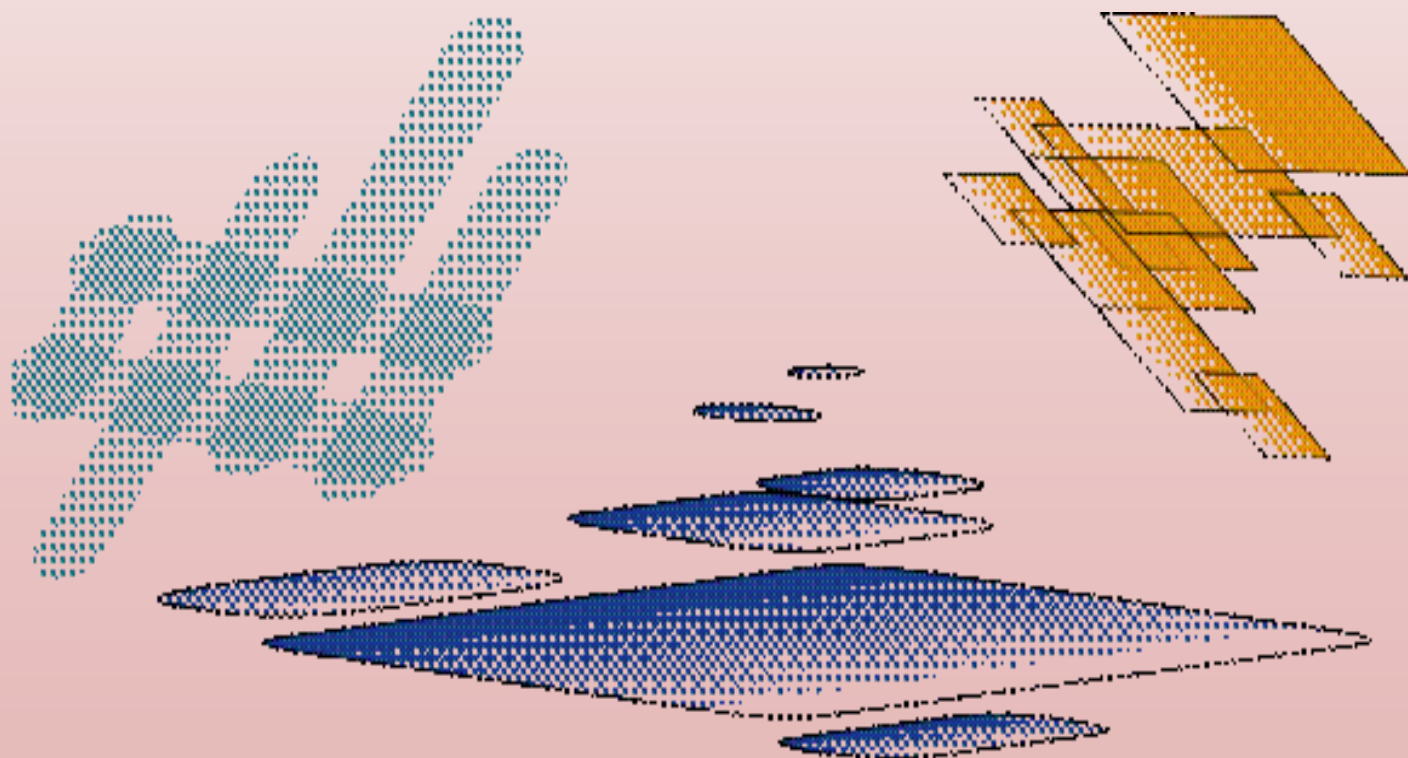
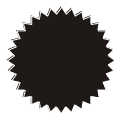


LA SOLUCIÓN NUMÉRICA ELEMENTAL CON OCTAVE



MATIAUDA MARIO EUGENIO
YACHECEN CARLOS GUSTAVO



EDITORIAL UNIVERSITARIA DE MISIONES

San Luis 1870

Posadas - Misiones – Tel-Fax: (03752) 428601

Correos electrónicos:

edunam-admini@arnet.com.ar

edunam-direccion@arnet.com.ar

edunam-produccion@arnet.com.ar

edunam-ventas@arnet.com.ar

Colección: Cuadernos de Cátedra

Coordinación de la edición: Claudio Oscar Zalazar

ISBN 978-950-579-190-3

Impreso en Argentina

©Editorial Universitaria

Matiauda, Mario Eugenio

La solución numérica elemental con octaves. - 1a ed. - Posadas : EDUNAM - Editorial Universitaria de la Universidad Nacional de Misiones, 2011.

Internet.

ISBN 978-950-579-190-3

1. Matemática . 2. Solución Numérica. I. Título.

CDD 510

Fecha de catalogación: 10/03/2011

INDICE

UNIDAD 1 : RESOLVER $f(x) = 0$

1.1 Método de la bisección.....	6
1.2 Método del punto fijo	6
1.3 Método de Newton	7
1.4 Método de la secante	7
1.5 Método de la posición falsa.....	8
1.6 Método de Steffensen	8
1.7 Polinomios. Raíces. Métodos.....	8
1.7.1 Método de Horner para ubicar raíces de un polinomio P de grado n.....	9
1.7.2 Método de Muller	9

UNIDAD 2 : INTERPOLACIÓN

2.1 Interpolación. Empleo de polinomios.....	11
2.2 Polinomio de Lagrange	11
2.3 Polinomio de Hermite	12
2.4 Método de Neville	13
2.5 Diferencias divididas	14
2.6 Polinomios osculadores. Polinomios de Hermite	16
2.7 Adaptador cúbico	16

UNIDAD 3 : DIFERENCIACIÓN E INTEGRACIÓN NUMÉRICA

3.1 Diferenciación numérica. Formulas de aproximación.....	17
3.2 Diferenciación numérica. Técnica de Richardson	18
3.3 Integración numérica.....	18
3.3.1 Fórmula de Newton.....	18
3.3.2 Regla compuesta del trapezio	19
3.3.3 Regla compuesta de Simpson	19
3.3.4 Integración de Romberg	20
3.3.5 Regla trapezoidal recursiva	21
3.4 Fórmulas Gaussianas	21
3.4.1 Fórmula de Gauss-Legendre	21
3.4.2 Fórmula de Gauss-Laguerré.....	22
3.4.3 Fórmula de Gauss-Chebyshev	22

3.5 Cuadratura adaptativa utilizando la regla de Simpson's.....	22
3.6 Regla de Simpson's	23
3.7 Cuadratura de Gauss-Legendre.....	23

UNIDAD 4 : ECUACIONES DIFERENCIALES ORDINARIAS

4.1 Métodos para resolver P.V.I.	24
4.1.1 Método de Euler	24
4.1.2 Serie de Taylor	24
4.1.3 Método de Runge-Kutta.....	25
4.1.4 Métodos multipasos.....	25
4.1.4.1 Método de Adams-Bashforth	26
4.1.4.2 Método de Adams-Moulton	27
4.1.5 Método de Milne	28
4.1.6 Predictor-Corrector	28
4.1.7 Uso de la extrapolación	28

UNIDAD 5 : SISTEMAS LINEALES

5.1 Representación de un sistema lineal. Sustitución hacia atrás.....	29
5.2 Técnica de Gauss Jordan	30
5.3 Pivoteo	30
5.4 Factorización matricial.....	31
5.4.1 Factorización LU.....	31
5.5 Técnicas de repetición para sistemas lineales	34
5.5.1 Técnica de Jacobi.....	35
5.5.2 Técnica de Gauss-Seidel.....	35
5.6 Técnica de relajación	36
5.7 Gradiente conjugado	38

UNIDAD 6 : APROXIMACIÓN

6.1 Polinomios de Chebyshev	39
6.2 Aproximación mínimo-máximo (o de Chebyshev).....	39
6.3 Aproximación por valores propios	39
6.4 Técnicas de aproximación.....	39
6.4.1 Método de potencias.....	39
6.4.2 Método de la potencia inversa	40

6.4.3 Método simétrico de potencias	41
6.4.4 Método clásico de Jacobi.....	41
6.4.5 Técnicas de deflación	42
6.4.6 Técnicas de Householder	42
6.4.7 Algoritmo QR	42
6.5 Ortogonalización de Gram-Schmidt	42

UNIDAD 7 : SISTEMAS DE ECUACIONES NO LINEALES

7.1 Método de Newton	43
7.2 Técnicas cuasi-Newton	43
7.3 Técnicas de mayor pendiente	44
7.4 Problema de valor de frontera para EDO's y en derivadas parciales.....	44
7.5 Técnica de disparo para el problema lineal	44
7.6 Técnica de disparo para el problema no lineal.....	45
7.7 Diferencias finitas para problemas lineales	45

UNIDAD 8 : ECUACIONES EN DERIVADAS PARCIALES

8.1 Ecuación general	47
8.2 Elíptica o de Poisson.....	47
8.3 Ecuación parabólica.....	48
8.4 Ecuación hiperbólica	49

Prólogo

En diversas ocasiones, situaciones del mundo físico se desean representar matemáticamente pero, esos problemas no encuentran una resolución analítica exacta o deben ser encarados desde la óptica Numérica.

Varias de esas situaciones se incluyeron en la Solucion Numérica Elemental del autor, conteniendo la alternativa de búsqueda de resultados a través de la computadora.

En esta presentación se aborda la misma línea, simplemente iniciando al uso de Octave, sofá libre y de permanente actualización y con amplia Potencialidad de emulación de software bajo licencia (caso MatLab).

Por ello no se abunda en desarrollo de los métodos sino directamente su ejecución bajo Octave.

RESOLVER $f(x)=0$

SOLUCIONES DE ECUACIONES UNIVARIABLES

Para una función $f(x)=0$, el problema más simple de la aproximación numérica será hallar la raíz x que anula f .

1.1 METODO DE LA BISECCIÓN

Se parte de f definida en I real con extremos a y b pertenecientes a I , bajo el supuesto de que $f(a) \cdot f(b) < 0$.

$$p_i = \frac{a_i + b_i}{2} \quad \text{Con } p_i = \text{punto medio}$$

Se toma el punto medio $\frac{a+b}{2}$ si $f(\frac{a+b}{2}) = 0$ ya hemos encontrado la raíz $x = \frac{a+b}{2}$. En caso contrario, si $f(\frac{a+b}{2}) \cdot f(b) < 0$ entonces hacemos $a = \frac{a+b}{2}$ y volvemos a subdividir el nuevo intervalo $[a, b]$. Si, por el contrario, $f(a) \cdot f(\frac{a+b}{2}) < 0$ entonces hacemos $b = \frac{a+b}{2}$ y volvemos a empezar. Las sucesivas subdivisiones del intervalo $[a, b]$. Van aproximando la raíz.

$$\text{Criterio de detención} \quad |p_n - p_{n-1}| \leq \frac{b-a}{2^n} < \varepsilon \quad n \geq 1$$

[Algoritmo bisección](#)

1.2 METODO DEL PUNTO FIJO

El problema de hallar raíces de $h(q) = 0$ con $q = \text{Punto fijo}$

$$h(x) = x - f(x) \quad \text{o} \quad h(x) = x + \alpha f(x), \quad \alpha \in \mathbf{R}$$

Si la función $h(x)$ tiene un punto fijo en P , entonces la función definida por $f(x) = x - h(x)$ tiene un cero en P .

La forma de punto fijo es más fácil de analizar que la búsqueda de raíces.

Si $h \in C[a, b]$ y $h(x) \in [a, b]$ para toda $x \in [a, b]$, h tiene un punto fijo en $[a, b]$.

Si también $g'(x)$ existe en $]a, b[$ con una constante $m < 1$, de modo que

$$|h'(x)| \leq m < 1 \quad \forall x \in]a, b[$$

Que garantiza que el punto fijo es único en $[a, b]$.

Algoritmo fixed_point

1.3 METODO DE NEWTON

Partiendo de un p_0 de arranque (requiere la elección adecuada de p_0), se genera la sucesión $\{p_n\}$ definida a través de:

$$p_n = p_{n-1} - \frac{g(p_{n-1})}{g'(p_{n-1})} \quad \text{para } n \geq 1$$

Es una técnica de iteración funcional, donde las aproximaciones se obtienen usando tangentes sucesivas.

$$\begin{aligned} \text{Criterio de detención} \quad & |p_n - p_{n-1}| < \varepsilon \quad \varepsilon > 0 \quad \text{o} \\ & \frac{|p_N - p_{N-1}|}{|p_N|} < \varepsilon \quad p_N > 0 \end{aligned}$$

Algoritmo newton

1.4 METODO DE LA SECANTE

Se parte de las aproximaciones de arranque x_0 y x_1 , donde las posteriores aproximaciones se obtienen usando secantes sucesivas; evitando así el problema de conocer la derivada de la función como en la técnica de Newton, así sustituye la derivada por la relación de las diferencias

$$x_n = x_{n-1} - \frac{g(x_{n-1})(x_{n-1} - x_{n-2})}{g(x_{n-1}) - g(x_{n-2})}$$

Algoritmo secant

1.5 METODO DE LA POSICIÓN FALSA

Genera aproximaciones del mismo modo que el método de la secante, pero incorpora el acorralamiento de la raíz, asegurando que la misma quede entre dos iteraciones sucesivas.

