completando la inducción.

Finalmente, observamos que la sucesión $\{x_n\}$ es monótona decreciente y está acotada inferiormente por r_d, por lo que converge a un límite $L \geq r_d$. Evaluando el límite en la ecuación de Newton-Raphson,

$$L = L - \frac{f(L)}{f'(L)}.$$

Dado que $f'(L) > 0$, se deduce que $f(L) = 0$, lo que implica que L es una raíz de $f(x)$. Como r_d es la única raíz mayor o igual a r_d, concluimos que $L = r_d$, demostrando que la sucesión de Newton-Raphson converge a r_d.

Este resultado se tiene porque la función auxiliar $g(x) = x - \frac{f(x)}{f'(x)}$, que define la iteración de Newton-Raphson, es continua para $x > r_d$, garantizando que el comportamiento monótono de la sucesión se mantenga y que su límite sea efectivamente r_d.

(b) Para un polinomio,

$$P(x) = a_d x^d + \cdots + a_0, \quad a_d \neq 0,$$

tal que sus d raíces son reales y distintas, se propone el siguiente método que aproxima los valores de todas sus raíces:

(a) Se comienza con un valor x_0 mayor que

$$M = \text{máx}\left(1, \sum_{i=0}^{d-1} \frac{|a_i|}{|a_d|}\right).$$

(Nota: M es una cota para el módulo de todas las raíces del polinomio).

(b) Se genera a partir de x_0 la sucesión de Newton-Raphson, que, según el ítem anterior, converge a la raíz más grande de P, llamémosla r_d; obteniéndose de este modo un valor aproximado $\tilde{r}_d$.

(c) Se divide P por $x - \tilde{r}_d$ y se desprecia el resto, dado que $r_d \approx \tilde{r}_d$. Se redefine ahora P como el resultado de esta división y se comienza nuevamente desde el primer ítem, para hallar las otras raíces.

Problema 2 continúa en la siguiente página...

Aplicar este método para aproximar todas las raíces del polinomio

$$P(x) = 2x^3 - 4x + 1.$$

Solución:
A continuación, se implementará el algoritmo descrito en MATLAB para el polinomio $P(x) = 2x^3 - 4x + 1$, utilizando el método de Newton-Raphson combinado con división polinómica para encontrar sus raíces reales.

```
% Programa para encontrar todas las raíces reales de un polinomio
    usando el método descrito

% Definir el polinomio P(x) mediante sus coeficientes en orden
    descendente
% Ejemplo: P(x) = 2x^3 -4x +1 (raíces: 1, 2, 3)
original_coeffs = [2, 0, -4, 1]; % Guardar coeficientes originales

% Parámetros
tol = 1e-6;        % Tolerancia para la convergencia de
    Newton-Raphson
max_iter = 100;    % Máximo número de iteraciones por raíz

% Inicializar vectores
d = length(original_coeffs) - 1;     % Grado inicial del polinomio
current_coeffs = original_coeffs;    % Coeficientes del polinomio
    actual
roots_approx = zeros(1, d);          % Almacenar las raíces
    aproximadas

% Bucle para encontrar todas las raíces
for k = d:-1:1
    % Verificar que current_coeffs sea válido
    if isempty(current_coeffs) || length(current_coeffs) < 2
        error('Error: Coeficientes inválidos en iteración %d', k);
    end

    % Paso (a): Calcular la cota M
    ad = abs(current_coeffs(1));          % Coeficiente principal
        |a_d|
    sum_abs = sum(abs(current_coeffs(2:end))) / ad;  % Sum
        |a_i|/|a_d|
    M = max(1, sum_abs);
    x0 = M + 1;  % Elegir x0 mayor que M

    fprintf('Iteración %d: M = %f, x0 = %f\n', k, M, x0);
```

```
30
31      % Paso (b): Método de Newton-Raphson
32      x = x0;
33      for iter = 1:max_iter
34          % Evaluar P(x) y P'(x)
35          Px = polyval(current_coeffs, x);
36          P_prime_coeffs = polyder(current_coeffs);
37          P_prime_x = polyval(P_prime_coeffs, x);
38
39          % Verificar derivada no nula
40          if abs(P_prime_x) < eps
41              error('Derivada cercana a cero en x = %f, iteración
                    %d', x, k);
42          end
43
44          % Actualizar x
45          x_new = x - Px / P_prime_x;
46
47          % Comprobar convergencia
48          if abs(x_new - x) < tol
49              break;
50          end
51          x = x_new;
52
53          if iter == max_iter
54              warning('Máximo de iteraciones alcanzado para la raíz
                    %d.', k);
55          end
56      end
57
58      % Guardar la raíz aproximada
59      r_tilde = x_new;
60      roots_approx(k) = r_tilde;
61
62      % Paso (c): Dividir P(x) por (x - r_tilde)
63      [Q, R] = deconv(current_coeffs, [1, -r_tilde]);
64      current_coeffs = Q;  % Redefinir P(x) como el cociente
65
66      fprintf('Raíz %d aproximada: %f\n', k, r_tilde);
67  end
68
69  % Mostrar resultados
70  disp('Raíces aproximadas del polinomio:');
71  disp(roots_approx);
```

```
72
73 % Verificación con raíces exactas usando los coeficientes
       originales
74 if ~isempty(original_coeffs)
75     exact_roots = roots(original_coeffs);
76     disp('Raíces exactas (para comparación):');
77     disp(sort(exact_roots)); % Ordenadas de menor a mayor
78 else
79     disp('No se pueden calcular raíces exactas: coeficientes
           originales inválidos.');
80 end
```

La implementación del algoritmo en MATLAB para el polinomio $P(x) = 2x^3 - 4x + 1$ produce los siguientes resultados:

$$\begin{aligned} \text{Iteración 3:} \quad & M = 2{,}500000, \quad x_0 = 3{,}500000 \\ & \text{Raíz 3 aproximada: } 1{,}267035 \\ \text{Iteración 2:} \quad & M = 1{,}661657, \quad x_0 = 2{,}661657 \\ & \text{Raíz 2 aproximada: } 0{,}258652 \\ \text{Iteración 1:} \quad & M = 1{,}525687, \quad x_0 = 2{,}525687 \\ & \text{Raíz 1 aproximada: } -1{,}525687 \end{aligned}$$

