

Análisis Numérico: Parcial 3

3 de marzo de 2025

Universidad Nacional de Colombia

Jorge Mauricio Ruiz Vera

Andrés David Cadena Simons
Sandra Natalia Florez Garcia

acadenas@unal.edu.co
sflorezga@unal.edu.co

Problema 1:

Dada la función

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

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 $x_n \rightarrow 1$ aunque esta iteración no está 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:

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

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 :

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

Por lo tanto, $x_n \rightarrow 1$.

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

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

Evaluyendo 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.

Para obtener un algoritmo que converja cuadráticamente a la raíz $x = 1$, podemos usar el **método de Newton**:

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

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

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

Aplicamos la fórmula de Newton:

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

Este método tiene **convergencia cuadrática**, lo que significa que el error disminuye aproximadamente como el cuadrado del error anterior en cada iteración.

Problema 2:

Sea

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

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

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

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

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

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:

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

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

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

(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.

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.

```
1  % Programa para encontrar todas las raíces reales de un polinomio
   usando el método descrito
2
3  % Definir el polinomio P(x) mediante sus coeficientes en orden
   descendente
4  % Ejemplo: P(x) = 2x^3 -4x +1 (raíces: 1, 2, 3)
5  original_coeffs = [2, 0, -4, 1]; % Guardar coeficientes originales
6
7  % Parámetros
8  tol = 1e-6;          % Tolerancia para la convergencia de
   Newton-Raphson
9  max_iter = 100;      % Máximo número de iteraciones por raíz
10
11 % Inicializar vectores
12 d = length(original_coeffs) - 1;    % Grado inicial del polinomio
13 current_coeffs = original_coeffs;    % Coeficientes del polinomio
   actual
14 roots_approx = zeros(1, d);          % Almacenar las raíces
   aproximadas
15
16 % Bucle para encontrar todas las raíces
17 for k = d:-1:1
18     % Verificar que current_coeffs sea válido
19     if isempty(current_coeffs) || length(current_coeffs) < 2
20         error('Error: Coeficientes inválidos en iteración %d', k);
21     end
22
23     % Paso (a): Calcular la cota M
24     ad = abs(current_coeffs(1));      % Coeficiente principal
       |a_d|
25     sum_abs = sum(abs(current_coeffs(2:end))) / ad; % Sum
       |a_i|/|a_d|
26     M = max(1, sum_abs);
27     x0 = M + 1; % Elegir x0 mayor que M
28
29     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
42             %d', x, k);
43     end
44
45     % Actualizar x
46     x_new = x - Px / P_prime_x;
47
48     % Comprobar convergencia
49     if abs(x_new - x) < tol
50         break;
51     end
52     x = x_new;
53
54     if iter == max_iter
55         warning('Máximo de iteraciones alcanzado para la raíz
56             %d.', k);
57     end
58
59     % Guardar la raíz aproximada
60     r_tilde = x_new;
61     roots_approx(k) = r_tilde;
62
63     % Paso (c): Dividir P(x) por (x - r_tilde)
64     [Q, R] = deconv(current_coeffs, [1, -r_tilde]);
65     current_coeffs = Q; % Redefinir P(x) como el cociente
66
67     fprintf('Raíz %d aproximada: %f\n', k, r_tilde);
68 end
69
70 % Mostrar resultados
71 disp('Raíces aproximadas del polinomio:');
72 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.');
```

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

Iteración 3: $M = 2,500000$, $x_0 = 3,500000$
Raíz 3 aproximada: 1,267035

Iteración 2: $M = 1,661657$, $x_0 = 2,661657$
Raíz 2 aproximada: 0,258652

Iteración 1: $M = 1,525687$, $x_0 = 2,525687$
Raíz 1 aproximada: $-1,525687$

Las raíces aproximadas del polinomio son:

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

Para comparación, las raíces exactas son:

$-1,5257$
 $0,2587$
 $1,2670$

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 $x_0 = a, x_1 = a + h, \dots, x_n = b$, donde $h = \frac{b-a}{n}$. Considerar la poligonal $l(x)$ que interpola a f en los puntos $x_i, i = 0, 1, \dots, n$. 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

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

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.