$y - f(a) = \frac{f(b) - f(a)}{b - a} (x - a)$ Como x_1 es el valor de x para $y=0$, se tiene

$$x_1 = a - \frac{f(a)}{f(b) - f(a)} (x - a) = \frac{af(b) - bf(a)}{f(b) - f(a)} \quad \text{si} \quad \left| \frac{x_1 - a}{x_1} \right| < \varepsilon \quad \text{ó} \quad \left| \frac{x_1 - b}{x_1} \right| < \varepsilon$$

para una ε pre-establecida, entonces x_1 es la raíz buscada. De lo contrario, se calcula $f(x_1)$ y elegir entre a y b aquella cuya f tenga signo opuesto de $f(x_1)$. Con x_1 (en) ese punto calculado x_2 mediante la fórmula de la secante. El proceso iterativo debe continuar hasta obtener la raíz con la precisión pre-establecida.

[Algoritmo regula](#)

1.6 METODO DE STEFFENSEN

Genera la secuencia:

$$p_0^{(0)}, \quad p_1^{(0)} = g(p_0^{(0)}), \quad p_2^{(0)} = g(p_1^{(0)}), \quad p_0^{(1)} = \{\Delta^2\}(p_0^{(0)}), \quad p_1^{(1)} = g(p_0^{(1)}), \dots$$

donde Δ^2 indica que se usa la ecuación $\hat{p} = p_n - \frac{(\Delta p_n)^2}{\Delta^2 p_n}$, para $n \geq 0$.

la cual genera cada tercer término; los demás usan la iteración del punto fijo en el término anterior.

Para encontrar rápidamente una solución de la ecuación de punto fijo $x = g(x)$ dada una aproximación inicial P_0 ; donde se supone que tanto $g(x)$ y $g'(x)$ son continuas, $|g'(x)| < 1$, y que la iteración de punto fijo ordinario converge lentamente (lineal) para p

[Algoritmo steff](#)

1.7 POLINOMIOS. RAÍCES. MÉTODOS

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad \text{Polinomio de grado } n$$

Con a_i = coeficientes de P (son constantes)

1.7.1 Método de Horner

Sea $b_n = a_n$

$$b_{n-k} = xb_{n-k+1} + a_{n-k} \quad , \quad k = 1, 2, \dots, n. \quad (\text{Fórmula de recurrencia})$$

Basándonos en la fórmula de recurrencia dada, sustituyendo x por z , si z es una raíz de $P(x)$, podemos escribir que:

$$P(x) = (x - z)Q(x)$$

Donde $Q(x) = b_n x^{n-1} + b_{n-1} x^{n-2} + \dots + b_1 = \frac{P(x)}{x - z}$ en efecto

$$\begin{aligned} & (b_n x^{n-1} + b_{n-1} x^{n-2} + \dots + b_1)(x - z) \\ &= b_n x^n + (b_{n-1} - zb_n)x^{n-1} + \dots \\ &+ (b_1 - zb_2)x + (b_0 - zb_1) \\ &= a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = P(x) \end{aligned}$$

Concluimos que cualquier raíz de $Q(x)$ es también una raíz de $P(x)$, esto nos permite trabajar con un polinomio de grado $n-1$, o sea, con $Q(x)$, para calcular las raíces subsecuentes de $P(x)$, este proceso recibe el nombre de deflación, con este se evita que un mismo cero sea calculado varias veces.

[Algoritmo horner](#)

[Algoritmo newtonhorner](#)

1.7.2 Método de Muller

Es una extensión del método de la secante, utiliza tres aproximaciones iniciales: x_0 y x_1, x_2 .

Considerando el polinomio cuadrático:

$$f_2(x) = a(x - x_2)^2 + b(x - x_2) + c$$

Con

$$c = f(x_2)$$

$$b = \frac{(x_0 - x_2)[f(x_1) - f(x_2)] - (x_1 - x_2)^2[f(x_0) - f(x_2)]}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)}$$

$$a = \frac{(x_1 - x_2)[f(x_0) - f(x_2)] - (x_0 - x_2)[f(x_1) - f(x_2)]}{(x_0 - x_2)(x_1 - x_2)(x_0 - x_1)}$$

Hallando la raíz, se implementa la solución convencional, pero debido al error de redondeo potencial, se usará una formulación alternativa:

$$x_3 - x_2 = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \quad \text{despejando} \quad x_3 = x_2 + \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

La gran ventaja de este método es que se pueden localizar tanto las raíces reales como las imaginarias.

La elección de las fórmulas anteriores equivale a aproximar $f(x)$ por la parábola que pasa por los puntos $(x_{n-3}, f(x_{n-3}))$, $(x_{n-2}, f(x_{n-2}))$, $(x_{n-1}, f(x_{n-1}))$, y calcular posteriormente las derivadas de dicha parábola.

[Algoritmo muller](#)

[Algoritmo muller2](#)

INTERPOLACION

2.1 INTERPOLACION. EMPLEO DE POLINOMIOS

El problema general de la interpolación de funciones consiste en, a partir del conocimiento del valor de una función (y eventualmente de sus derivadas) en un conjunto finito de puntos, aproximar el valor de la función fuera de ese conjunto finito de puntos.

Según el teorema de aproximación

$$|f(x) - P(x)| < \varepsilon \quad \text{Con } \varepsilon > 0, \quad \forall x \in [a, b]$$

El uso de los polinomios tiene consigo ventajas como su derivabilidad e integrabilidad, dando también polinomio.

El planteo del problema será hallar un polinomio interpolante que brinde una aproximación precisa en todo el intervalo, de allí la no adecuación de los polinomios de Taylor que concentran la información alrededor de un punto, dígase x_0 .

2.2 POLINOMIO DE LAGRANGE

$$f(x_k) = P(x_k) \quad k = 0, 1, 2, \dots, n$$

$$P(x) = f(x_0)L_{n,0}(x) + \dots + f(x_n)L_{n,n}(x) = \sum_{k=0}^n f(x_k)L_{n,k}(x)$$

Con

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n)}{(x_k-x_0)(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n)}$$

$$L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{(x-x_i)}{(x_k-x_i)} \quad k = 0, 1, 2, \dots, n$$

Uno de los inconvenientes que presenta la interpolación de Lagrange, es que el grado del polinomio necesario, no se conoce, hasta la determinación de los cálculos necesarios.

[Algoritmo lagrange](#)

2.3 POLINOMIO DE HERMITE

En ocasiones, resulta de interés interpolar no sólo el valor de la función en ciertos puntos $\{x_i\}_{i=0,\dots,N}$, sino también el valor de sus derivadas. Un ejemplo clásico de ello es el desarrollo de Taylor de una función en un punto a . En este caso, aproximamos $f(x)$ por un polinomio de grado N , $P_N(x)$ tal que $f(x)$ y $P_N(x)$ poseen las mismas derivadas en el punto a desde el orden 0 hasta el orden N .

$$P_N(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \dots + \frac{f^{(N)}(a)}{N!}(x-a)^{(N)}$$

El error de interpolación viene dado por la fórmula

$$f(x) - P_N(x) = \frac{f^{(N+1)}(\xi)}{(N+1)!}(x-a)^{(N+1)}$$

donde ξ es un valor intermedio entre x y a . En el caso general, donde buscamos un polinomio $P(x)$ tal que él y todas sus derivadas hasta un cierto orden M coincidan con una función $f(x)$ en los puntos $\{x_i\}_{i=0,\dots,N}$, se utilizan los denominados polinomios base de Hermite $H_{i,j}(x)$, que son polinomios de grado menor o igual que $(N+1)(M+1)-1$ dados por las siguientes condiciones:

$$\frac{\partial^l H_{i,j}}{\partial x^l}(x_k) = \begin{cases} 1 & \text{si } l = j \text{ y } k = i \\ 0 & \text{si } l \neq j \text{ o } k \neq i \end{cases}$$

A partir de los polinomios base de Hermite, el polinomio interpolador de Hermite se define como:

$$P(x) = \sum_{i=0}^N \sum_{j=0}^M \frac{\partial^j f}{\partial x^j}(x_i) H_{i,j}(x)$$

[Algoritmo hermite](#)

2.4 MÉTODO DE NEVILLE

Si $N_{i,j}$ con $0 \leq i \leq j$ representa el polinomio interpolante de grado j en los $(j+1)$ números $x_{i-j}, x_{i-j+1}, \dots, x_{i-1}, x_i$

$$N_{i,j} = P_{i-j, i-j+1, \dots, i-1, i}$$

Se dispondrá de un diagrama para los polinomios interpolantes:

$$x_0 \quad P_0 = N_{0,0}$$

$$x_1 \quad P_1 = N_{1,0} \quad P_{0,1} = N_{1,1}$$

$$x_2 \quad P_2 = N_{2,0} \quad P_{1,2} = N_{2,1} \quad P_{0,1,2} = N_{2,2}$$

$$x_3 \quad P_3 = N_{3,0} \quad P_{2,3} = N_{3,1} \quad P_{1,2,3} = N_{3,2} \quad P_{0,1,2,3} = N_{3,3}$$

$$x_4 \quad P_4 = N_{4,0} \quad P_{3,4} = N_{4,1} \quad P_{2,3,4} = N_{4,2} \quad P_{1,2,3,4} = N_{4,3} \quad P_{0,1,2,3,4} = N_{4,4}$$

→ Desplazarse hacia la derecha aumenta el grado del polinomio



i más grande

$$N_{i,j} = \frac{(x - x_{i-j})N_{i,j-1} - (x - x_i)N_{i-1,j-1}}{x_i - x_{i-j}}$$

[Algoritmo neville1](#)

[Algoritmo neville2](#)

2.5 DIFERENCIAS DIVIDIDAS

Para calcular las diferencias divididas de una función $f(x)$ en $x_0, x_1, x_2, \dots, x_n$ construimos una tabla de diferencias divididas

x_i	$f[x_i]$	$[x_i, x_j]$	$f[x_i, x_j, x_k]$...
x_0	$f[x_0] = f_0$			
		$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$		
x_1	$f[x_1] = f_1$		$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$	
		$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1}$...
x_2	$f[x_2] = f_2$		$f[x_1, x_2, x_3] = \frac{f[x_2, x_3] - f[x_1, x_2]}{x_3 - x_1}$	
		$f[x_2, x_3] = \frac{f[x_3] - f[x_2]}{x_3 - x_2}$...
x_3	$f[x_3] = f_3$		$f[x_2, x_3, x_4] = \frac{f[x_3, x_4] - f[x_2, x_3]}{x_4 - x_2}$	
		$f[x_3, x_4] = \frac{f[x_4] - f[x_3]}{x_4 - x_3}$	\vdots	...
x_4	$f[x_4] = f_4$	\vdots		
\vdots	\vdots			

La primera columna contiene los puntos x_k , $k = 0, 1, \dots, n$

La segunda contiene los valores de la función en x_k , $k = 0, 1, \dots, n$

En las columnas 3, 4, 5, ..., están las diferencias divididas de orden 1, 2, 3, Cada una de estas es, una fracción cuyo numerador es siempre una diferencia entre dos diferencias divididas consecutivas, y de orden inmediatamente inferior y, cuyo denominador es la diferencia entre los dos extremos de los puntos involucrados.

$$f(x_0, x_1) = \frac{f_1 - f_0}{x_1 - x_0}$$

Para orden n :

$$f(x_0, x_1, \dots, x_n) = \frac{f(x_1, \dots, x_n) - f(x_0, \dots, x_{n-1})}{x_n - x_0}$$

Fórmulas de diferencias divididas interpolantes de Newton

$$P(x) = f[x_0] + \sum_{k=1}^n f[x_0, \dots, x_k](x - x_0) \dots (x - x_{k-1}) \text{ para cada } k = 0, 1, \dots, n$$

Fórmula de Newton de diferencias divididas hacia adelante

Una simplificación de la fórmula de interpolación de diferencias divididas se obtiene considerando igual espaciado entre los símbolo, x_0, x_1, \dots, x_n

Denotando $h = x_{i+1} - x_i$ para $i = 0, 1, \dots, n-1$ con $x = x_0 + sh$.

Será $x - x_i = (s-i)h$, convirtiendo a

$$P(x) = \sum_{k=0}^n s(s-1)\dots(s-k+1)h^k f[x_0, x_1, \dots, x_k]$$

Fórmula de Newton de diferencias divididas hacia atrás

Si los nodos tienen espacios iguales con $x = x_n + sh$ y $x = x_i + (s+n-i)h$ entonces

$$P_n(x) = P_n(x_n + sh) = f[x_n] + sh f[x_n, x_{n-1}] + \dots$$

$$+ s(s+1)h^2 f[x_n, x_{n-1}, x_{n-2}] + s(s+1)\dots(s+n-1)h^n f[x_n, \dots, x_0].$$

Con $s = \frac{x - x_n}{h}$

Diferencias centradas de Stirling

Si se busca aproximar un valor de x que se ubica cerca del centro de la tabla, no son adecuadas las expresiones de Newton.

Se emplean las fórmulas de diferencias centradas, según sea el caso, y dada la amplitud de expresiones se mencionan la de Stirling.

$$\begin{aligned} P(x) = P_{2m+1}(x) &= f[x_0] + \frac{sh}{2} (f[x_1, x_0] + f[x_0, x_1]) + s^2 h^2 f[x_{-1}, x_1, x_0] \\ &+ \frac{s(s^2-1)h^3}{2} (f[x_{-1}, x_0, x_1, x_2] + f[x_{-2}, x_{-1}, x_0, x_1]) + \dots \\ &+ s^2 (s^2-1)(s^2-4)\dots(s^2-(m-1)^2) h^{2m} f[x_{-m}, \dots, x_m] \\ &+ \frac{s(s^2-1)\dots(s^2-m^2)h^{2m+1}}{2} (f[x_{-m}, \dots, x_{m+1}] + f[x_{-m-1}, \dots, x_m]) \end{aligned}$$

para $n=2m+1$ impar, y si $n=2m$ es par se elimina el último término de la misma expresión.