Las raíces aproximadas del polinomio son:

$$-1{,}5257 \quad 0{,}2587 \quad 1{,}2670$$

Para comparación, las raíces exactas son:

$$\begin{gathered} -1{,}5257 \\ 0{,}2587 \\ 1{,}2670 \end{gathered}$$

Por consiguiente, el método propuesto es una estrategia eficiente y ordenada para determinar todas las raíces reales de un polinomio. Al iniciar con un valor $x_0 > M$, donde M es una cota basada en los coeficientes del polinomio, se garantiza la convergencia de la sucesión de Newton-Raphson hacia la raíz más grande. Posteriormente, al dividir el polinomio por $(x - \tilde{r}_d)$, el problema se reduce de manera progresiva, transformándolo en una secuencia de cálculos más simples y asegurando que cada raíz se encuentre de manera estructurada.

Además, este enfoque aprovecha la convergencia cuadrática de Newton-Raphson, lo que permite obtener aproximaciones precisas en pocas iteraciones. A diferencia de otros

métodos como la bisección, que requiere definir intervalos específicos, aquí se trabaja con una cota general basada en los coeficientes del polinomio, lo que lo hace práctico incluso sin un análisis previo. Su eficiencia y precisión lo convierten en una herramienta útil para calcular raíces de manera sistemática y sin depender de estimaciones iniciales cercanas a cada una.

Problema 3:

Sea $f \in C[a,b]$, y sean $x_0 = a, x_1 = a+h, \cdots, x_n = b$, donde $h = \frac{b-a}{n}$. Considerar la poligonal $l(x)$ que interpola a f en los puntos $x_i, i = 0, 1, \cdots, n$. Probar que

1.
$$|f(x) - l(x)| \leq \frac{h^2}{2} \max_{x\in[a,b]} |f''(x)|$$

Solución:

Suponga el intervalo $[x_i, x_{i+1}]$, entonces

$$l(x) = \frac{f(x_{i+1}) - f(x_i)}{h}(x - x_i) + f(x_i),$$

luego

$$l(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{f''(\xi_1)}{2}(x_{i+1} - x_i)(x - x_i)$$

de dónde podemos deducir

$$f(x) - l(x) = (f(x) - f(x_i)) - (f'(x_i)(x - x_i) + \frac{f''(\xi_1)}{2}(x_{i+1} - x_i)(x - x_i)$$

Aplicando el teorema de Taylor en $f(x)$

$$f(x) - l(x) = f'(x_i)(x - x_i) + \frac{f''(\xi_2)}{2}(x - x_i)^2$$

de lo que se puede afirmar que

$$\begin{aligned} f(x) - f(x_i) &= \frac{f''(\xi_2)}{2}(x - x_i)^2 - \frac{f''(\xi_1)}{2}(x_{i+1} - x_i)(x - x_i) \\ &= \frac{(x - x_i)}{2}\left(f''(\xi_2)(x - x_i) - f''(\xi_1)(x_{i+1} - x_i)\right) \quad \text{Pero como } f \in C^2[a,b]. \\ &= \frac{(x - x_i)}{2} f''(\xi)((x - x_i) - (x_{i+1} - x_i)) \\ &= \frac{f''(\xi)}{2}(x - x_i)(x - x_{i+1}) \end{aligned}$$

luego usando esto se puede concluir que

$$\begin{aligned} |f(x) - l(x)| &= \left|\frac{f''(\xi)}{2}(x - x_i)(x - x_{i+1})\right| \\ &= \frac{h^2}{2} \max_{x\in[a,b]} |f''(\xi)| \end{aligned}$$

lo que concluye el resultado.

Problema 3 continúa en la siguiente página...

2.

$$|f'(x) - l'(x)| \leq h \max_{x\in[a,b]} |f''(x)|.$$

Solución:

Recordemos que $l(x) = \frac{f(x_{i+1})-f(x_i)}{h}(x - x_i) + f(x_i)$, luego $l'(x) = \frac{f(x_{i+1})-f(x_i)}{h} = f'(c)$ para algún $c \in [x_i, x_{i+1}]$. Ahora, aplicando el teorema de Taylor se tiene que:

$$f'(x) = f'(x_i) + f''(\xi_1)(x - x_i),$$

de lo que se sigue que

$$\begin{aligned} |f'(x) - l'(x)| &= |f'(x_i) + f''(\xi_1)(x - x_i) - f'(c)| \\ &= |f'(x_i) + f''(\xi_1)(x - x_i) - f'(x_i) - f''(\xi_2)(x - x_i)| \\ &= |f''(\xi_1)(x - x_i) - f''(\xi_2)(c - x_i)| \\ &\leq h \max_{x\in[a,b]} |f''| \end{aligned}$$

lo que concluye el resultado.

Problema 4:

(Silueta de la mano) Para dibujar la silueta de su mano, siga los siguientes pasos:

a) Preparamos una tabla de abcisas y ordenadas usando los siguientes comandos de Matlab:

```
figure('position',get(0,'screensize'))
axes('position',[0 0 1 1])
[x,y] = ginput;
```

b) Dibuje su mano en un papel y póngalo sobre la pantalla del computador. Use el ratón para seleccionar alrededor de 37 puntos que delineen su mano. Termine la instrucción `ginput` oprimiendo enter.

c) Grafique los puntos (x, y) obtenidos y la mano correspondiente mediante el comando `plot` de Matlab.

d) Implemente el método de splines cúbicos.

e) Interpole por separado los puntos (i, x_i) e (i, y_i) mediante splines cúbicos usando su programa.

f) Grafique la curva parametrizada que se obtiene.

g) Estime el área de su mano usando la fórmula del área de Gauss:

$$A = \frac{1}{2}\left|\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n\right|$$

Solución:

Para construir los splines cúbicos personalizados, primero se parametrizaron las coordenadas en función de un parámetro auxiliar t, que en este caso corresponde a la enumeración de los puntos dados. Esto se debe a que la curva no es una función, por ende se describe mediante las coordenadas $(x(t), y(t))$. Luego, se planteó un sistema de ecuaciones tridiagonal para determinar los coeficientes necesarios para la interpolación spline.