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 abscisas 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 delinear 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:

- 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.

```

1      % Configurar la figura para la entrada de puntos
2      figure('position', get(0, 'screensize'))
3      axes('position', [0 0 1 1])
4      disp('Haga clic en varios puntos y presione Enter cuando termine');
5
6      [x, y] = ginput(); % Obtener puntos del usuario
7      x = [x; x(1)]; % Cerrar el polígono
8      y = [y; y(1)];
9
10     plot(x, y, 'bo-', 'LineWidth', 1.5, 'MarkerSize', 8);
11     hold on;
12     disp([x, y]);
13
14     % Verificación de puntos mínimos
15     if length(x) < 4

```

```
16     error('Se necesitan al menos 4 puntos para la interpolación  
       spline cúbica');  
17 end  
18  
19 % Parametrización  
20 t = 1:length(x);  
21 t_fine = linspace(1, length(x), 100); % Puntos finos para suavidad  
22  
23 % Interpolación con spline de MATLAB  
24 x_spline = spline(t, x, t_fine);  
25 y_spline = spline(t, y, t_fine);  
26 plot(x_spline, y_spline, 'g--', 'LineWidth', 1.5, 'DisplayName',  
       'Spline de MATLAB');  
27  
28 % Número de incógnitas en el sistema tridiagonal  
29 n = length(x) - 2;  
30  
31 % Calcular segundas diferencias (dx, dy)  
32 dx = 6 * (x(3:end) - 2*x(2:end-1) + x(1:end-2));  
33 dy = 6 * (y(3:end) - 2*y(2:end-1) + y(1:end-2));  
34  
35 % Vectores de la matriz tridiagonal  
36 a = ones(n-1, 1); % Subdiagonal  