[Algoritmo divdiff1](#)

[Algoritmo newpoly](#)

2.6 POLINOMIOS OSCULADORES. POLINOMIOS DE HERMITE

Polinomio con grado $\leq 2n+1$ y oscilación de primer orden, expresado por

$$P_{2n+1}(x) = \sum_{j=0}^n f(x_j) P_{n,j}(x) + \sum_{j=0}^n f'(x_j) \bar{P}_{n,j}(x)$$

$$\text{Siendo } P_{n,j}(x) = \left[1 - 2(x - x_j) L'_{n,j}(x_j) \right] L_{n,j}^2(x_j)$$

$$\bar{P}_{n,j}(x) = (x - x_j) L_{n,j}^2(x_j)$$

Con $L_{n,j}$ el j -ésimo polinomio de coeficientes de Lagrange (grado n).

Dado lo laborioso de determinar los polinomios de Lagrange con sus derivadas, alternativamente para aproximar por Hermite se hace uso de la fórmula de diferencia para los polinomios de Lagrange.

2.7 ADAPTADOR CÚBICO

El empleo de polinomios cúbicos garantiza la derivabilidad en el intervalo y posee derivadas segundas continuas, sin suponer que coincidan derivadas del polinomio interpolantes con las de la función.

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_jc_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1})$$

Para $j = 0, 1, \dots, n-1$

h = al paso entre nodos para $j = 0, 1, \dots, n-1$ y $a_n = f(x_n)$

Siendo:

$$c_{j+1} = c_j + 3d_j h_j$$

$$a_{j+1} = a_j + b_j h_j + \frac{h_j^2}{3}(2c_j + c_{j+1}) \quad \text{con } b_j = \frac{1}{h_j}(a_{j+1} - a_j) - \frac{h_j}{3}(2c_j + c_{j+1})$$

$$b_{j+1} = b_j + h_j(c_j + c_{j+1})$$

el conjunto $\{c_j\}_{j=0}^n$ serán las incógnitas.

Al conocer los c_j , se podrán hallar los coeficientes b_j y d_j , para poder generar los

polinomios cúbicos $\{A_j\}_{j=0}^{n-1}$.

[Algoritmo linear spline](#)

[Algoritmo spline eval](#)

DIFERENCIACIÓN E INTEGRACIÓN NUMERICAS

3.1 DIFERENCIACIÓN NUMERICA. FÓRMULAS DE APROXIMACIÓN

Para un intervalo con $(n+1)$ puntos diferentes $\{x_0, x_1, \dots, x_n\}$ con $f \in C^{n+1}$ sobre dicho intervalo

$$f'(x) = \sum_{j=0}^n f(x_j) L'_j(x) + \frac{f^{(n+1)}(\zeta(x))}{(n+1)!} \prod_{\substack{j=0 \\ j \neq 0}}^n (x_k - x_j)$$

que combina linealmente $(n+1)$ valores de $f(x)$, para $j = 0, 1, \dots, n$, conocida como fórmula de $(n+1)$ puntos.

Fórmulas de tres puntos

$$f'(x) = \frac{1}{2h} [-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)] + \frac{h^2}{3} f^{(3)}(\zeta_0)$$

Con ζ_0 entre x_0 y $x_1 + 2h$, y

$$f'(x) = \frac{1}{2h} [f(x_0 + h) - f(x_0 - h)] + \frac{h^2}{6} f^{(3)}(\zeta_1)$$

Con ζ_1 entre $(x_0 - h)$ y $(x_0 + h)$

Fórmulas de cinco puntos

$$f'(x_0) = \frac{1}{12h} [f(x_0 - 2h) - 8f(x_0 - h) + 8f(x_0 + h) - f(x_0 + 2h)] + \frac{h^4}{30} f^{(5)}(\zeta)$$

Y

$$f'(x_0) = \frac{1}{12h} [-25f(x_0) + 48f(x_0 + h) - 36f(x_0 + 2h) + 16f(x_0 + 3h) + 3f(x_0 + 4h)] + \frac{h^4}{5} f^{(5)}(\zeta)$$

Con ζ_0 entre x_0 y $x_0 + 4h$

La diferenciación basada en $N+1$ puntos para aproximar $f'(x_0)$ numéricamente mediante la construcción del polinomio de Newton de n ésimo grado.

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + \dots + a_N(x - x_0) \dots (x - x_{N-1})$$

Y usando $f'(x_0) \approx P'(x_0)$ como la aproximación esperada. El método debe ser utilizado en x_0 . Los puntos se pueden cambiar $\{x_k, x_0, \dots, x_{k-1}, x_{k+1}, \dots, x_N\}$ para calcular $f'(x_k) \approx P'(x_k)$.

[Algoritmo diffnew](#)

La diferenciación usando límite para aproximar $f'(x_0)$, generando la secuencia

$$f'(x) \approx D_k = \frac{f(x+10^{-k}h) - f(x+10^{-k-1}h)}{2(10^{-k}h)} \quad k = 0, \dots, n$$

Ya que $|D_{n+1} - D_n| \geq |D_n - D_{n-1}|$ o $|D_n - D_{n-1}| < \text{tolerancia}$, que es un intento de encontrar la mejor aproximación $f'(x) \approx D_n$.

[Algoritmo diffliim](#)

3.2 DIFERENCIACIÓN NUMERICA. TÉCNICA DE RICHARDSON

La expresión general será:

$$R = R(h) + \sum_{j=1}^{m-1} M_j h^j + O(h^m) \quad \text{para } c/j = 2, 3, \dots, m$$

Llamando $R(h)$ la expresión que aproxima el valor no conocido R , con un error de truncado $O(h)$, para ciertas constantes M_1, M_2, M_3, \dots :

Existe una aproximación para cada j del tipo:

$$R(h) = R_{j-1} + \frac{R_{h-1}(h/2) - R_{j-1}(h)}{2^{j-1} - 1}$$

La técnica (extrapolante) es aplicable siempre y cuando el error de truncado para una fórmula sea del tipo:

$$\sum_{j=1}^{m-1} M_j h^{\alpha_j} + O(h^{\alpha_m})$$

Para constantes M_j y que se verifique $\alpha_1 < \alpha_2 < \dots < \alpha_m$

[Algoritmo diffext](#)

3.3 INTEGRACIÓN NUMERICA

Si $P(x)$ aproxima a $f(x)$, se tendrá: $\int_a^b P(x)dx \approx \int_a^b f(x)dx$

3.3.1 Fórmula de Newton

$$\int_{x_0}^{x_1} P(x)dx = \frac{h}{2} (f(x_0) + f(x_1))$$

$$\int_{x_0}^{x_2} P(x)dx = \frac{h}{3} (f(x_0) + 4f(x_1) + f(x_2))$$

$$\int_{x_0}^{x_3} P(x)dx = \left(\frac{3h}{8}\right)(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3))$$

3.3.2 Regla Compuesta del Trapecio

Usa segmentos de recta como aproximación a $f(x)$.

$$\int_{x_0}^{x_n} f(x)dx = \frac{h}{2}[f(x_0) + 2f(x_1) + \dots + 2f(x_{n-1}) + f(x_n)]$$

Incluye nodos del intervalo abierto (x_0, x_n)

O también es posible expresarlo cómo:

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + f(b)) + h \sum_{k=1}^{M-1} f(x_k)$$

por muestreo $f(x)$ en los $M+1$ puntos equiespaciados

$x_k = a + kh$, para $k = 0, 1, 2, \dots, M$. Siendo $x_0 = a$ y $x_M = b$

[Algoritmo traprl](#)

3.3.3 Regla compuesta de Simpson

$$\int_{x_0}^{x_n} f(x)dx = \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

Incluye los puntos extremos del intervalo cerrado $[x_0, x_n]$.

O también es posible expresarlo cómo:

$$\int_a^b f(x)dx \approx \frac{h}{3}(f(a) + f(b)) + \frac{2h}{3} \sum_{k=1}^{M-1} f(x_{2k}) + \frac{4h}{3} \sum_{k=1}^M f(x_{2k-1})$$

por muestreo $f(x)$ en los $2M+1$ puntos equiespaciados

$x_k = a + kh$, para $k = 0, 1, 2, \dots, 2M$. Siendo $x_0 = a$ y $x_{2M} = b$

[Algoritmo simplr](#)

3.3.4 Integración compuesta de Romberg

Emplea la regla compuesta de los trapecios para una primaria aproximación y posteriormente utiliza la técnica de Richardson para mejoramientos de las aproximaciones.

$$R_{k,1} = \frac{1}{2} \left[R_{k-1,1} + h_{k-1} \sum_{i=1}^{k-2} f(a + (2i-1)h_k) \right] \quad k = 2, 3, \dots, n$$

Para cada $k = 2, 3, 4, \dots, n$ y $j = 2, \dots, k$ se puede reducir la notación expresando

$$R_{k,j} = R_{k,j-1} + \frac{R_{k,j-1} - R_{k-1,j-1}}{4^{j-1} - 1}$$

Conformando un diagrama característico

$$\begin{array}{ccccccc} R_{1,1} & & & & & & \\ R_{2,1} & R_{2,2} & & & & & \\ R_{3,1} & R_{3,2} & R_{3,3} & & & & \\ \cdot & & & & & & \\ \cdot & & & O & & & \\ R_{n,1} & L & L & L & R_{n,n} & & \end{array}$$

O también es posible expresarlo cómo:

$$\int_a^b f(x) dx \approx R(J, J)$$

mediante la generación de una tabla de aproximaciones $R(J, K)$ para $J \geq K$ y con $R(J+1, J+1)$ como la respuesta final. Las aproximaciones $R(J, K)$ se almacenan en una matriz triangular inferior especial. Los elementos de $R(J, 0)$ de la columna 0 se calculan usando la regla trapezoidal secuencial basado en 2^J subintervalos de $[a, b]$, entonces $R(J, K)$ se calcula utilizando la regla de Romberg.

Los elementos de la fila J son:

$$R(J, K) = R(J, K-1) + \frac{R(J, K-1) - R(J-1, K-1)}{4^K - 1}$$

para $1 \leq K \leq J$.

Algoritmo de romber

3.3.5 Regla trapezoidal recursiva

Para aproximar:

$$\int_a^b f(x)dx \approx \frac{h}{2} \sum_{k=1}^{2^J} f(x_{k-1}) + f(x_k)$$

Mediante el uso de la regla del trapecio y sucesivamente cada vez mayor el número de subintervalos de $[a, b]$. En la iteración J -ésima para $f(x)$ en los 2^J+1 puntos igualmente espaciados.

Algoritmo trapezoidal

3.4 FÓRMULAS GAUSSIANAS

Presentación general

$$\int_a^b \mu(x) f(x) dx \cong \sum_{i=1}^n a_i f(x_i)$$

Con $\mu(x)$ una función de peso, que en el caso de $\mu(x) = 1$ se presenta la forma más simple.

$$a_i = \int_a^b \mu(x) L_i(x) dx \quad L_i(x) = \text{función multiplicadora de Lagrange.}$$

Los x_1, \dots, x_n representan los ceros del polinomio $P_n(x)$, de grado n , de una familia que verifica ortogonalidad:

$$\int_a^b \mu(x) P_n(x) P_m(x) dx = 0 \quad \text{para} \quad m \neq n$$

con los coeficientes dependientes de $\mu(x)$, entonces $\mu(x)$ ejerce influencia sobre a_i, x_i aunque no esté taxativamente presente en la fórmula de Gauss.

3.4.1 Fórmula de Gauss-Legendre

$$\mu(x) = 1$$

$$(a, b) = (-1, 1)$$

Los polinomios ortogonales son polinomios de Legendre

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n \quad P_0(x) = 1$$

$$x_i \text{ raíces del polinomio y } a_i = \frac{2(1-x_i^2)}{n^2 |P_{n-1}(x_i)|^2}$$

Tanto los x_i como los a_i están tabulados para reemplazar en

$$\int_a^b f(x)dx \cong \sum_{i=1}^n a_i f(x_i)$$

Algoritmo gausslegendre

3.4.2 Formula de Gauss-Laguerre

$$\int_0^{\infty} e^{-x} f(x) dx \cong \sum_{i=1}^n a_i f'(x_i)$$

$$L_n(x) = e^x \frac{d^n}{dx^n} (e^{-x} x^n)$$

Y

$$a_i = \frac{(n!)^2}{x_i [L'(x_i)]^2}$$

3.4.3 Formula de Gauss-Chebyshev

$$\int_{-1}^1 \frac{f(x)}{\sqrt{1-x^2}} dx \cong \left(\frac{\pi}{n} \right) \sum_{i=1}^n f(x_i)$$

El polinomio n-simo de Chebyshev: $T_n(x) = \cos(n \arccos(x))$

x_i los ceros del polinomio $T_n(x)$.

3.5 Cuadratura adaptativa utilizando la regla de Simpson's

Para aproximar la integral:

$$\begin{aligned} \int_a^b f(x) dx \approx \sum_{k=1}^M & (f(x_{4k-4}) + 4f(x_{4k-3}) + 2f(x_{4k-2}) \\ & + 4f(x_{4k-1}) + f(x_{4k})) \end{aligned}$$

La regla compuesta de Simpson's es ampliada para los $4M$ subintervalos $[x_{4k-4}, x_{4k}]$, donde $[a,b]=[x_0,x_{4M}]$ y $x_{4k-4+j}=x_{4k-4}+jh_k$, para cada $k=1,\dots,M$ y $J=1,\dots,4$.

[Algoritmo adapt](#)

3.6 Regla de Simpson's

El algoritmo "srule", es una modificación de la regla de Simpson's. El resultado es un vector Z que contiene los resultados de la regla de Simpson en el intervalo $[a_0, b_0]$. El algoritmo "adapt" utiliza srule para llevar a cabo la regla de Simpson's en cada uno de los subintervalos generado por el proceso de cuadratura adaptativa.