El sistema tridiagonal que se resuelve tiene la forma:

$$\begin{bmatrix} a_1 & b_1 & 0 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & 0 & \cdots & 0 \\ 0 & b_2 & a_3 & b_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & b_{n-3} & a_{n-2} & b_{n-2} \\ 0 & 0 & \cdots & 0 & b_{n-2} & a_{n-1} \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \vdots \\ \sigma_{n-2} \\ \sigma_{n-1} \end{bmatrix} = 6 \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{bmatrix}$$

donde:

Problema 4 continúa en la siguiente página...

- σ_k representa los coeficientes de los polinomios de grado tres.
- a_k son los elementos de la diagonal principal y se definen como:

$$a_k = 2(h_k + h_{k+1})$$

  En este caso, como se parametrizó con la enumeración, se tiene $h_k = 1$, por lo que $a_k = 4$.

- b_k son los elementos de las diagonales superior e inferior:

$$b_k = h_k$$

  Dado que $h_k = 1$, entonces $b_k = 1$.

- d_k es el término independiente, calculado con diferencias finitas de segundo orden:

$$d_k = \frac{y_{k+1} - y_k}{h_k} - \frac{y_k - y_{k-1}}{h_{k-1}}$$

  Como $h_k = 1$, la expresión se simplifica a:

$$d_k = (y_{k+1} - y_k) - (y_k - y_{k-1})$$

Este sistema se resuelve **de manera independiente para las coordenadas x y y**, permitiendo obtener los valores de σ_k que luego se emplean para construir los polinomios cúbicos de interpolación. La solución del sistema se realiza mediante el **método de Thomas**, una técnica eficiente para resolver sistemas tridiagonales. Con los valores obtenidos, se emplean los polinomios cúbicos de interpolación en cada subintervalo, generando así una curva suave que pasa por los puntos dados.

```
    % Configurar la figura para la entrada de puntos
figure('position', get(0, 'screensize'))
axes('position', [0 0 1 1])
disp('Haga clic en varios puntos y presione Enter cuando termine');

[x, y] = ginput(); % Obtener puntos del usuario
x = [x; x(1)]; % Cerrar el polígono
y = [y; y(1)];

plot(x, y, 'bo-', 'LineWidth', 1.5, 'MarkerSize', 8);
hold on;
disp([x, y]);

% Verificación de puntos mínimos
if length(x) < 4
```

```
    error('Se necesitan al menos 4 puntos para la interpolación spline cúbica');
end

% Parametrización
t = 1:length(x);
t_fine = linspace(1, length(x), 100); % Puntos finos para suavidad

% Interpolación con spline de MATLAB
x_spline = spline(t, x, t_fine);
y_spline = spline(t, y, t_fine);
plot(x_spline, y_spline, 'g--', 'LineWidth', 1.5, 'DisplayName', 'Spline de MATLAB');

% Número de incógnitas en el sistema tridiagonal
n = length(x) - 2;

% Calcular segundas diferencias (dx, dy)
dx = 6 * (x(3:end) - 2*x(2:end-1) + x(1:end-2));
dy = 6 * (y(3:end) - 2*y(2:end-1) + y(1:end-2));

% Vectores de la matriz tridiagonal
a = ones(n-1, 1);    % Subdiagonal
b = 4 * ones(n, 1);  % Diagonal principal
c = ones(n-1, 1);    % Superdiagonal

% Resolver el sistema tridiagonal para sigma_x y sigma_y
sigma_x_interior = thomas_solver(a, b, c, dx);
sigma_y_interior = thomas_solver(a, b, c, dy);

% Agregar condiciones de frontera (sigma = 0 en extremos)
sigma_x = [0; sigma_x_interior; 0];
sigma_y = [0; sigma_y_interior; 0];

% Interpolación personalizada usando splines cúbicos
x_int = [];
y_int = [];
for i = 1:length(x)-1
    t_i = linspace(t(i), t(i+1), 10);
    for k = 1:length(t_i)
        x_val = spline_polinomios(t_i(k), t(i), t(i+1), x(i), x(i+1), sigma_x(i), sigma_x(i+1));
        y_val = spline_polinomios(t_i(k), t(i), t(i+1), y(i), y(i+1), sigma_y(i), sigma_y(i+1));
```

```
56          x_int = [x_int, x_val];
57          y_int = [y_int, y_val];
58      end
59  end
60
61  % Graficar resultado personalizado
62  plot(x_int, y_int, 'r-', 'LineWidth', 1.5, 'DisplayName', 'Spline
        Personalizado');
63
64  % Cálculo del área con la fórmula de Gauss
65  n_points = length(x);
66  sum1 = 0;
67  sum2 = 0;
68  for i = 1:n_points-1
69      sum1 = sum1 + x(i) * y(i+1);
70      sum2 = sum2 + x(i+1) * y(i);
71  end
72  sum1 = sum1 + x(n_points) * y(1); % Cerrar polígono
73  sum2 = sum2 + x(1) * y(n_points);
74
75  Ar = 0.5 * abs(sum1 - sum2);
76  disp(['Área calculada: ', num2str(Ar)]);
77
78  % Configurar la gráfica
79  title('Interpolación Spline Cúbica');
80  xlabel('X'); ylabel('Y');
81  grid on;
82  legend('Puntos Originales', 'Spline de MATLAB', 'Spline
        Personalizado', 'Location', 'best');
83  hold off;
84
85  area_correcta = polyarea(x, y);
86  disp(['Área con polyarea: ', num2str(area_correcta)]);
87
88  % --- Función para resolver sistemas tridiagonales con el algoritmo
        de Thomas ---
89  function x = thomas_solver(a, b, c, d)
90      n = length(d);
91      cp = c;
92      dp = d;
93      bp = b;
94
95      % Eliminación hacia adelante
96      for i = 2:n
```

```
97          m = a(i-1) / bp(i-1);
98          bp(i) = bp(i) - m * cp(i-1);
99          dp(i) = dp(i) - m * dp(i-1);
100     end
101
102     % Sustitución hacia atrás
103     x = zeros(n, 1);
104     x(n) = dp(n) / bp(n);
105     for i = n-1:-1:1
106         x(i) = (dp(i) - cp(i) * x(i+1)) / bp(i);
107     end
108 end
109
110 % --- Función de interpolación spline cúbica ---
111 function q = spline_polinomios(t, t_k, t_k1, y_k, y_k1, sigma_1,
        sigma_2)
112     term1 = (sigma_1 / 6) * ((t_k1 - t)^3 - (t_k1 - t));
113     term2 = (sigma_2 / 6) * ((t - t_k)^3 - (t - t_k));
114     term3 = y_k * (t_k1 - t);
115     term4 = y_k1 * (t - t_k);
116     q = term1 + term2 + term3 + term4;
117 end
```