37 b = 4 * ones(n, 1); % Diagonal principal  
38 c = ones(n-1, 1); % Superdiagonal  
39  
40 % Resolver el sistema tridiagonal para sigma_x y sigma_y  
41 sigma_x_interior = thomas_solver(a, b, c, dx);  
42 sigma_y_interior = thomas_solver(a, b, c, dy);  
43  
44 % Agregar condiciones de frontera (sigma = 0 en extremos)  
45 sigma_x = [0; sigma_x_interior; 0];  
46 sigma_y = [0; sigma_y_interior; 0];  
47  
48 % Interpolación personalizada usando splines cúbicos  
49 x_int = [];  
50 y_int = [];  
51 for i = 1:length(x)-1  
52     t_i = linspace(t(i), t(i+1), 10);  
53     for k = 1:length(t_i)  
54         x_val = spline_polinomios(t_i(k), t(i), t(i+1), x(i), x(i+1),  
                                   sigma_x(i), sigma_x(i+1));  
55         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.

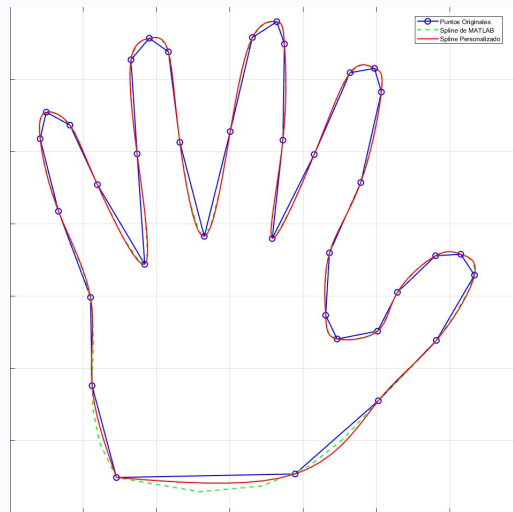


Figura 2: Spline de la mano de Andrés.

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:

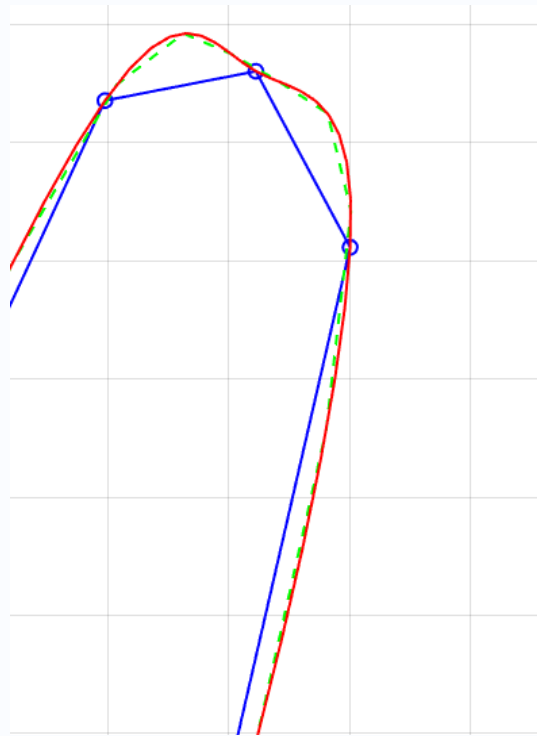
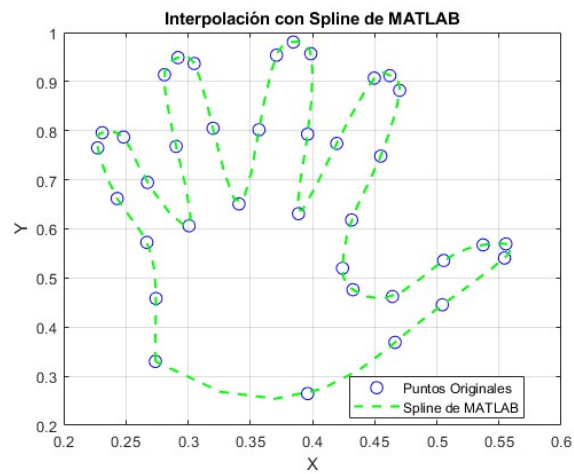
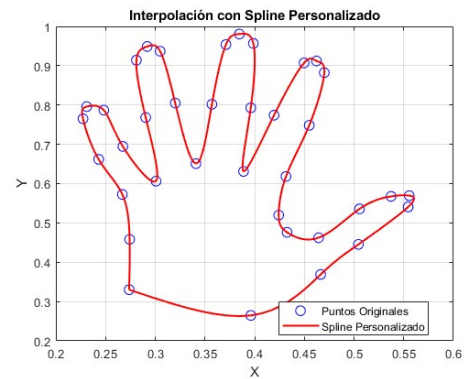
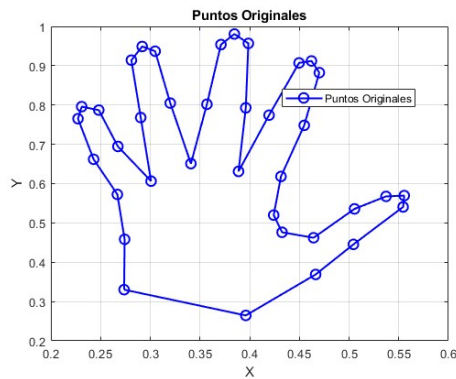


Figura 3: Vista ampliada de la gráfica en un dedo.

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 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 $h \rightarrow 0$.

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:

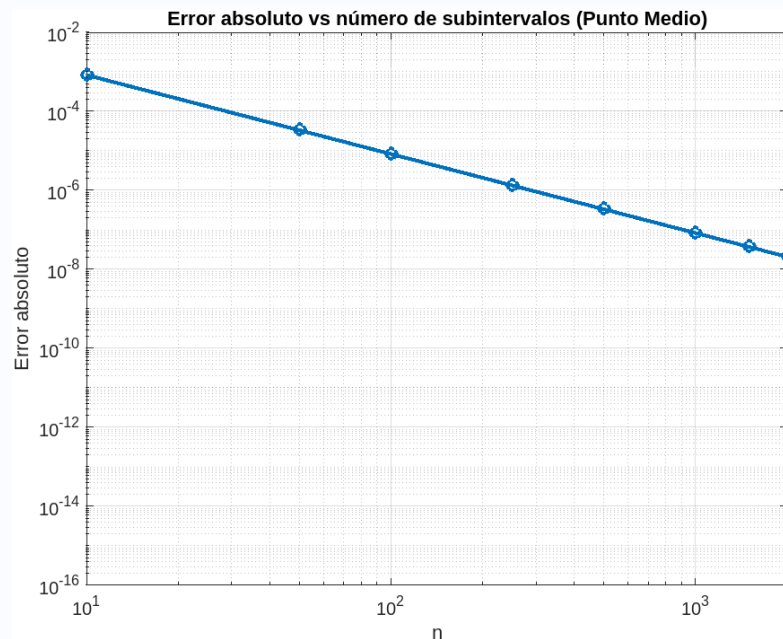


Figura 4: Error absoluto en el método del punto medio.

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 tomó 0,001995 segundos.