[Algoritmo srule](#)

3.7 Cuadratura de Gauss-Legendre

Para aproximar la integral:

$$\int_a^b f(x)dx \approx \frac{b-a}{2} \sum_{k=1}^N W_{N,k} f(t_{N,k})$$

Por muestreo $f(x)$ en los N puntos desigualmente espaciados $\{t_{N,k}\}_{k=1}^N$.

Los cambios de variable:

$$t = \frac{a+b}{2} + \frac{b-a}{2}x \quad y \quad dt = \frac{b-a}{2}dx$$

Son usados. Las abscisas $\{x_{N,k}\}_{k=1}^N$ y el correspondiente peso $\{W_{N,k}\}_{k=1}^N$ se deben obtener de una tabla de valores conocidos.

[Algoritmo gauss_q](#)

ECUACIONES DIFERENCIALES ORDINARIAS

4.1 METODOS PARA RESOLVER P.V.I.

$$\text{P.V.I. } y' = f(t, y) \quad a \leq t \leq b \quad y(a) = \alpha$$

En esta ecuación f es una función real dada de en t , donde y es una función incógnita de variable independiente t . Tanto y como f pueden ser vectores, en cuyo caso tendremos un sistema de ecuaciones diferenciales de primer orden.

Una ecuación diferencial conjuntamente con una condición inicial constituye un problema de valor inicial; siendo:

$$\begin{cases} y' = f(t, y) \\ y(x_0) = y_0 \end{cases}$$

4.1.1 Método de Euler

Se basa en la ecuación diferencial, en el tamaño de paso h y en el paso particular de la aproximación, con el error local de truncamiento $= O(h)$.

$$w_{i+1} = w_i + hf(t_i, w_i) \quad i = 0, 1, 2, \dots, N-1, \text{ (ecuación de diferencias)}$$

h = tamaño de paso, con distribución uniforme en $[a, b]$.

$$t_i = a + ih, \quad \text{para cada } i = 0, 1, 2, \dots, N.$$

$w_i \approx y(t_i)$ denota el valor exacto de la solución en t_i
se suele emplear para el método una expresión del tipo:

$$\mu_0 = \alpha + \delta_0$$

$$\mu_{i+1} = \mu_i + hf(t_i, w_i) + \delta_{i+1} \quad i = 0, 1, 2, \dots, N-1$$

Siendo δ_i error de redondeo asociado con μ_i .

El mínimo valor de $E(h)$ se da para $h = \sqrt{2\delta/M}$

O sea h superiores inducirán a incrementar el error total en la aproximación.

4.1.2 Serie de Taylor

El método de Euler es el método de Taylor de orden uno.
Para órden dos:

$$w_0 = 0.5,$$

$$w_{i+1} = w_i + h \left[\left(1 + \frac{h}{2}\right)(w_i - t_i^2 + 1) - ht_i \right]$$

Para orden cuatro:

$$w_0 = 0.5,$$

$$w_{i+1} = w_i + h \left[\left(1 + \frac{h}{2} + \frac{h^2}{6} + \frac{h^3}{24}\right)(w_i - t_i^2) - \left(1 + \frac{h}{3} + \frac{h^2}{12}\right)ht_i + 1 + \frac{h}{2} + \frac{h^2}{6} + \frac{h^3}{24} \right]$$

$$i = 0, 1, \dots, N-1.$$

para ζ_i en (t_i, t_{i+1})

$$\tau_{i+1}(h) = \frac{y_{i+1} - y_i}{h} - T^{(n)}(t_i, y_i) = \frac{h^n}{(n+1)!} f^{(n)}(\zeta_i, y(s_i)) \quad i = 0, 1, 2, \dots, N-1$$

con el error local de truncamiento = $O(h^n)$ $n = \text{derivada admitida}$.

4.1.3 Métodos de Runge-Kutta

Fórmulas más difundidas

$$\kappa_1 = hf(t, y)$$

$$\kappa_2 = hf\left(t + \frac{h}{2}, y + \frac{\kappa_1}{2}\right) \quad O(h^2)$$

$$\kappa_3 = hf\left(t + \frac{h}{2}, y + \frac{\kappa_2}{2}\right) \quad O(h^3)$$

$$\kappa_4 = hf(t + h, y + \kappa_3) \quad O(h^4)$$

$$y(t+h) \approx y(t) + \frac{1}{6}(\kappa_1 + 2\kappa_2 + 2\kappa_3 + \kappa_4) \quad \text{cuarto orden}$$

[Algoritmo rk4](#)

4.1.4 MÉTODOS MULTIPASO

Dado el P.V.I. $y' = f(t, y) \quad a \leq t \leq b \quad y(a) = b$

La ecuación de diferencias w_{i+1} en t_{i+1} ($m \in \mathbb{Z}^+, y \geq 1$) será:

$$w_{i+1} = c_{m-1}w_i + c_{m-2}w_{i-1} + \dots + c_0w_{i+1-m} + h[b_m f(t_{i+1}, w_{i+1}) + b_{m-1}f(t_i, w_i) + \dots + b_0f(t_{i+1-m}, w_{i+1-m})]$$

$m = \text{paso} > 1$; $t_i = \text{paso de red}$; $h = \frac{(b-a)}{N}$; a_0, a_1, \dots, a_{m-1} y b_0, b_1, \dots, b_m son constantes.

$$i = m-1, m, \dots, N-1$$

Valores iniciales dados $w_0 = \alpha, w_1 = \alpha_1, w_2 = \alpha_2, \dots, w_{m-1} = \alpha_{m-1}$

4.1.4.1 Métodos de Adams-Bashforth: EXPLICITOS

A. De dos pasos

$$w_0 = \alpha \quad w_1 = \alpha_1$$

$$w_{i+1} = w_i + \frac{h}{2} [3f(t_i, w_i) - f(t_{i-1}, w_{i-1})] \quad i = 1, 2, \dots, N-1$$

El error local de truncamiento es $\tau_{i+1}(h) = \frac{5}{12} y^{(3)}(\mu_i) h^2$ con algún μ_i entre (t_{i-1}, t_{i+1}) .

B. de tres pasos

$$w_0 = \alpha \quad w_1 = \alpha_1 \quad w_2 = \alpha_2$$

$$w_{i+1} = w_i + \frac{h}{12} [23f(t_i, w_i) - 16f(t_{i-1}, w_{i-1}) + 5f(t_{i-2}, w_{i-2})]$$

$$i = 2, 3, \dots, N-1$$

El error local de truncamiento es $\tau_{i+1}(h) = \frac{3}{8} y^{(4)}(\mu_i) h^3$ con algún μ_i entre (t_{i-2}, t_{i+1}) .

C. De cinco pasos

$$w_0 = \alpha, w_1 = \alpha_1, w_2 = \alpha_2, w_3 = \alpha_3, w_4 = \alpha_4$$

$$w_{i+1} = w_i + \frac{h}{720} [1901f(t_i, w_i) - 2774f(t_{i-1}, w_{i-1}) + 2616f(t_{i-2}, w_{i-2}) - 1274f(t_{i-3}, w_{i-3}) + 251f(t_{i-4}, w_{i-4})]$$

$$i = 4, 5, \dots, N-1$$

El error local de truncamiento es $\tau_{i+1}(h) = \frac{95}{288} y^{(6)}(\mu_i) h^5$ con algún μ_i entre (t_{i-4}, t_{i+1}) .

Para los implícitos se debe usar $(t_{i+1}, f(t_{i+1}, y(t_{i+1})))$ como un nodo de interpolación

extra para aproximar la integral $\int_{t_i}^{t_{i+1}} f(t, y(t)) dt$.

4.1.4.2 Métodos de Adams-Moulton: IMPLICITOS

A. De dos pasos

Las diferencias ordinarias de orden 1 y 2, están dadas por:

$$\Delta f_n = f_{n+1} - f_n,$$

$$\Delta^2 f_n = f_{n+2} - 2f_{n+1} + f_n$$

para sustituir a las diferencias ordinarias en la expresión anterior y agrupando términos semejantes, obtenemos:

$$\text{Dados } w_0 = \alpha \quad \text{y} \quad w_1 = \alpha_1$$

$$w_{i+1} = w_i + \frac{h}{12} [5f(t_{i+1}, w_{i+1}) + 8f(t_i, w_i) - f(t_{i-1}, w_{i-1})]$$

$$i = 1, 2, \dots, N-1$$

El error local de truncamiento es:

$$\tau_{i+1}(h) = -\frac{1}{24} y^{(4)}(\mu_i) h^3 \quad \text{con } t_{i-1} < \mu_i < t_{i+1}$$

B. De tres pasos

$$w_0 = \alpha \quad w_1 = \alpha_1 \quad w_2 = \alpha_2$$

$$w_{i+1} = w_i + \frac{h}{24} [9f(t_{i+1}, w_{i+1}) + 19f(t_i, w_i) - 5f(t_{i-1}, w_{i-1}) + f(t_{i-2}, w_{i-2})]$$

$$i = 1, 2, \dots, N-1$$

$$\text{El error local de truncamiento es: } \tau_{i+1}(h) = -\frac{19}{720} y^{(5)}(\mu_i) h^4 \quad \text{con } t_{i-2} < \mu_i < t_{i+1}$$

C. De cuatro pasos

$$w_0 = \alpha \quad w_1 = \alpha_1 \quad w_2 = \alpha_2 \quad w_3 = \alpha_3$$

$$w_{i+1} = w_i + \frac{h}{720} [251f(t_{i+1}, w_{i+1}) + 646f(t_i, w_i) - 264f(t_{i-1}, w_{i-1}) + 106f(t_{i-2}, w_{i-2}) - 19f(t_{i-3}, w_{i-3})]$$

$$i = 3, 4, \dots, N-1$$

El error local de truncamiento es:

$$\tau_{i+1}(h) = -\frac{3}{160} y^{(6)}(\mu_i) h^5 \quad \text{con } t_{i-3} < \mu_i < t_{i+1}$$

[Algoritmo abm](#)

4.1.5 Método de Milne

Técnicas denominadas de predicción-corrección, empleando el explícito para predecir la aproximación y la implícita para corregirla.

Otros métodos multipasos se generan por integración o interpolación de polinomios en intervalos $[t_j, t_{i+1}]$ con $j \leq i-1$ para aproximar a $y(t_{i+1})$.

Así está el método de Milne, entre $[t_{i-3}, t_{i+1}]$:

$$w_{i+1} = w_{i-3} + \frac{4h}{3} [2f(t_i, w_i) - f(t_{i-1}, w_{i-1}) + 2f(t_{i-2}, w_{i-2})]$$

Algoritmo milne

4.1.6 Predictor-Corrector

Si usamos el primer orden (explícito) el método de Euler hacia adelante para inicializar el método de Crank-Nicolson, obtenemos el método de Heun (también llama método de Euler mejorado), que es un método de segundo orden explícito de Runge-Kutta

$$u_{n+1}^* = u_n + hf_n$$

$$u_{n+1} = u_n + \frac{h}{2} [f_n + f(t_{n+1}, u_{n+1}^*)]$$

El paso explícito constituye un factor de predicción, mientras que la implícita se llama corrector. Este tipo de métodos disfrutan de la orden de precisión del método corrector. Sin embargo, el ser explícito, se somete a una restricción de estabilidad que suele ser el mismo que el del método predictor. Por lo tanto, no son adecuados para integrar un problema de Cauchy en intervalos no acotados.

Algoritmo heun

4.1.7 USO DE LA EXTRAPOLACIÓN

condición inicial $w_0 = \alpha = y(a)$ y para w_1 se utiliza la técnica de Euler.

$$w_{i+1} = w_{i-1} + 2hf(t_i, w_i) \quad i \geq 1$$

Al llegar al valor t , se corrige el punto extremo que comprende las dos aproximaciones del punto medio, generando una aproximación $w(t, h)$ a $y(t)$ dada por.

$$y(t) = w(t, h) + \sum_{k=1}^{\infty} \delta_k h^{2k}$$

Con δ_k ctes., ligadas a las derivadas de $y(t)$, pero es independiente de h .

SISTEMAS LINEALES

$$A\vec{x} = \vec{b}$$

Planteo generalizado:

$$\begin{array}{l} R_1 \quad a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ R_2 \quad a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ R_n \quad a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{array}$$

5.1 REPRESENTACIÓN DE UN SISTEMA LINEAL. SUSTITUCIÓN HACIA ATRÁS

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{array}$$

Un sistema puede llevarse a la forma:

$$\bar{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ 0 & \ddots & a_{22} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \dots & a_{nn} & a_{n,n+1} \end{bmatrix} \quad \text{con } a_{i,n+1} = b_i \quad \text{para } i = 1, 2, \dots, n$$

Permitiendo la sustitución hacia atrás.

Sustitución hacia atrás, solamente si todos los elementos de la diagonal son no ceros.

Primero calcula $x_N = b_N / a_{NN}$ y entonces realiza:

$$x_k = \frac{b_k - \sum_{j=k+1}^N a_{kj}x_j}{a_{kk}} \quad \text{para } k = N-1, N-2, \dots, 1.$$

Algoritmo backsub

Triangularización superior, seguido por la sustitución hacia atrás. Para construir la solución $AX = B$, primero reducción de la matriz aumentada $[A | B]$ a la forma triangular superior y luego realizar la sustitución hacia atrás

Algoritmo uptrbk (utilizado por backsub)

5.2 TÉCNICA DE GAUSS-JORDAN

Partiendo de una matriz aumentada, por hipótesis tenemos

$$a_{11} \neq 0, \text{ pues } \det(A) \neq 0$$

La solución se obtiene usando $x_i = \frac{a_{i,n+1}^{(i)}}{a_{ii}^{(i)}}$ para cada $i = 1, 2, 3, \dots, n$.