La implementación del código da un área de $0{,}10296u^2$ para la mano de Sandra y un área de $0{,}09439u^2$ para la de Andrés. A continuación, se comparan los resultados gráficos:

Figura 1: Spline de la mano de Sandra.

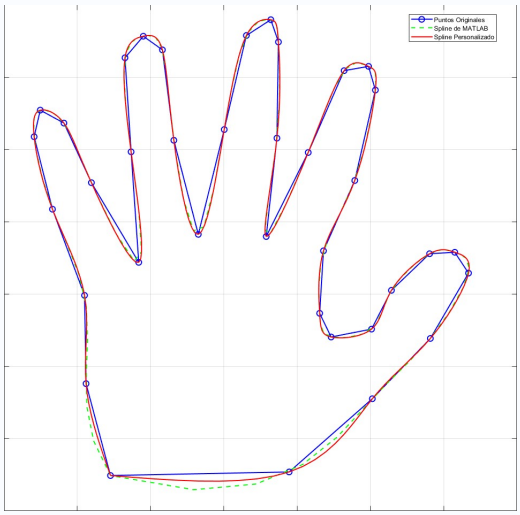


Figura 2: Spline de la mano de Andrés.

Los gráficos muestran la interpolación realizada mediante splines cúbicos. En cada imagen, los trazos azules representan los puntos originales, la curva roja corresponde al spline cúbico implementado manualmente y la curva verde representa la interpolación generada por MATLAB.

La interpolación realizada con splines cúbicos genera una aproximación suave y continua a partir de los puntos dados, evitando las oscilaciones bruscas que podrían aparecer con otros métodos, como la interpolación polinómica global. La implementación personalizada resuelve un sistema tridiagonal para obtener las segundas derivadas en los puntos de control, garantizando una transición fluida entre segmentos. Esto se traduce en una curva interpolada que respeta la forma general de los datos originales sin generar artefactos inesperados.

Al comparar la interpolación con el spline de MATLAB, se pueden notar ligeras diferencias en ciertos tramos. Esto puede deberse a la forma en que cada método maneja las condiciones de frontera y distribuye los puntos interpolados. MATLAB emplea una parametrización interna que puede afectar la forma final de la curva, mientras que la interpolación personalizada usa una parametrización directa basada en la posición secuencial de los puntos. Como resultado, aunque ambas curvas siguen la misma estructura general, hay pequeñas variaciones en la forma en que se suavizan ciertas transiciones. Estas diferencias pueden observarse con mayor claridad en la siguiente figura:

Problema 4 continúa en la siguiente página...

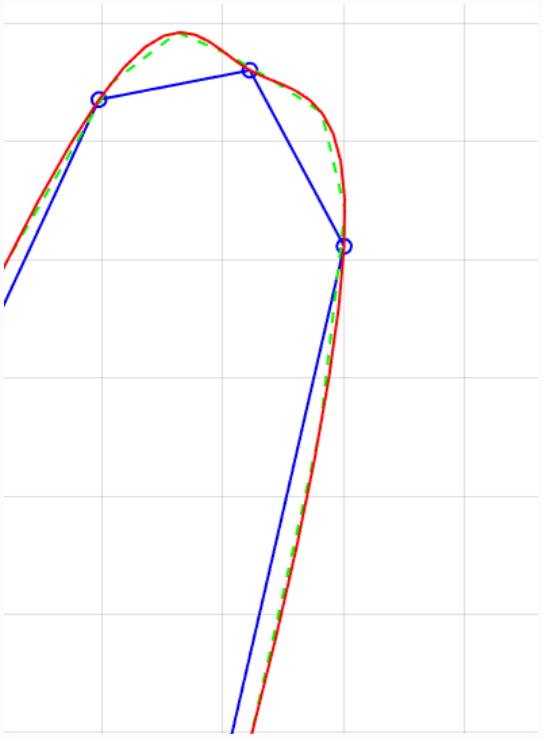
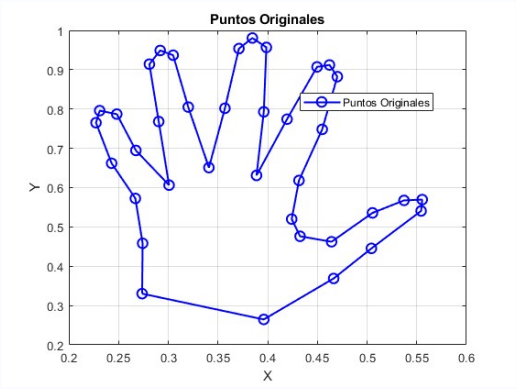


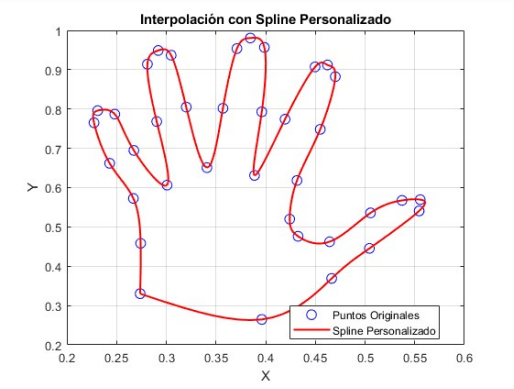
Figura 3: Vista ampliada de la gráfica en un dedo.