```
1  format rational
2
3  % Creamos la función f
4  f = @(x) 4 ./ (1 + x.^2);
5
6  % Parámetros de la integral y el método
7  a = 0;
8  b = 1;
9  valores_n = [10, 50, 100, 250, 500, 1000, 1500, 2000];
10 I_exacto = pi;
11
12 % Se crea un vector para guardar los errores
13 errores = zeros(size(valores_n));
14
15 % Método del punto medio
16 tic;
17 for k = 1:length(valores_n)
18     n = valores_n(k);
19     % Aquí se encuentra el tamaño del intervalo
20     h = (b - a) / n;
21     % Se calculan los puntos medios y los guarda en el vector
22     x_medios = a + ((2*(0:1:n-1)+1)*h)/2;
23     % Regla del punto medio compuesta
24     I_punto_medio = h * sum(f(x_medios));
25     % Calculo del error
26     errores(k) = abs(I_punto_medio - I_exacto);
27 end
28 tiempo = toc
29
30 % Graficar log-log del error vs n
31 figure;
32 loglog(valores_n, errores, '-o', 'LineWidth', 2);
33 xlabel('n');
34 ylabel('Error absoluto');
35 ylim([1e-16, 1e-2]);
36 title('Error absoluto vs número de subintervalos (Punto
37       Medio)');
37 grid on;
```

■ Trapecio.

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

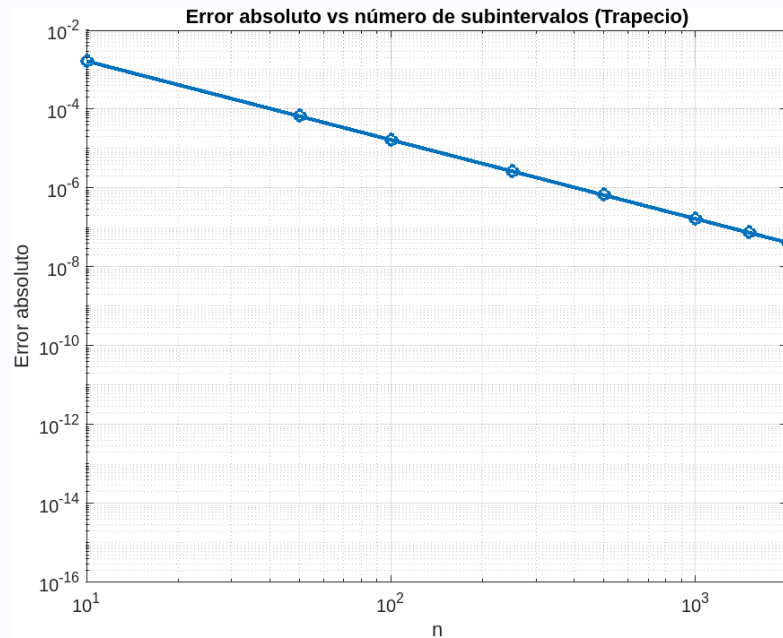


Figura 5: Error absoluto en el método del trapecio.

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 tomó 0,002719 segundos.