Prescindiendo de la sustitución hacia atrás en la eliminación gaussiana.

Requiere $\left(\frac{n^3}{2} + n^2 - \frac{n}{2}\right)$ productos/divisiones y $\left(\frac{n^3}{2} - \frac{n}{2}\right)$ sumas/ restas.

[Algoritmo gjelim](#)

5.3 PIVOTEO

Una pista para seleccionar es tomar el menor $p \geq k$ tal que:

$$|a_{p,k}^{(k)}| = \max_{k \leq i \leq h} |a_{i,k}^{(k)}| \quad \text{y efectuar } (R_k) \leftrightarrow (R_p)$$

(Técnica parcial de pivoteo)

Otra técnica es la de reescalamiento de columnas. El primer paso del procedimiento consiste en definir, para cada renglón, un factor escalar s_i

Por medio de:

$s_i = \max |a_{ij}|$ con $1 \leq j \leq n$ cuando $s_i = 0$ (solución múltiple); de no ser así, el intercambio adecuado de renglones para poner ceros en la primer columna se determina seleccionando el menor entero k con:

$$\frac{|ak_1|}{s_k} = \max_{j=1,2,\dots,n} \frac{|a_{i,j}|}{s_j}$$

Y se realiza $(R_1) \leftrightarrow (R_k)$

[Algoritmo gauss con pivoteo](#)

5.4 FACTORIZACION MATRICIAL

Si A es no singular, admite el producto SI (triangular superior por triangular inferior), la factorización es de gran provecho, para la resolución de sistemas lineales.

Si la eliminación gaussiana para $A\vec{x} = \vec{b}$ se efectúa sin cambios de filas, A podrá factorizarse como el producto IS .

$$A=IS$$

Con

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ m_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ m_{n1} & \cdots & m_{n,n-1} & 1 \end{bmatrix} \quad \text{Matriz triangular inferior } c_{ij} = 0 \text{ para } i < j$$

$$S = \begin{bmatrix} a_{11}^{(1)} & & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & a_{22}^{(2)} & \ddots & \vdots \\ \vdots & \ddots & & \ddots & a_{n-1,n}^{(n-1)} \\ 0 & \cdots & 0 & & a_{n,n}^{(n)} \end{bmatrix} \quad \text{Matriz triangular superior } c_{ij} = 0 \text{ para } i > j$$

La solución para $ISx=B$ puede ser obtenida definiendo $y=Sx$, entonces se resuelven los dos sistemas:

- 1°) $Iy=B$ para y
- 2°) $Sx=y$ para x

5.4.1 Factorización LU

Tenemos que demostrar que la descomposición se puede hacer a una matriz de orden $n = k$. Es entonces la matriz de orden k . Empezamos esta serie en sub-matrices de la forma

$$A = \begin{pmatrix} A_{k-1} & r \\ s^t & a_{kk} \end{pmatrix}$$

Donde r y s son vectores columna, ambos con $k - 1$ componentes. Tenga en cuenta que la matriz A_{k-1} es de orden $k-1$. Por lo tanto, por hipótesis de inducción esta puede ser descompuesta en forma $A_{k-1} = L_{k-1}U_{k-1}$. Utilizando las matrices L_{k-1} y U_{k-1} formamos las siguientes matrices:

$$L = \begin{pmatrix} L_{k-1} & 0 \\ m^t & 1 \end{pmatrix}; \quad U = \begin{pmatrix} U_{k-1} & p \\ 0 & u_{kk} \end{pmatrix}$$

Donde m y p son vectores columna, ambos con $k-1$ componentes. Tenga en cuenta que m , p y u_{kk} son desconocidos.

Así, se establece que la matriz A se puede descomponer en LU trataremos de determinarlos.

Realización del producto LU, se deduce que:

$$LU = \begin{pmatrix} L_{k-1}U_{k-1} & L_{k-1}p \\ m^t S_{k-1} & m^t p + u_{kk} \end{pmatrix}$$

Estudiamos ahora la ecuación $LU=A$, esto es:

$$\begin{pmatrix} L_{k-1}U_{k-1} & L_{k-1}p \\ m^t U_{k-1} & m^t p + u_{kk} \end{pmatrix} = \begin{pmatrix} A_{k-1} & r \\ s^t & a_{kk} \end{pmatrix}$$

De esta igualdad concluimos

$$L_{k-1}U_{k-1} = A_{k-1},$$

$$L_{k-1}p = r,$$

$$m^t U_{k-1} = s^t,$$

$$m^t p + u_{kk} = a_{kk}.$$

Tenga en cuenta que la primera ecuación tiene por hipótesis la inducción, y por lo tanto L_{k-1} y U_{k-1} son determinadas de manera única. Además, ni L_{k-1} ni U_{k-1} son singulares (o A_{k-1} también sería singular, contradiciendo la hipótesis). Por lo tanto:

$$L_{k-1}p = r \Rightarrow p = L_{k-1}^{-1}r,$$

$$m^t U_{k-1} = s^t \Rightarrow m^t = s^t U_{k-1}^{-1},$$

$$m^t p + u_{kk} = a_{kk} \Rightarrow u_{kk} = a_{kk} - m^t p$$

Por lo tanto p, m y u_{kk} son determinadas unívocamente en este orden, L y U son determinados únicamente. Al final:

$$\det(A) = \det(L) * \det(U) = 1 * \det(U_{k-1}) * u_{kk}$$

$$= u_{11} u_{22} \dots u_{k-1,k-1} u_{kk},$$

Cabe señalar que la descomposición LU constituye uno de los algoritmos más eficientes para el cálculo del determinante de una matriz.

La resolución del sistema $Ax = b$ implica hacer $Ly = b$, $Ux = y$

Factorización con pivoteo

Algoritmo lufact

Según determinadas variaciones para la técnica se conocen los métodos:

- **Doolittle**: requiere que $i_{11} = i_{22} = \dots = i_{nn} = 1$
- **Crout**: requiere que los elementos de la diagonal principal de S sean todos 1.
- **Choleski**: requiere $i_{ii} = s_{ii}$ para cada i .

Este tipo de factorización presentado es ventajoso cuando no se requiere el intercambio de filas para ajustar el error de redondeo.

Matrices de permutación

Se utiliza cuando se requieren intercambios de renglones

- Si P es matriz de permutación tiene inversa y $P^{-1} = P^t$.
- Para cualquier A , no singular, habrá una P tal que $PA\vec{x} = P\vec{b}$ se resuelve sin intercambiar filas, con lo que PA se puede factorizar como $PA=IS$

Y al ser $P^{-1} = P^t \Rightarrow A = (P^t I) S$

$(P^t I)$, no será triangular inferior (salvo que $P=Id$).

Ejemplo:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ 2/3 & 0 & 1 \end{pmatrix}, \quad S = \begin{pmatrix} 3 & 3 & 1 \\ 0 & -2 & 14/3 \\ 0 & 0 & -5/3 \end{pmatrix}$$

P será

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Matrices características

- a) **Choleski** : para una definida positiva requiere menos operaciones que LDL' pero implica hallar n raíces cuadradas; con L con $a_{ii} = 1$ y D diagonal con $a_{ii} > 0$. A puede factorizarse como LDL' .
- b) **Doolittle o de Crout** : para matrices de banda que disponen sus elementos no nulos en derredor de la diagonal. Las más comunes, $p=q=2$ y los de $p=q=4$, pentadiagonales.
Las primeras, asociadas a aproximaciones de splines cúbicos o aproximaciones lineales por parte en problemas de valores de contorno, los pentadiagonales de utilidad para problemas de valores de frontera.
El trabajar con matrices de banda facilita los algoritmos de factorización (por el número de operaciones).

5.5 TÉCNICAS DE REPETICIÓN PARA SISTEMAS LINEALES

Transformar el sistema original $A\vec{x} = \vec{b}$ en uno equivalente $\vec{x} = W\vec{x} + \vec{c}$ para cierta matriz fija W y un vector \vec{c} .

La sucesión se va calculando como $\vec{x}^{(k)} = W\vec{x}^{(k-1)} + \vec{c} \quad k = 1, 2, 3, \dots$,

Métodos iterativos de resolución de sistemas lineales

Estas técnicas consisten en transformar un sistema de la forma

$$Au = b$$

en una ecuación de punto fijo de la forma: $u = Mu + c$

de tal manera que, al hacer iteraciones de la forma:

$$u^n = M u^{n-1} + c$$

se obtenga que u^n converge hacia u , la solución del sistema original.

Consideremos el sistema de ecuaciones

$$\begin{aligned} 2x - y &= 1 \\ -x + 2y - z &= 0 \\ -y + 2z &= 1 \end{aligned}$$

Buscar la solución de este sistema es equivalente a buscar un vector $u = (x, y, z)$ que verifique que

$$x = \frac{1+y}{2}, \quad y = \frac{x+z}{2}, \quad z = \frac{1+y}{2}$$

Hacer iteraciones de esta ecuación de punto fijo consiste en partir de una aproximación inicial (x_1, y_1, z_1) y hacer iteraciones de la forma:

$$x_n = \frac{1+y_{n-1}}{2}, \quad y_n = \frac{x_{n-1}+z_{n-1}}{2}, \quad z_n = \frac{1+y_{n-1}}{2}$$

En este caso, la solución exacta del sistema es $u = (1, 1, 1)$.

Si hacemos iteraciones del esquema anterior a partir de la aproximación inicial $u^1 = (0, 0, 0)$, obtenemos que

$$x_2 = \frac{1+0}{2} = \frac{1}{2}, y_2 = \frac{0+0}{2} = 0, z_2 = \frac{1+0}{2} = \frac{1}{2}$$

las sucesivas iteraciones se van aproximando a la solución $u = (1, 1, 1)$.

Si el esquema iterativo $u^n = M u^{n-1} + c$ converge hacia un vector u , entonces u verifica que $u = Mu + c$

Existen diferentes métodos para convertir un sistema de la forma $Au = b$ en una ecuación de punto fijo $u = Mu + c$. Todas se basan en descomponer A de la forma $A = L + D + U$, donde D es la matriz diagonal que corresponde a la parte diagonal de A , L es la matriz triangular inferior que corresponde a la parte de A situada por debajo de la diagonal, y U es la matriz triangular superior que corresponde a la parte de A situada por encima de la diagonal.

5.5.1 Técnica de Jacobi

Este método consiste en tomar

$$M_j = D^{-1}(-L - U)$$

$$c_j = D^{-1}b$$

Es el que se ha utilizado en el ejemplo anterior. El paso de una iteración a otra del método de Jacobi puede expresarse de la siguiente forma:

$$u_1^n = \frac{-a_{12}u_2^{n-1} - \dots - a_{1N}u_N^{n-1} + b_1}{a_{11}}$$

$$u_2^n = \frac{-a_{21}u_1^{n-1} - a_{23}u_3^{n-1} \dots - a_{2N}u_N^{n-1} + b_2}{a_{22}}$$

$$u_N^n = \frac{-a_{N1}u_1^{n-1} - a_{N2}u_2^{n-1} \dots - a_{NN-1}u_{N-1}^{n-1} + b_N}{a_{NN}}$$

[Algoritmo jacobi](#)

5.5.2 Técnica de Gauss-Seidel

Este método consiste en tomar:

$$M_{GS} = (D + L)^{-1}(-U)$$

$$c_{GS} = (D + L)^{-1}b$$

A efectos prácticos, la aplicación de este método no requiere el cálculo directo de la matriz inversa $(D + L)^{-1}$, puesto que el paso de una iteración a otra puede hacerse de la siguiente forma:

$$u_1^n = \frac{-a_{12}u_2^{n-1} - \dots - a_{1N}u_N^{n-1} + b_1}{a_{11}}$$

$$u_2^n = \frac{-a_{21}u_1^n - a_{23}u_3^{n-1} \dots - a_{2N}u_N^{n-1} + b_2}{a_{22}}$$

$$u_N^n = \frac{-a_{N1}u_1^n - a_{N2}u_2^n \dots - a_{NN-1}u_{N-1}^n + b_N}{a_{NN}}$$

Si hacemos un barrido para el cálculo de la solución de arriba hacia abajo, y vamos actualizando las componentes del vector aproximación según las vamos calculando, obtenemos el método de Gauss-Seidel. Por tanto, básicamente, podemos decir que la diferencia entre el método de Gauss-Seidel y el método de Jacobi es que en el método de Gauss-Seidel se actualiza el vector aproximación después del cálculo de cada componente, y en el caso de Jacobi se actualiza sólo al final, después de haber calculado todas las componentes por separado.

[Algoritmo gseid](#)

5.6 Técnica de relajación

El objetivo de este método es intentar mejorar el método de Gauss-Seidel introduciendo un parámetro de relajación w . Se toman, en este caso:

$$M_w = (D + wL)^{-1} ((1 - w)D - wU)$$

$$c_w = w(D + wL)^{-1} b$$

Estas nuevas matrices permiten realizar un promediado entre el resultado obtenido por Gauss-Seidel y el estado de la solución en la etapa anterior, de la forma siguiente:

$$u_1^n = w \frac{-a_{12}u_2^{n-1} - \dots - a_{1N}u_N^{n-1} + b_1}{a_{11}} + (1-w)u_1^{n-1}$$

$$u_2^n = w \frac{-a_{21}u_1^n \dots - a_{2N}u_N^{n-1} + b_2}{a_{22}} + (1-w)u_2^{n-1}$$

$$u_N^n = w \frac{-a_{N1}u_1^n \dots - a_{NN-1}u_{N-1}^n + b_N}{a_{NN}} + (1-w)u_N^{n-1}$$

La elección del parámetro w es, en general, un problema difícil. Sin embargo, en el caso de matrices tridiagonales, es decir, matrices con todos los elementos nulos salvo la diagonal principal y sus codiagonales, el siguiente resultado muestra la forma de calcular el valor óptimo de w .