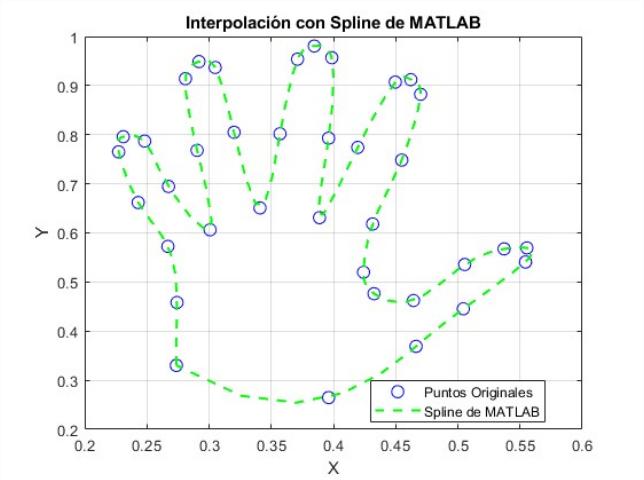
En términos de exactitud, ambas interpolaciones ofrecen una aproximación adecuada a la forma original. La interpolación manual proporciona una representación fiel y coherente con los datos ingresados, manteniendo una curva suave que se ajusta a la estructura esperada. Aunque hay pequeñas diferencias con MATLAB, estas no implican una pérdida de precisión significativa, sino más bien una variación en la forma en que cada método interpola los segmentos individuales.

Para visualizar mejor cada interpolación, se presentan las tres gráficas por separado. En estas representaciones individuales, se puede apreciar con mayor claridad cómo se ajusta cada spline a los puntos originales.

Problema 4 continúa en la siguiente página...


Puntos Originales
Puntos Originales
X
Y


Interpolación con Spline Personalizado
Puntos Originales
Spline Personalizado
X
Y


Interpolación con Spline de MATLAB
Puntos Originales
Spline de MATLAB
X
Y

Problema 5:

Observe que

$$I = \int_0^1 \frac{4}{1+x^2} = \pi$$

1. Use las reglas compuestas del punto medio, del trapecio y de Simpson para aproximar I para varios tamaños de paso de integración $h = \frac{1}{n}, n = 10, 50, 100, 250, 500, 1000, 1500, 2000$. Grafique el logaritmo del error absoluto versus n para cada paso. Describa el efecto de redondeo de los errores cuando $h \to 0$.

Solución:

Veamos lo que sucede para cada caso:

- Punto Medio.
  Note que al aplicar el algoritmo para la integral obtenemos la siguiente gráfica de errores:

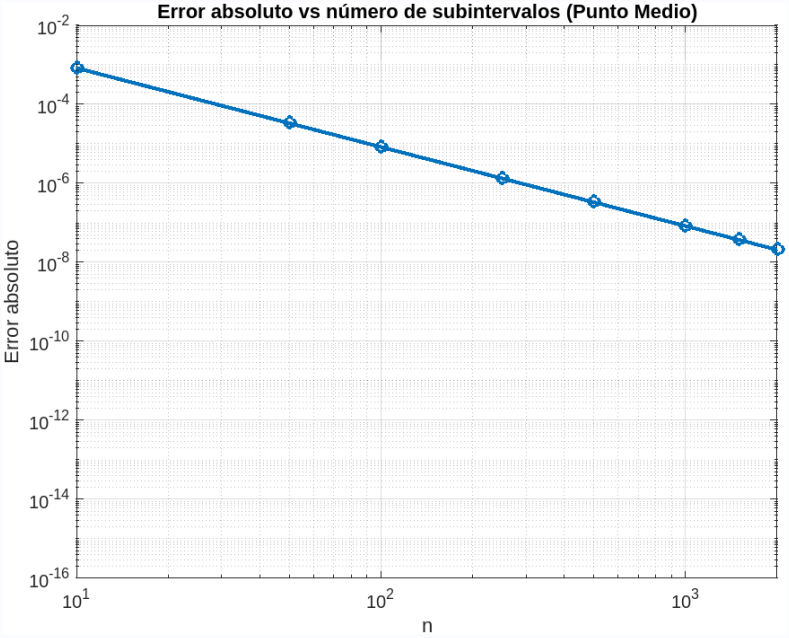


Figura 4: Error absoluto en el método del punto medio.

Aquí cabe recordar que el método del punto medio tiene convergencia de orden $O(h^2)$ y su implementación es bastante sencilla, la podremos ver en el siguiente código que aplicando el método se tomo 0,001995 segundos.