```
1  format rational
2
3  % Creamos la función f
4  f = @(x) 4 ./ (1 + x.^2);
5
6  % Parámetros de la integral y el método
7  a = 0;
8  b = 1;
9  valores_n = [10, 50, 100, 250, 500, 1000, 1500, 2000];
10 I_exacto = pi;
11
12 % Se crea un vector para guardar los errores
13 errores_trapezio = zeros(size(valores_n));
14
15 % Método del trapecio
16 for k = 1:length(valores_n)
17     n = valores_n(k);
18     h = (b - a) / n;
19     % Extremos de la partición
20     x_extremos = a:h:b;
21     m = length(x_extremos);
22
23     % Aplicar la regla del trapecio compuesta
24     I_trapezio = (h/2) * (f(a) + 2*sum(f(x_extremos(2:m-1)))
25         + f(b));
26
27     % Calculo del error
28     errores_trapezio(k) = abs(I_trapezio - I_exacto);
29 end
30
31 % Graficar error del método del trapecio
32 figure;
33 loglog(valores_n, errores_trapezio, '-o', 'LineWidth', 2);
34 xlabel('n');
35 ylabel('Error absoluto');
36 ylim([1e-16, 1e-2]);
37 title('Error absoluto vs número de subintervalos (Trapezio)');
38 grid on;
```


- Simpson.

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

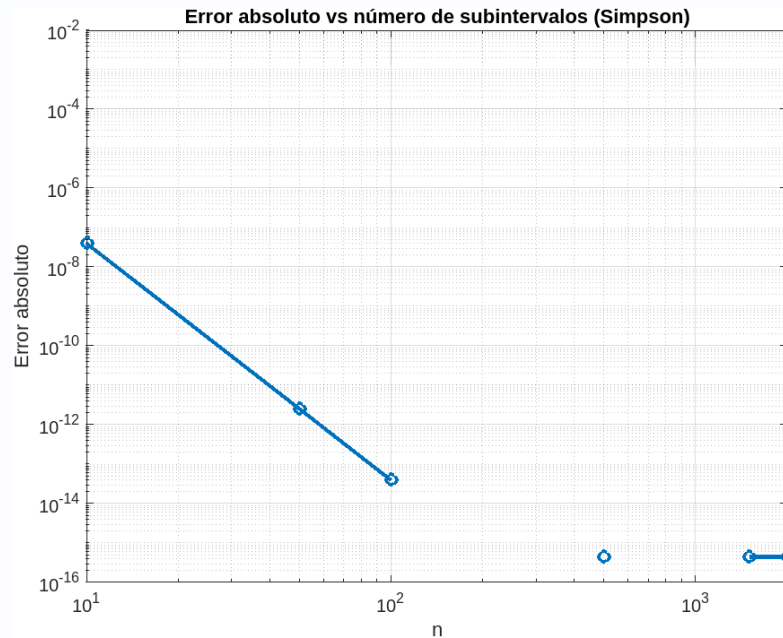


Figura 6: Error absoluto en el método de Simpson.

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 tomó 0,003878 segundos.