Si A es una matriz tridiagonal y $\rho(M_j) < 1$, entonces el valor de w que optimiza la velocidad de convergencia del método es:

$$w_{opt} = \frac{2}{1 + \sqrt{1 - \rho(M_j)^2}}$$

Como puede observarse de la expresión anterior, el valor de w_{opt} se encuentra siempre entre 1 y 2.

CONCLUSIÓN: el teorema se concluye teniendo en cuenta que cualquier norma de una matriz es siempre mayor o igual que su radio espectral.

Este resultado se puede generalizar un poco al caso de matrices irreducibles de la siguiente forma:

A) Una matriz A es irreducible si un sistema de la forma $Au = b$ no puede descomponerse en dos sub-sistemas independientes de dimensión menor.

B) Una matriz es irreducible si el cambio de cualquier valor del vector b del sistema $Au = b$ afecta a todos los elementos del vector u .

Si A es una matriz irreducible y se verifica que

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \forall i$$

ó

$$|a_{jj}| \geq \sum_{i \neq j} |a_{ij}| \quad \forall j$$

Con la desigualdad estricta en al menos una fila o columna, entonces los métodos iterativos convergen.

OTRA FORMA DE PLANTEAR LA TÉCNICA DE RELAJACIÓN

$$\vec{x}_i^{(k)} = \vec{x}_i^{(k-1)} + q \frac{r_{ii}^{(k)}}{a_{ii}}$$

vector residuo $\vec{r}_{i+1}^{(k)}$ asociado al vector, definido por $\vec{r} = \vec{b} - A\vec{x}$ entonces habrá un $\vec{r}_i^{(k)} = (\vec{r}_{1i}^{(k)}, \vec{r}_{2i}^{(k)}, \dots, \vec{r}_{ni}^{(k)})^t$ que se busca converja a cero.

- Si $0 < q < 1$, técnicas de subrelajación, para sistemas no convergentes por G-S.
- $q > 1$, para acelerar convergencia de sistemas convergentes por G-S (métodos SOR o sobrerelajación).

Para el cálculo se usará:

$$\vec{x}_i^{(k)} = (1-q)\vec{x}_i^{(k-1)} + \frac{q}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} (a_{ij}\vec{x}_j^{(k)}) - \sum_{j=i+1}^n (a_{ij}\vec{x}_j^{(k-1)}) \right]$$

Matricialmente

$$\vec{x}^{(k)} = (D - qI)^{-1} [(1-q)D + qS] \vec{x}^{(k-1)} + q(D - qI)^{-1} \vec{b}$$

5.7 GRADIENTE CONJUGADO

Se busca una secuencia de n direcciones conjugadas, y luego se calculan los coeficientes.

La solución \mathbf{x}^* de $A\vec{x} = \vec{b}$ es también el único minimizador de:

$$f(x) = \frac{1}{2} \vec{x}^T A \vec{x} - \vec{b}^T \vec{x}, \quad \vec{x} \in \mathbb{R}^n$$

$$r_k = \vec{b} - A\mathbf{x}_k$$

r_k es el negativo del gradiente de f en $\mathbf{x} = \mathbf{x}_k$, con lo que se puede mostrar, ya que las direcciones p_k son conjugadas entre sí, que una expresión para tales direcciones es:

$$p_{k+1} = r_k - \frac{p_k^T A r_k}{p_k^T A p_k} p_k$$

[Algoritmo conjugad](#)

APROXIMACIÓN

6.1 POLINOMIOS DE CHEBYSCHEV

$p(x) = a_0 H_0(x) + \dots + a_m H_m(x)$ polinomio de aproximación

6.2 APROXIMACIÓN MÍNIMO –MÁXIMO (O DE CHEBYSHEV)

$$P(x) = mx + c$$

$$m = \frac{y(b) - y(a)}{b - a} \quad c = \frac{y(a) + y(b)}{2} - \frac{(a + b)[y(b) - y(a)]}{2(b - a)}$$

6.3 APROXIMACIÓN POR VALORES PROPIOS

El teorema de descenso Gerschgorin es útil para determinar los límites globales a los valores propios de una matriz $A_{n \times n}$. El término "global", los límites que encierran todos los valores propios.

Dada $A_{n \times n}$ $R_i = \left\{ z \in \mathbb{C} / |z - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}| \right\}$ R_i es el círculo en el plano complejo con a_{ii} como centro.

Los auto-valores de A están dentro de $R = \bigcup_{i=1}^n R_i$ (círculo de Gerschgorin)

El función Gerschgorin devuelve la parte inferior y superior de los límites globales a los valores propios de una matriz tridiagonal simétrica A.

[Algoritmo Gerschgorin](#)

6.4 Técnicas de Aproximación

6.4.1 Métodos de Potencias

Sea una matriz A que posee una base de autovectores tal que en módulo su autovalor máximo λ_{\max} es único. Sea un vector u^1 no ortogonal al subespacio engendrado por los autovectores del autovalor λ_{\max} , entonces, si definimos la secuencia

$$\lim_{n \rightarrow \infty} \text{sign}((u^n, u^{n-1})) \|u^n\| = \lambda_{\max}$$

$\lim_{n \rightarrow \infty} (\text{sign}((u^n, u^{n-1})))^n \frac{u^n}{\|u^n\|}$ es un autovector de λ_{\max}

$$u^n = A \frac{u^{n-1}}{k u^{n-1} k}$$

Además, dicho autovector tiene norma 1.

$\text{sign}((u^n, u^{n-1}))$ es el signo del producto escalar de u^n y u^{n-1} , es decir $\text{sign}((u^n, u^{n-1})) = 1$ si $(u^n, u^{n-1}) \geq 0$ y $\text{sign}((u^n, u^{n-1})) = -1$ si $(u^n, u^{n-1}) < 0$

Para calcular el valor propio dominante λ_1 y su vector propio asociado V_1 para la matriz $A_{n \times n}$. se supone que los valores propios λ_i tienen la propiedad de posición dominante $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_n| \geq 0$.

[Algoritmo power1](#)

6.4.2 Método de la potencia inversa

El método anterior también se puede utilizar para el cálculo del autovalor de módulo menor λ_{\min} , teniendo en cuenta que:

$$\lambda_{\min} = \frac{1}{\max\{\lambda_i' \text{ autovalores de } A^{-1}\}}$$

Por tanto, si aplicamos el método anterior a A^{-1} , obtenemos que la

secuencia $u = A^{-1} \frac{u^{n-1}}{\|u^{n-1}\|}$ verifica que $\lim_{n \rightarrow \infty} \text{sign}((u^n, u^{n-1})) \frac{u^n}{\|u^n\|} = \frac{1}{\lambda_{\min}}$

$\lim_{n \rightarrow \infty} \text{sign}((u^n, u^{n-1})) \frac{u^n}{\|u^n\|}$ es un autovector de λ_{\min}

En los casos prácticos, se evita calcular directamente A^{-1} , y se obtiene u^n resolviendo el sistema

$$A u^n = \frac{u^{n-1}}{\|u^{n-1}\|}$$

Para calcular el valor propio dominante λ_1 y su vector propio asociado V_j para la matriz $A_{n \times n}$. Se supone que los n valores propios tienen la propiedad $\lambda_1 < \lambda_2 < \dots < \lambda_n$ y que α es un número real tal que:

$$|\lambda_j - \alpha| < |\lambda_i - \alpha| \quad \text{para cada } i = 1, 2, \dots, j-1, j+1, \dots, n$$

[Algoritmo invpow](#)

6.4.3 Método simétrico de Potencias

$$\vec{x}^{(0)} : q = \frac{\vec{x}^{(0)t} A \vec{x}^{(0)}}{\vec{x}^{(0)t} \vec{x}^{(0)}}$$

Para elegir q se considera que si \vec{x} es vector propio de A respecto al valor propio λ , se dará que $A\vec{x} = \lambda\vec{x}$, entonces $\vec{x}^t A \vec{x} = \lambda \vec{x}^t \vec{x}$ y $\lambda = \frac{\vec{x}^t A \vec{x}}{\vec{x}^t \vec{x}}$

6.4.4 METODO CLASICO DE JACOBI

El clásico método de Jacobi, o simplemente el método de Jacobi, es un método numérico utilizado para determinar la auto-valores y la auto-vectores de matrices simétricas. Teniendo en cuenta la matriz A , se realiza una secuencia de rotación

$$A_1 = A ; \quad A_2 = U_1^t A_1 U_1 \rightarrow A_3 = U_2^t A_2 U_2 \rightarrow \\ \rightarrow \dots \rightarrow A_{k+1} = U_k^t A_k U_k \approx D$$

donde U_i , $i = 1, 2, \dots, k$ son matrices de rotación, y D es una matriz diagonal.

La secuencia de las matrices de A_k es calculada a través de una repetición de:

$$A_{k+1} = U_k^t A_k U_k \quad (K = 1, 2, \dots)$$

Como $A_1 = A$, obtenemos:

$$A_{k+1} = U_k^t U_{k-1}^t \dots U_2^t U_1^t A U_1 U_2 \dots U_{k-1} U_k = V^t A V$$

Donde $V = U_1 U_2 \dots U_{k-1} U_k$.

Con la hipótesis de que $A_k \approx D$, obtenemos que $D = V^t A V$, donde V es una matriz ortogonal, porque la matriz V es producto de matrices ortogonales. Así, D contiene la auto-valores de A y V contiene su correspondientes auto-vectores (en columnas), es decir, la j -ésima columna de V es un auto-vector correspondiente a auto-valor λ_j .

Para calcular tanto los auto-valores como los auto-vectores $\{\lambda_j, V_j\}_{j=1}^n$ de la matriz simétrica $A_{n \times n}$:

[Algoritmo jacobi1](#)

6.4.5 Técnicas de Deflación

Se emplean para la determinación de aproximaciones a los otros valores propios de una matriz después de haber encontrado la aproximación al valor propio dominante. Consiste en generar una nueva matriz B , con valores propios iguales a los de A , salvo que el valor propio dominante de A se sustituya por el valor propio 0 en B . La técnica es muy sensible a errores de redondeo.

6.4.6 Técnica de Householder

Para reducir la matriz simétrica $A_{n \times n}$ a la forma tridiagonal mediante el uso de $n - 2$ transformaciones de householder.

La transformación puede ser expresada como:

$$QA'Q = A' - wu^T - uw^T$$

que nos da el siguiente procedimiento de cálculo que ha de llevarse a cabo con:
 $i = 1, 2, \dots, n - 2$.

[Algoritmo house](#)

6.4.7 Algoritmo QR

Supongamos que A es una matriz simétrica real, En la sección anterior vimos cómo el método de Householder se utiliza para construir una matriz tridiagonal similar. El método QR se utiliza para encontrar todos los valores propios de una matriz tridiagonal. rotaciones similares a los que utiliza el método de Jacobi se utilizan para construir una matriz ortogonal $Q_1 = Q$ y una triangular superior $R = R_1$ de manera que $A_1 = A$ tiene la factorización:

$$A^{(i+1)} = R^{(i)} Q^{(i)} = (Q^{(i)T} A^{(i)}) Q^{(i)} = Q^{(i)T} A^{(i)} Q^{(i)}$$

[Algoritmo qr1](#)

[Algoritmo qr2](#)

6.5 Ortogonalización de Gram-Schmidt

Este método permite obtener de una matriz $A_{n \times n}$ una matriz Q con columnas ortonormales y R , una matriz triangular superior de tal forma que $A = Q^*R$.

$$\text{proj}_{\mathbf{u}} \mathbf{v} = \frac{\langle \mathbf{v}, \mathbf{u} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle} \mathbf{u}.$$

[Algoritmo grams](#)

SISTEMAS DE ECUACIONES NO LINEALES

Forma general:

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned}$$

7.1 MÉTODO DE NEWTON

$$\vec{x}^{(k)} = F(\vec{x}^{(k-1)}) = \vec{x}^{(k-1)} - J(\vec{x}^{(k-1)})^{-1} G(\vec{x}^{(k-1)}) \quad k \geq 1$$

Método de Newton para sistemas no lineales, esperando convergencia cuadrática, si se conoce $\vec{x}^{(0)}$ preciso y exista $J(\vec{q})^{-1}$.

$$J(\vec{x}) = \begin{bmatrix} \frac{\partial g_1(\vec{x})}{\partial x_1} & \frac{\partial g_1(\vec{x})}{\partial x_2} & \dots & \frac{\partial g_1(\vec{x})}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial g_n(\vec{x})}{\partial x_1} & \frac{\partial g_n(\vec{x})}{\partial x_2} & \dots & \frac{\partial g_n(\vec{x})}{\partial x_n} \end{bmatrix}$$

En la búsqueda de un punto fijo, vimos la transformación $x = g(x)$. Para el caso de funciones de R^n en $R^n \Rightarrow$ Una función G de $D \subset R^n$ en R^n tiene un punto fijo en $p \in D$ si $G(p) = p$

[Algoritmo newtons](#)

7.2 TÉCNICAS CUASI-NEWTON

Método de Broyden

$$A_1 = J(\vec{x}^{(0)}) + \frac{\left[\vec{G}(\vec{x}^{(1)}) - \vec{G}(\vec{x}^{(1)}) - J(\vec{x}^{(1)}) (\vec{x}^{(1)} - \vec{x}^{(0)}) \right] (\vec{x}^{(1)} - \vec{x}^{(0)})^t}{\left\| \vec{x}^{(1)} - \vec{x}^{(0)} \right\|_e^2}$$

$\vec{x}^{(1)} - \vec{x}^{(0)}$ es el vector ortogonal

[Algoritmo broyden](#)

7.3 TÉCNICAS DE MAYOR PENDIENTE

$$\vec{x}^{(1)} = \vec{x}^{(0)} - a \vec{\nabla} g(\vec{x}^{(0)}), \quad a > 0$$

La dirección de la máxima disminución del valor de g en x es la dirección dada por $-\vec{\nabla} g(x)$

7.4 PROBLEMA DE VALOR DE FRONTERA PARA ECUACIONES DIFERENCIALES ORDINARIAS Y EN DERIVADAS PARCIALES

$$y'' = f(x, y, y') \quad \text{en} \quad [a, b]$$

$$\text{Sujeta a: } y(a) = \alpha \wedge y(b) = \beta$$

Cuando $y'' = f(x, y, y')$ en $[a, b]$ tiene la forma:

$$y'' = p(x)y' + q(x)y + r(x) \quad \text{en} \quad [a, b] \quad \text{el problema es lineal,}$$

Si además Satisface:

1. $p(x), q(x), y, r(x)$ continuas en $[a, b]$
2. $q(x) > 0$ en $[a, b]$

El problema tiene solución única.