Problema 5 continúa en la siguiente página...

```
format rational

% Creamos la función f
f = @(x) 4 ./ (1 + x.^2);

% Parámetros de la integral y el método
a = 0;
b = 1;
valores_n = [10, 50, 100, 250, 500, 1000, 1500, 2000];
I_exacto = pi;

% Se crea un vector para guardar los errores
errores = zeros(size(valores_n));

% Método del punto medio
tic;
for k = 1:length(valores_n)
    n = valores_n(k);
    % Aquí se encuentra el tamaño del intervalo
    h = (b - a) / n;
    % Se calculan los puntos medios y los guarda en el vector
        x_medios
    x_medios = a + ((2*(0:1:n-1)+1)*h)/2;
    % Regla del punto medio compuesta
    I_punto_medio = h * sum(f(x_medios));
    % Calculo del error
    errores(k) = abs(I_punto_medio - I_exacto);
end
tiempo = toc

% Graficar log-log del error vs n
figure;
loglog(valores_n, errores, '-o', 'LineWidth', 2);
xlabel('n');
ylabel('Error absoluto');
ylim([1e-16, 1e-2]);
title('Error absoluto vs número de subintervalos (Punto
    Medio)');
grid on;
```

- Trapecio.
  Note que al aplicar el algoritmo para la integral obtenemos la siguiente gráfica de errores:

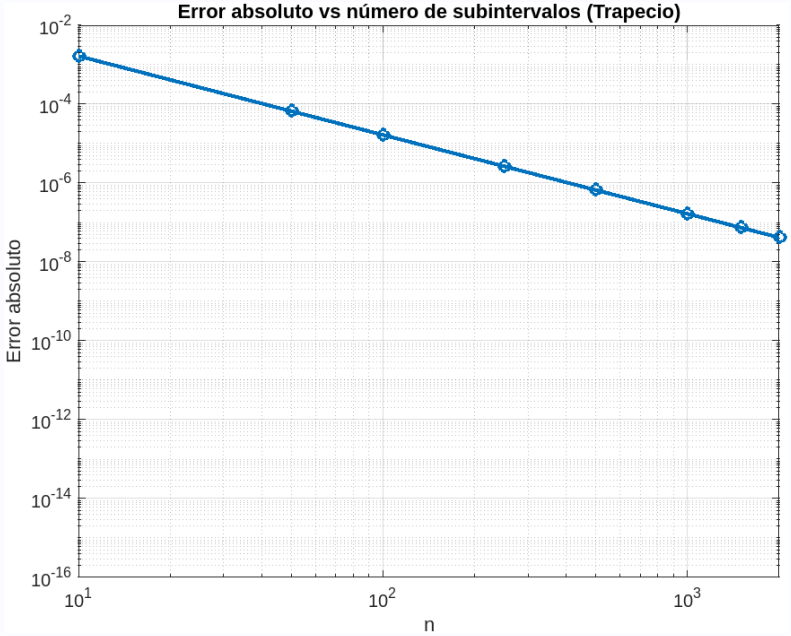


Figura 5: Error absoluto en el método del trapecio.

Aquí cabe recordar que el método del trapecio tiene convergencia de orden $O(h^2)$ y su implementación es (en cuestiones de operaciones) sencilla, la podremos ver en el siguiente código que aplicando el método se tomo 0,002719 segundos.

```
format rational

% Creamos la función f
f = @(x) 4 ./ (1 + x.^2);

% Parámetros de la integral y el método
a = 0;
b = 1;
valores_n = [10, 50, 100, 250, 500, 1000, 1500, 2000];
I_exacto = pi;

% Se crea un vector para guardar los errores
errores_trapecio = zeros(size(valores_n));

% Método del trapecio
for k = 1:length(valores_n)
    n = valores_n(k);
    h = (b - a) / n;
    % Extremos de la partición
    x_extremos = a:h:b;
    m = length(x_extremos);

    % Aplicar la regla del trapecio compuesta
    I_trapecio = (h/2) * (f(a) + 2*sum(f(x_extremos(2:m-1)))
        + f(b));

    % Calculo del error
    errores_trapecio(k) = abs(I_trapecio - I_exacto);
end

% Graficar error del método del trapecio
figure;
loglog(valores_n, errores_trapecio, '-o', 'LineWidth', 2);
xlabel('n');
ylabel('Error absoluto');
ylim([1e-16, 1e-2]);
title('Error absoluto vs número de subintervalos (Trapecio)');
grid on;
```

- Simpson.
  Note que al aplicar el algoritmo para la integral obtenemos la siguiente gráfica de errores:

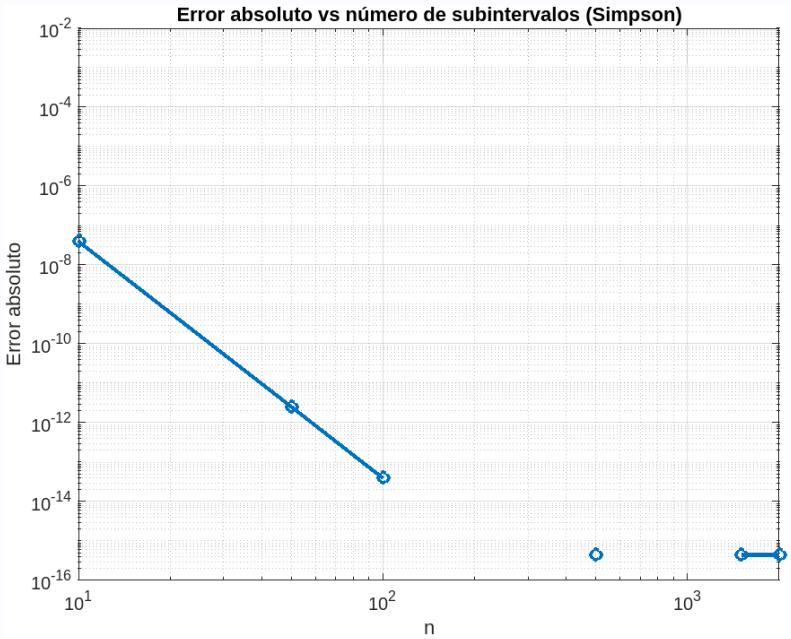


Figura 6: Error absoluto en el método de Simpson.