```
1  format rational
2  % Creamos la función f
3  f = @(x) 4 ./ (1 + x.^2);
4  % Parámetros de la integral y el método
5  a = 0;
6  b = 1;
7  valores_n = [10, 50, 100, 250, 500, 1000, 1500, 2000];
8  I_exacto = pi;
9  % Se crea un vector para guardar los errores
10 errores_simpson = zeros(size(valores_n));
11 % Método de Simpson
12 tic;
13 for k = 1:length(valores_n)
14     n = valores_n(k);
15     % Aquí nos aseguramos que n sea par (Simpson necesita n
16     % par)
17     if mod(n, 2) == 1
18         n = n + 1;
19     end
20     h = (b - a) / n;
21     % Extremos de la partición
22     x_extremos = a:h:b;
23     m = length(x_extremos);
24     % Aplicar la regla de Simpson compuesta
25     I_simpson = (h/3) * (f(a) + 4*sum(f(x_extremos(2:2:m-1)))
26     + 2*sum(f(x_extremos(3:2:m-2))) + f(b));
27     % Calculo del error
28     errores_simpson(k) = abs(I_simpson - I_exacto);
29 end
30 tiempo = toc
31 % Graficar error del método de Simpson
32 figure;
33 loglog(valores_n, errores_simpson, '-o', 'LineWidth', 2);
34 xlabel('n');
35 ylabel('Error absoluto');
36 ylim([1e-16, 1e-2]);
37 title('Error absoluto vs número de subintervalos (Simpson)');
38 grid on;
```

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

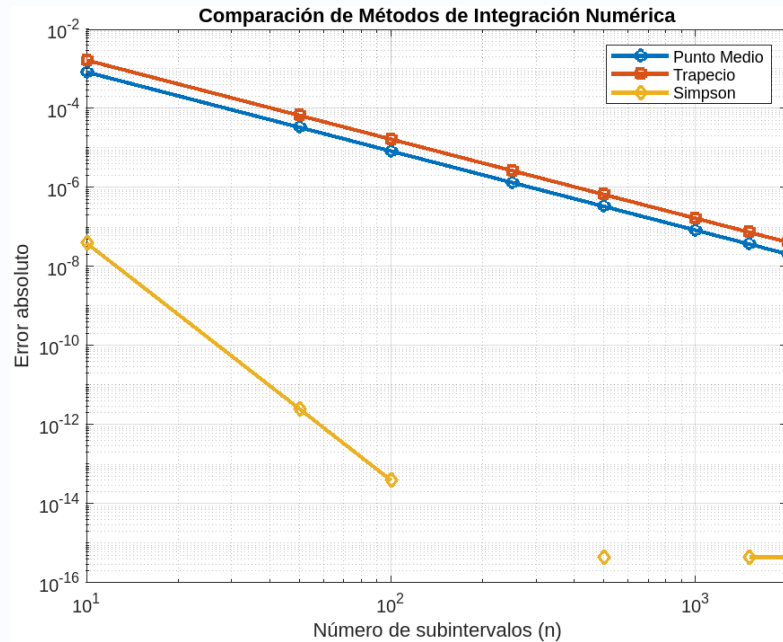


Figura 7: Comparación de los métodos.

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 $h \rightarrow 0$ 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 "cero", 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 . Grafique 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:

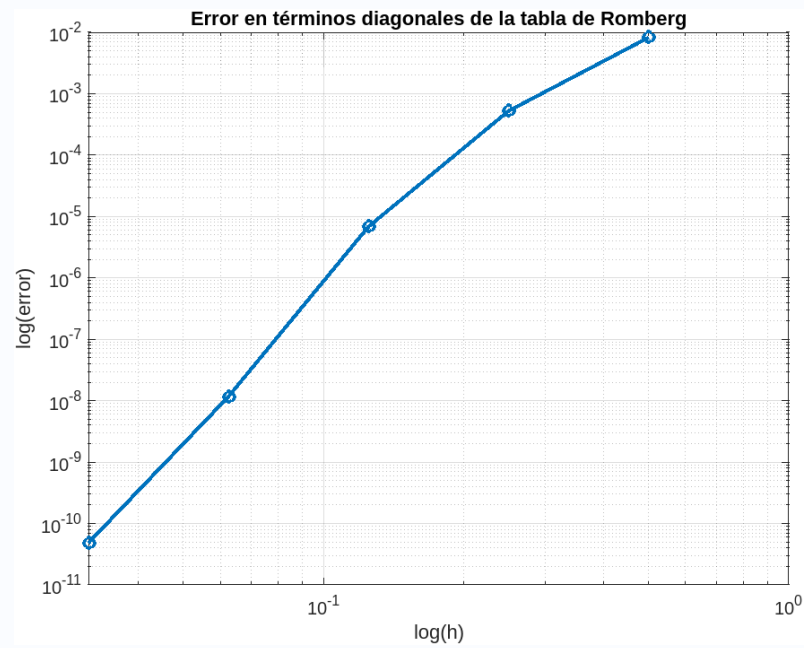


Figura 8: Error en los términos diagonales en la tabla de extrapolación versus $\log h$.

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 casi una recta, de lo cual podemos deducir al estar en escala logarítmica 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.

```
1  format long
2  % Definimos la función
3  f = @(x) 4 ./ (1 + x.^2);
4  % Límites de integración
5  a = 0;
6  b = 1;
7  % Valor exacto de la integral
8  I_exacto = pi;
9  % Tolerancia
10 eps = 1e-8;
11 % Calculamos la integral con Romberg y obtenemos errores
12 tic;
13 [I_romberg, hs, errores] = romberg(f, a, b, I_exacto, eps);
14 toc
15 % Graficamos log(error) vs log(h)
16 figure;
17 loglog(hs, errores, '-o', 'LineWidth', 2);
18 xlabel('log(h)');
19 ylabel('log(error)');
20 title('Error en términos diagonales de la tabla de Romberg');
21 grid on;
22 % --- FUNCIONES ---
23 function [I_romberg, hs, errores] = romberg(f, a, b, I_exacto,
24     eps)
25     if nargin < 5
26         % Precisión por defecto
27         eps = 1e-8;
28     end
29     % Matriz de Romberg
30     R = zeros(1, 1);
31     % Primer trapecio
32     R(1,1) = 0.5 * (b - a) * (f(a) + f(b));
33     print_row(R(1,1));
34     % Inicialización de vectores para la gráfica
35     hs = [];
36     errores = [];
37     % Número de iteraciones
38     n = 1;
39     while true
40         % Refinamos h
41         h = (b - a) / 2^n;
42         % Nueva regla del trapecio
43         R(n+1,1) = 0.5 * R(n,1) + h * sum(f(a + (2*(1:2^(n-1)) -
44             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
```