7.5 TECNICA DE DISPARO PARA EL PROBLEMA LINEAL

$$y_1(x) \Rightarrow y'' = p(x)y' + q(x)y + r(x), \quad a \leq x \leq b, \quad y(a) = \alpha, \quad y'(a) = 0$$

$$y_2(x) \Rightarrow y'' = p(x)y' + q(x)y, \quad a \leq x \leq b, \quad y(a) = 0, \quad y'(a) = 1$$

Ambos problemas tienen solución única y se cumple:

$$y(x) = y_1(x) + \frac{\beta - y_1(b)}{y_2(b)} y_2(x) \quad (\text{Solución única de la ecuación diferencial con valor en la frontera}).$$

[Algoritmo disparo Lineal](#)

7.6 TECNICA DE DISPARO PARA EL PROBLEMA NO LINEAL

$$y''=f(x,y,y'), \quad a \leq x \leq b, \quad y(a)=\alpha, \quad y'(a)=t$$

donde se eligen los parámetros

$$t = t_k \text{ de manera que } \lim_{k \rightarrow \infty} y(b, t_k) = y(b) = \beta$$

la solución en x como en t , requerirá la forma de los PVI como:

- $y''(x,t)=f(x,y(x,t),y'(x,t)), \quad a \leq x \leq b, \quad y(a,t)=\alpha, \quad y'(a,t)=t$ y
- $z''(x,t)=(\partial f / \partial y)(x,y,y')z(x,t)+(\partial f / \partial y')(x,y,y')z'(x,t), \quad a \leq x \leq b, \quad z(a,t)=0, \quad z'(a,t)=1$

simbolizando $z(x,t)$ a $(\partial y / \partial t)(x,t)$

entonces en cada iteración se deberá resolver ambos PVI, con:

$$t_k = t_{k-1} - \frac{y(b, t_{k-1}) - \beta}{z(b, t_{k-1})}$$

Algoritmo disparo no Lineal

7.7 DIFERENCIAS FINITAS PARA PROBLEMAS LINEALES

para la ecuación general de orden dos $y''(x_i) = p(x_i)y'(x_i) + q(x_i)y(x_i) + r(x_i)$ en $[a,b]$ con $\alpha=y(a)$, $\beta=y(b)$ y N el número de etapas.

Al desarrollar y en el tercer polinomio de Taylor evaluando x_i entre x_{i+1} y x_{i-1} , suponiendo que $y \in C^4[x_{i-1}, x_{i+1}]$ tendremos:

$$y(x_{i+1}) = y(x_i + h) = y(x_i) + hy'(x_i) + \frac{h^2}{2} y''(x_i) + \frac{h^3}{6} y'''(x_i) + \frac{h^4}{24} y^{(4)}(\xi_i^+)$$

$$y(x_{i-1}) = y(x_i - h) = y(x_i) - hy'(x_i) + \frac{h^2}{2} y''(x_i) - \frac{h^3}{6} y'''(x_i) + \frac{h^4}{24} y^{(4)}(\xi_i^-)$$

Sumando estas dos ecuaciones y aplicando el teorema del valor intermedio, podemos llegar a las fórmulas de diferencias centradas para $y''(x_i)$ e $y'(x_i)$ (aquí no se detalla).

La utilización de las fórmulas de diferencias centradas de:

$$y''(x_i) = p(x_i)y'(x_i) + q(x_i)y(x_i) + r(x_i) \text{ genera la ecuación:}$$

$$\frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1}))}{h^2} = p(x_i) \left[\frac{y(x_{i+1}) - y(x_{i-1}))}{2h} \right] + \dots$$

$$q(x_i)y(x_i) + r(x_i) - \frac{h^2}{12} [2p(x_i)y'''(\eta_i) - y^{(4)}(\xi_i)]$$

Este método de diferencias finitas con $O(h^2)$ para el error de

truncamiento, junto con las condiciones de frontera $\alpha=y(a)$, $\beta=y(b)$ se emplea

para definir: $w_0 = \alpha$, $w_{N+1} = \beta$ y

$$\frac{-w_{i+1} + 2w_i - w_{i-1}}{h^2} + p(x_i) \left[\frac{w_{i+1} - w_{i-1}}{2h} \right] + \dots$$

$$q(x_i)w_i = -r(x_i) \quad \text{para } i = 1, 2, \dots, N$$

El sistemas de ecuaciones resultantes se expresa en forma de una matriz tridiagonal de $N \times N$ como $Aw = b$.

ECUACIONES EN DERIVADAS PARCIALES

8.1 ECUACIÓN GENERAL

$$[Af_{xx} + 2Bf_{xy} + Cf_{yy}] + Df_x + Ef_y + Ff = G$$

con A, B, \dots, G funciones reales de x e y , en base a su analogía con el discriminante de secciones cónicas:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

se las describirá como hiperbólicas, elípticas y parabólicas.

El objetivo es buscar un método numérico de aproximación de la solución exacta. Se formularán las EDP clásicas en base a su aproximación por diferencias finitas

8.2 ELÍPTICA O DE POISSON

$$\frac{\partial^2 f}{\partial x^2}(x, y) + \frac{\partial^2 f}{\partial y^2}(x, y) = g(x, y) \quad \text{Si } g(x, y) \equiv 0 \quad \text{se obtiene la ecuación de Laplace.}$$

Si la función representa un campo de temperaturas, las restricciones del problema a través de la distribución de temperatura en el borde de la región, son las condiciones de frontera de Dirichlet, dadas por:

$$f(x, y) = r(x, y) \quad \forall (x, y) \text{ en } S, \text{ frontera de la región } U.$$

$$U = \{(x, y) / a < x < b, c < y < d\}$$

El método de resolución consiste en una adaptación del método de diferencias finitas para problemas con valor de frontera tomando:

$$h = \frac{(a-b)}{n} \quad \text{y} \quad k = \frac{(d-c)}{m} \quad \text{obteniendo una cuadrícula con los puntos de red, en}$$

cada punto de red utilizamos la serie de Taylor en la variable x alrededor de x_i para generar la fórmula de la diferencias centradas.

$$\frac{\partial^2 f}{\partial x^2}(x_i, y_j) = \frac{f(x_{i+1}, y_j) - 2f(x_i, y_j) + f(x_{i-1}, y_j))}{h^2} - \frac{h^2}{12} \frac{\partial^4 f}{\partial x^4}(\zeta_i, y_j)$$

$$\zeta_i \in (x_{i-1}, x_{i+1})$$

Lo mismo para la variable y alrededor de y_i

$$\frac{\partial^2 f}{\partial y^2}(x_i, y_j) = \frac{f(x_i, y_{j+1}) - 2f(x_i, y_j) + f(x_i, y_{j-1}))}{k^2} - \frac{h^2}{12} \frac{\partial^4 f}{\partial y^4}(x_i, \eta_j)$$

$$\eta_j \in (y_{j-1}, y_{j+1})$$

El uso de estas fórmulas permite expresar la ecuación de Poisson en los puntos (x_i, y_i) como:

$$\frac{f(x_{i+1}, y_j) - 2f(x_i, y_j) + f(x_{i-1}, y_j))}{h^2} + \frac{f(x_i, y_{j+1}) - 2f(x_i, y_j) + f(x_i, y_{j-1}))}{k^2} = g(x_i, y_j) + \frac{h^2}{12} \frac{\partial^4 f}{\partial x^4}(\zeta_i, y_j) + \frac{k^2}{12} \frac{\partial^4 f}{\partial y^4}(x_i, \eta_j)$$

$$i = 1, 2, \dots, (n-1) \quad j = 1, 2, \dots, (m-1)$$

Para condiciones de frontera:

$$f(x_0, y_j) = r(x_0, y_j) \quad j = 0, 1, \dots, m$$

$$f(x_n, y_j) = r(x_n, y_j) \quad j = 0, 1, \dots, m$$

$$f(x_i, y_0) = r(x_i, y_0) \quad i = 1, 2, \dots, n-1$$

$$f(x_i, y_m) = r(x_i, y_m) \quad i = 1, 2, \dots, n-1$$

Para un error de truncado del orden $O(h^2 + k^2)$, se tendrá:

con $\mu_{i,j}$ aproximando a $f(x_i, y_j)$

$$2 \left[\left(\frac{h}{k} \right)^2 + 1 \right] \mu_{i,j} - (\mu_{i+1,j} + \mu_{i-1,j}) - \left(\frac{h}{k} \right)^2 (\mu_{i,j+1} + \mu_{i,j-1}) = -h^2 g(x_i, y_j) \quad (A)$$

(diferencias centradas)

$$i = 1, 2, \dots, n-1 \quad j = 0, 1, \dots, m$$

$$\mu_{0,j} = r(x_0, y_j) \quad j = 0, 1, \dots, m$$

$$\mu_{n,j} = r(x_n, y_j) \quad j = 0, 1, \dots, m$$

$$\mu_{0,j} = r(x_i, y_0) \quad i = 1, 2, \dots, n-1$$

$$\mu_{0,j} = r(x_i, y_m) \quad i = 1, 2, \dots, n-1$$

A través de (A) se aproxima a $f(x, y)$ en los puntos $(x_{i-1}, y_j), (x_i, y_j), (x_{i+1}, y_j), (x_i, y_{j-1})$ y (x_i, y_{j+1}) .

Algoritmo dirich (Método de Dirichlet para la ecuación de Laplace's)

Para aproximar la solución de $u_{xx}(x, y) + u_{yy}(x, y) = 0$ más

$R = \{(x, y) : 0 \leq x \leq a, 0 \leq y \leq b\}$ con $u(x, 0) = f_1(x)$, $u(x, b) = f_2(x)$,
para $0 \leq x \leq a$, y $u(0, y) = f_3(y)$, $u(a, y) = f_4(y)$, para $0 \leq y \leq b$.

Asumiendo que $\Delta x = \Delta y = h$ y que los enteros n y m existen tal que $a = nh$ y $b = mh$.

8.3 ECUACION PARABÓLICA

Para la ecuación de **Calor** (parabólica)- unidimensional

$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2}$ su fórmula en diferencias regresivas (incondicionalmente estable) será:

$$\frac{u_{i,j} - u_{i,j-1}}{k} - \alpha^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} = 0$$

para $i = 1, 2, \dots, m-1$; y $j = 1, 2, \dots$

este método de diferencias tiene la representación matricial

$$\begin{bmatrix} (1+2\lambda) & -\lambda & 0 & & 0 \\ -\lambda & & & & \\ 0 & & & 0 & \\ & & & -\lambda & \\ 0 & & 0 & -\lambda & (1+2\lambda) \end{bmatrix} \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \vdots \\ \vdots \\ u_{m-1,j} \end{bmatrix} = \begin{bmatrix} u_{1,j-1} \\ u_{2,j-1} \\ \vdots \\ \vdots \\ u_{m-1,j-1} \end{bmatrix}$$

O sea $A\mathbf{w}^{(j)} = \mathbf{w}^{(j-1)}$ para toda $j = 1, 2, \dots$

Indicando la necesidad de resolver un sistema lineal para hallar $\mathbf{w}^{(j)}$ a partir de $\mathbf{w}^{(j-1)}$, siendo estable para cualquier λ .

Dado que $\lambda > 0$, la matriz A es definida positiva y con dominancia diagonal estricta, tridiagonal, pudiendo resolverse por algoritmos ya vistos (caso Crout).

Para evitar la falta de precisión, generada por el error de truncado, se requiere que los intervalos de tiempo sean mucho más pequeños que los de espacio ($h > k$), haciendo necesario un método que permita tomar valores similares para h y k , y estable para todo λ .

Algoritmo crnich (Crank-Nicholson método para la ecuación del calor)

Para aproximar la solución de $u_t(x,t) = c^2 u_{xx}(x,t)$ más

$R = \{(x,t) : 0 \leq x \leq a, 0 \leq t \leq b\}$ con $u(x,0) = f(x)$, para $0 \leq x \leq a$.

y $u(0,t) = c_1$, $u(a,t) = c_2$, para $0 \leq t \leq b$.