Aquí cabe recordar que el método tiene convergencia de orden $O(h^4)$ y su implementación es aunque un poco más compleja en la teoría, sigue siendo eficiente cuando hablamos computacionalmente, la podremos ver en el siguiente código que aplicando el método se tomo 0,003878 segundos.

```
format rational
% Creamos la función f
f = @(x) 4 ./ (1 + x.^2);
% Parámetros de la integral y el método
a = 0;
b = 1;
valores_n = [10, 50, 100, 250, 500, 1000, 1500, 2000];
I_exacto = pi;
% Se crea un vector para guardar los errores
errores_simpson = zeros(size(valores_n));
% Método de Simpson
tic;
for k = 1:length(valores_n)
    n = valores_n(k);
    % Aquí nos aseguramos que n sea par (Simpson necesita n
        par)
    if mod(n, 2) == 1
        n = n + 1;
    end
    h = (b - a) / n;
    % Extremos de la partición
    x_extremos = a:h:b;
    m = length(x_extremos);
    % Aplicar la regla de Simpson compuesta
    I_simpson = (h/3) * (f(a) + 4*sum(f(x_extremos(2:2:m-1)))
        + 2*sum(f(x_extremos(3:2:m-2))) + f(b));
    % Calculo del error
    errores_simpson(k) = abs(I_simpson - I_exacto);
end
tiempo = toc
% Graficar error del método de Simpson
figure;
loglog(valores_n, errores_simpson, '-o', 'LineWidth', 2);
xlabel('n');
ylabel('Error absoluto');
ylim([1e-16, 1e-2]);
title('Error absoluto vs número de subintervalos (Simpson)');
grid on;
```

Problema 5 continúa en la siguiente página...

Ahora sería interesante ver la comparación entre los 3 métodos en la siguiente figura y la tabla de tiempos:

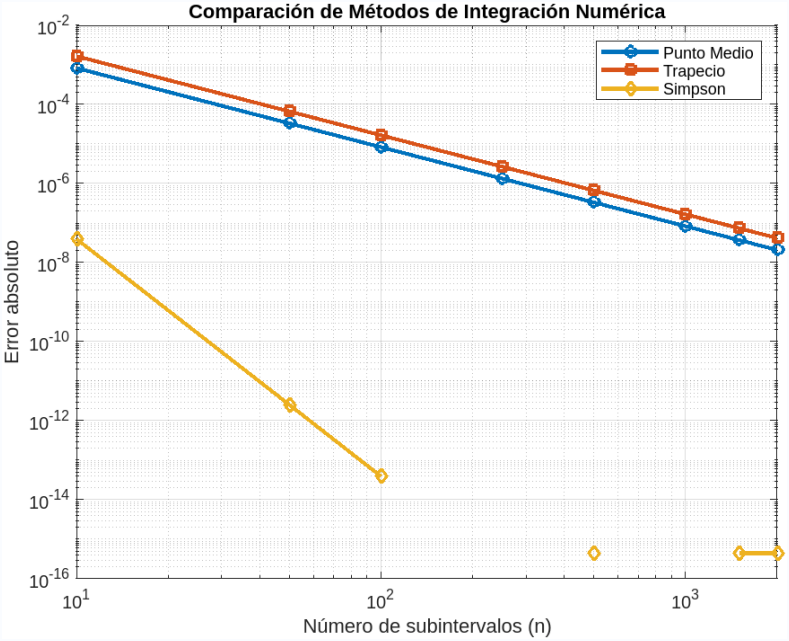


Figura 7: Comparación de los métodos.

Método	Tiempo (s)
Punto medio	$0,001995$
Trapecio	$0,002719$
Simpson	$0,003878$

De lo que podemos concluir lo siguiente:

- En la aplicación el método del punto medio aparenta ser superior al método del trapecio en cuestiones de errores y tiempo, no obstante la diferencia entre estos aparenta ser bastante baja para tenerlo en cuenta a la hora de realizar los cálculos y favorecer alguna de las 2, por lo que no se descarta que el uso del método del trapecio sea más efectivo en otro tipos de problemas en los que quizás sea mejor considerar los términos de borde en la función.
- Es claro que el método de Simpson es mucho más preciso al momento de aproximar el resultado, pues, desde la propia teoría (la convergencia en orden $O(h^4)$) vimos que al refinar la malla el error se reduciría bastante rápido, por lo que vemos que rápidamente con tomar $n = 250$ ya este había superado la precisión numérica de

maquina al ser tan cercano a 0 el error absoluto, situación que con los otros 2 métodos no se alcanzó a concluir de igual forma.

- Si bien en cuestión de tiempo vemos que aún podríamos permitirles a los métodos tener un refinamiento de malla aún más grande, es claro que al método de Simpson le fue suficiente con muy poco y que a los restantes 2 métodos aparenta aún faltarle mucho al refinamiento de la malla para alcanzar la precisión numérica que tiene el método de Simpson.

- Hablando sobre el efecto de redondeo de los errores cuando $h \to 0$ podemos ver mucho más claro lo que sucede en el método Simpson, pues como antes comentamos, cuando $n = 250$ ya alcanza el límite de precisión de maquina, no obstante en otros momentos vemos que el error se eleva, por lo que sería interesante ver que sucede con esto, veamos los errores imprimidos por el código.
  errores simpson =1.0e-07, *, 0.396505779320933, 0.000025401902803, 0.000000390798505, 0, 0.000000004440892, 0, 0.000000004440892, 0.000000004440892
  Podemos observar que, conforme h disminuye, el error de redondeo comienza a jugar un papel crucial. En particular, en el método de Simpson, notamos que el error se reduce progresivamente hasta alcanzar el límite de precisión de la máquina, momento en el cual se observa un error de 0. Sin embargo, a pesar de llegar a este aparente çero", el error vuelve a incrementarse en pasos posteriores. Este comportamiento se debe a la acumulación de errores de redondeo en los cálculos numéricos. A medida que h se hace más pequeño, el número de operaciones aritméticas necesarias para evaluar la integral aumenta, lo que introduce errores debidos a la precisión finita de la máquina. En consecuencia, en lugar de seguir disminuyendo indefinidamente, el error empieza a oscilar e incluso a aumentar en ciertos puntos debido a la pérdida de precisión numérica. En resumen, aunque disminuir h en principio reduce el error de truncamiento del método, llega un punto en el que el error de redondeo se vuelve dominante, lo que limita la precisión alcanzable en los cálculos numéricos.

2. Implemente el método de integración de Romberg para calcular I. Grafiqué el logaritmo del error en los términos diagonales en la tabla de extrapolación versus $log(h)$. Verifique sus resultados con la teoría.

Solución:

Note que al aplicar el algoritmo para la integral obtenemos la siguiente gráfica de errores:

Problema 5 continúa en la siguiente página...

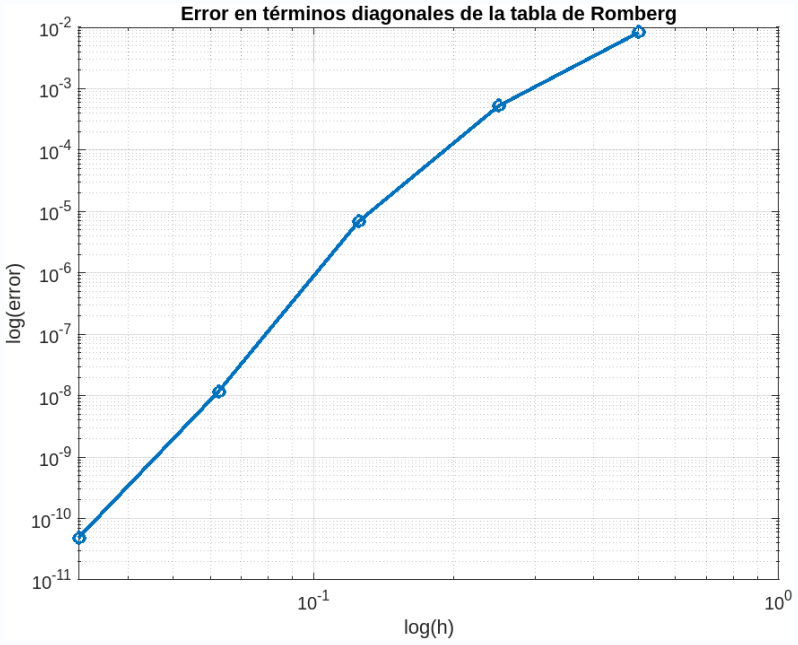


Figura 8: Error en los términos diagonales en la tabla de extrapolación versus $logh$.

El cuál se tardó 0,004838 segundos en ejecutarse.

De aquí podemos concluir lo siguiente:

- Respecto a la teoría, en efecto a valores pequeños de h mejora la aproximación del valor de la integral (más refinamiento, mejor aproximación).
- Note que la curva es çasiüna recta, de lo cuál podemos deducir al estar en escala logaritmica que el error $E(h) \approx Ch^p$ en donde el valor de p aumenta cada vez más con cada nivel de la extrapolación.
- Se alcanzan errores de hasta 10^{-10}, lo que indica que el método logra una precisión bastante alta con relativamente pocas iteraciones.

A continuación podemos ver la matriz:

$$\begin{pmatrix} 3{,}00000000 & 0 & 0 & 0 & 0 & 0 \\ 3{,}10000000 & 3{,}13333333 & 0 & 0 & 0 & 0 \\ 3{,}13117647 & 3{,}14156863 & 3{,}14211765 & 0 & 0 & 0 \\ 3{,}13898849 & 3{,}14159250 & 3{,}14159409 & 3{,}14158578 & 0 & 0 \\ 3{,}14094161 & 3{,}14159265 & 3{,}14159266 & 3{,}14159264 & 3{,}14159267 & 0 \\ 3{,}14142989 & 3{,}14159265 & 3{,}14159265 & 3{,}14159265 & 3{,}14159265 & 3{,}14159265 \end{pmatrix}$$

Y en la siguiente página se podrá encontrar el código.

Problema 5 continúa en la siguiente página...

```
format long
% Definimos la función
f = @(x) 4 ./ (1 + x.^2);
% Límites de integración
a = 0;
b = 1;
% Valor exacto de la integral
I_exacto = pi;
% Tolerancia
eps = 1e-8;
% Calculamos la integral con Romberg y obtenemos errores
tic;
[I_romberg, hs, errores] = romberg(f, a, b, I_exacto, eps);
toc
% Graficamos log(error) vs log(h)
figure;
loglog(hs, errores, '-o', 'LineWidth', 2);
xlabel('log(h)');
ylabel('log(error)');
title('Error en términos diagonales de la tabla de Romberg');
grid on;
% --- FUNCIONES ---
function [I_romberg, hs, errores] = romberg(f, a, b, I_exacto,
    eps)
    if nargin < 5
        % Precisión por defecto
        eps = 1e-8;
    end
    % Matriz de Romberg
    R = zeros(1, 1);
    % Primer trapecio
    R(1,1) = 0.5 * (b - a) * (f(a) + f(b));
    print_row(R(1,1));
    % Inicialización de vectores para la gráfica
    hs = [];
    errores = [];
    % Número de iteraciones
    n = 1;
    while true
        % Refinamos h
        h = (b - a) / 2^n;
        % Nueva regla del trapecio
        R(n+1,1) = 0.5 * R(n,1) + h * sum(f(a + (2*(1:2^(n-1)) -
            1) * h));
```

```
43          % Extrapolación de Richardson
44          for m = 2:n+1
45              R(n+1,m) = R(n+1,m-1) + (R(n+1,m-1) - R(n,m-1)) /
                    (4^(m-1) - 1);
46          end
47          % Imprimir la fila
48          print_row(R(n+1, 1:n+1));
49          % Guardamos h y el error absoluto en la diagonal
50          hs = [hs, h];
51          errores = [errores, abs(R(n+1,n+1) - I_exacto)];
52          % Criterio de convergencia
53          if abs(R(n+1,n) - R(n+1,n+1)) < eps
54              I_romberg = R(n+1,n+1);
55              return;
56          end
57          n = n + 1;
58      end
59  end
60  function print_row(row)
61      fprintf('%11.8f ', row);
62      fprintf('\n');
63  end
```