8.4 ECUACIÓN HIPERBÓLICA

Se presenta la ecuación hiperbólica a través de una situación problema

Sea la EDP $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$

con las condiciones de frontera $u(0,t) = 0 = u(1,t)$, $0 < t < 1$

y con las condiciones iniciales $u(x,0) = \sin(\pi x)$, $0 \leq x \leq 1$ y $u_t(x,0) = 0$, $0 \leq x \leq 1$

Sn exacta: $u(x, t) = \text{sen}(\pi x)\cos(2\pi t)$

La división rectangular del dominio (x, t) : $0 \leq x \leq a$, $0 \leq t \leq b$, denominando

$$u(x, t) = u_{i,j}$$

$$u(x + h, t) = u_{i+1,j}$$

$$u(x - h, t) = u_{i-1,j}$$

$$u(x, t + k) = u_{i,j+1}$$

$$u(x, t - k) = u_{i,j-1}$$

la ecuación en diferencias será

$$\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = c^2 \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{h^2}$$

Haciendo $\alpha = ck/h$

$$u_{i,j+1} - 2u_{i,j} + u_{i,j-1} = \alpha^2 (u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1})$$

Colocando en términos de $u_{i,j+1}$ y reordenando, se tiene

$$u_{i,j+1} = (2 - 2\alpha^2)u_{i,j} + \alpha^2(u_{i+1,j} + u_{i-1,j}) - u_{i,j-1}$$

La ecuación (arriba) es aplicable para $i = 2, 3, \dots, n - 1$ y $j = 2, 3, \dots, m - 1$, con el paso $(j + 1)$ de tiempo requiriendo de los j -ésimo y los $(j - 1)$ ésimo pasos; los valores del $(j - 1)$ ésimo paso vienen dados por las condiciones iniciales $u_{i,1} = f(x_i)$ pero, los valores del j ésimo paso no

se suelen proporcionar, por lo que se usa $g(x)$ para conseguir los valores de este paso.

Para asegurar la estabilidad de este método es necesario que:

$$\alpha = ck/h \leq 1.$$

Algoritmo finedif (diferencia finita para la ecuación de onda)

Para aproximar la solución de $u_{tt}(x, t) = c^2 u_{xx}(x, t)$ más

$R = \{(x, t) : 0 \leq x \leq a, 0 \leq t \leq b\}$ con $u(0, t) = 0$, $u(a, t) = 0$,

para $0 \leq t \leq b$, y $u(x, 0) = f(x)$, $u_t(x, 0) = g(x)$, para $0 \leq x \leq a$.

ANEXO GNU OCTAVE

Octave es un lenguaje de ordenador de alto nivel, orientado al cálculo numérico. Y su vez es un programa que permite interpretar este lenguaje de forma interactiva, mediante órdenes o comandos. Estas órdenes pueden ser introducidas a través de un entorno en forma de terminal o consola de texto. Por supuesto, *Octave* incluye, además, la posibilidad de ser utilizado de forma no interactiva, interpretando las órdenes contenidas en un fichero.

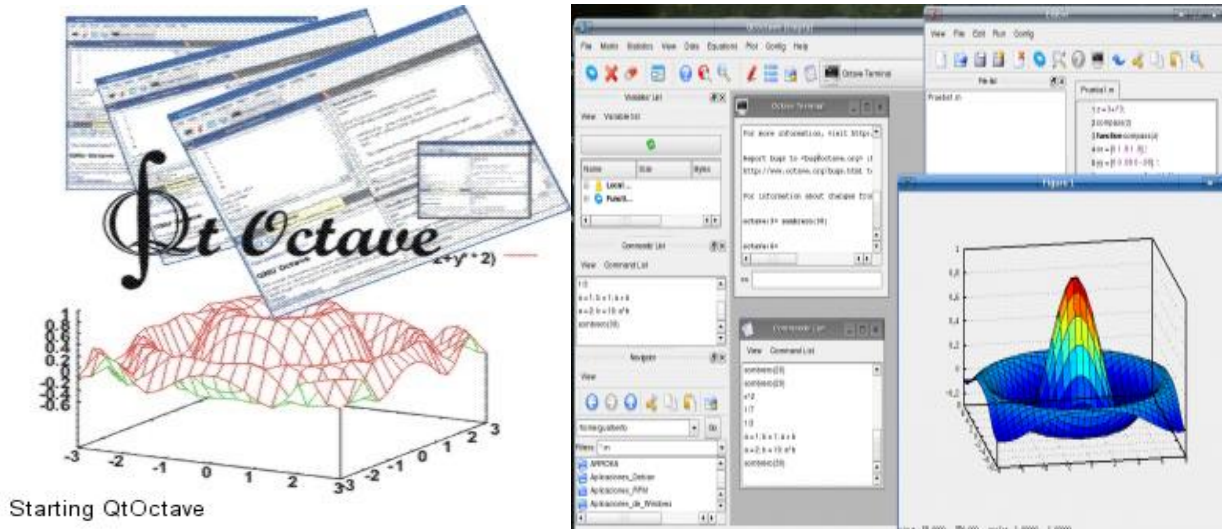
Estas características se pueden resumir diciendo que *Octave* es un lenguaje interpretado orientado al cálculo numérico matricial. Por supuesto, existe una gran variedad de lenguajes interpretados orientados a las matemáticas, algunos de ellos con licencia libre, como Máxima, Axiom, Scilab, EULER, Sage o R y otros con licencia privativa, como Matlab, Maple, Mathematica o S.

Los lenguajes interpretados, entre los cuales destaca la menor velocidad de proceso que lenguajes compilados como C, C++, Fortran, etc. Sin embargo, *Octave* y el resto de los lenguajes interpretados presentan una serie de ventajas muy significativa: la razón entre el esfuerzo empleado en el desarrollo del programa y el resultado final es muy pequeña, es decir, permiten obtener resultados razonables con un esfuerzo no demasiado importante. Este hecho, que está relacionado con la eliminación de la segunda etapa dentro del ciclo clásico de desarrollo de programas (escritura-compilado-depurado), es vital en:

- a) Desde sus orígenes, *Octave* es software libre, gracias a lo cual, su comunidad de desarrolladores y usuarios ha ido creciendo hasta llegar a ser muy significativa. Es un proyecto en colaboración cuyos miembros nos prestarán ayuda cuando la solicitemos, a través de sus listas de correo. Éstas son públicas y una gran fuente de información.
- b) Su lenguaje de programación es altamente compatible con el de Matlab, el conocido entorno de cálculo numérico (con licencia privativa) desarrollado por The MathWorks, Inc. De hecho, la mayor parte de la materia desarrollada en las siguientes páginas es, de hecho, un análisis del amplio lenguaje que es común a Matlab y *Octave* y al que, indistintamente, denominaremos “lenguaje *Octave*” o “lenguaje Matlab”.
- c) Está disponible en numerosos plataformas, entre ellas sistemas Unix (para los que fue originalmente desarrollado), Windows y MacOSX. Su código fuente contiene una potente librería de cálculo matricial para C++. De hecho, ha sido diseñado de forma que sea extensible dinámicamente, a través de nuevos procedimientos escritos en C++. Estas extensiones son programas “especiales” que tienen la extensión .oct y, como programas compilados, cuentan con un rendimiento mucho mayor que las funciones escritas en el lenguaje interpretado de *Octave*.
- d) interfaces gráficas de usuario que, en la actualidad, ofrece prestaciones más interesantes para un usuario medio: *qtOctave*. Este programa recibe su nombre de las bibliotecas QT. desarrolladas con la finalidad de facilitar la creación de interfaces gráficas.

Interfaz qtOctave

La interfaz gráfica permite, entre otras cosas, trabajar con el interprete de órdenes de Octave, ver el listado de variables, editar un archivo .m, y ver el histórico de órdenes.



qtOctave está disponible:

A través de su página web, <http://qt octave.wordpress.com/>. Esta página es el centro de comunicación con el autor y la comunidad de usuarios de qtOctave.

```
function [c,yc,err,P]=bisect(a,b,delta)
```



METODO DE LA BISECCIÓN

```
# Ejemplo de carga
# [c,yc,err,P]=bisect(0,2,0.001)
# función 'x*sin(x)-1'
f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
err=0;
P=[a b err];
ya=feval(f,a);
yb=feval(f,b);
if ya*yb>0,break,end
    max1=1+round((log(b-a)-log(delta))/log(2));
    for k=1:max1,
        c=(a+b)/2;
        yc=feval(f,c);
        if yc==0,a=c;
            b=c;
        elseif yb*yc>0,
            b=c;
            yb=yc;
        else
            a=c;
            ya=yc;
        end
        err=abs(b-a)/2;
        P=[P;a b err];
        if b-a<delta,break,end
            end
            c=(a+b)/2;
            yc=feval(f,c);
            err=abs(b-a)/2;
            disp(c);
            disp(yc);
            disp(err);
            disp('X0 X1 ERROR');
            disp(P);
        end
```

Dada la función $f(x) = x \sin x - 1$, hallar una raíz en el intervalo $[0,2]$ por el método de la bisección y una tolerancia de 10^{-3} . Igual para $x^3 + 4x^2 - 10$ en $[1,2]$ y tolerancia de 10^{-4}


```
octave:4> [c,yc,err,P]=bisect(0,2,0.001)
ingresef(x) entre comillas en términos de x 'x*sin(x)-1'
'x*sin(x)-1'
1.1138
-5.3832e-004
4.8828e-004
X0 X1 ERROR
0.00000 2.00000 0.00000
1.00000 2.00000 0.50000
1.00000 1.50000 0.25000
1.00000 1.25000 0.12500
1.00000 1.12500 0.06250
1.06250 1.12500 0.03125
1.09375 1.12500 0.01563
1.10938 1.12500 0.00781
1.10938 1.11719 0.00391
```

```

1.11328 1.11719 0.00195
1.11328 1.11523 0.00098
1.11328 1.11426 0.00049
c = 1.1138
yc = -5.3832e-004
err = 4.8828e-004
P =
0.00000 2.00000 0.00000
1.00000 2.00000 0.50000
1.00000 1.50000 0.25000
1.00000 1.25000 0.12500
1.00000 1.12500 0.06250
1.06250 1.12500 0.03125
1.09375 1.12500 0.01563
1.10938 1.12500 0.00781
1.10938 1.11719 0.00391
1.11328 1.11719 0.00195
1.11328 1.11523 0.00098
1.11328 1.11426 0.00049

```

```

function fixed_point(p0,N)  METODO DEL PUNTO FIJO
%Fixed_Point(p0, N) aproxima la raíz de la ecuación f(x) = 0
%reescrita en la forma x = g(x), partiendo con una aproximación inicial p0.
%La técnica iterativa se implementa N veces.
%El usuario tiene que entrar g(x)abajo

g=input('ingrese f(x) entre comillas en términos de x');
g=inline(g,'x');
i = 1;
p(1) = p0;
tol = 1e-05;
while i <= N
    p(i+1) = g(p(i));
    if abs(p(i+1)-p(i)) < tol %stopping criterion
        disp('el procedimiento fue exitoso luego de k iteraciones')
        k = i
        disp('la raíz de la ecuación es')
        p(i+1)
        return
    end
    i = i+1;
end

if abs(p(i)-p(i-1)) > tol | i > N
    disp('el procedimiento falló')

```



```

disp('Condición |p(i+1)-p(i)| < tol no se satisfizo')
tol
disp('Por favor, examine la secuencia de iteraciones')
p = p'
disp('Si observa convergencia, incremente el número máximo de
iteraciones')
disp('Si hay divergencia, intente otro p0 o reescriba g(x)')
disp('si |g'(x)|< 1 cerca de la raíz')
end

```

Dada la función $f(x)=x-x^3+4x^2-10/3x^2+8x$, encontrar la raíz por el método de punto fijo

Para la función $y = x - (x.^3 + 4*x.^2 - 10)/(3*x.^2 + 8*x);$

```

octave:7> fixed_point(1.5, 10)
ingresef(x) entre comillas en términos de x 'x- x^3-4*x^2+10'
'x- x^3-4*x^2+10'
el procedimiento falló
Condición |p(i+1)-p(i)| < tol no se satisfizo
tol = 1.0000e-005
Por favor, examine la secuencia de iteraciones
p =
1.5000e+000
-8.7500e-001
6.7324e+000
-4.6972e+002
1.0275e+008
-1.0849e+024
1.2771e+072
-2.0827e+216
NaN
NaN
NaN
Si observa convergencia, incremente el número máximo de iteraciones
Si hay divergencia, intente otro p0 o reescriba g(x)
si |g'(x)|< 1 cerca de la raíz


```

Tomando su equivalente
 $y = \sqrt{10./x - 4*x};$

```

octave:8> fixed_point(1.5, 10)
ingresef(x) entre comillas en términos de x 'sqrt(10./x - 4*x)'
'sqrt(10./x - 4*x)'
el procedimiento falló
Condición |p(i+1)-p(i)| < tol no se satisfizo
tol = 1.0000e-005
Por favor, examine la secuencia de iteraciones
p =
1.50000 - 0.00000i
0.81650 - 0.00000i
2.99691 - 0.00000i
0.00000 - 2.94124i
2.75362 + 2.75362i
1.81499 - 3.53453i
2.38427 + 3.43439i
2.18277 - 3.59688i
2.29700 + 3.57410i
2.25651 - 3.60656i
2.27918 + 3.60194i

```


```
function [p0,y0,err,P] = newton(p0,delta,max1)  METODO DE NEWTON
# Ejemplo carga [p0,y0,err,P] = newton(0.25*pi,0.01,12)
# función 'cos(x)-x'
# derivada '-sin(x)-1'
f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
epsilon = 1.0842e-19;
df = input('ingrese la derivada de f(x) entre comillas, en términos de x');
df = inline(df,'x');
P(1) = p0;
P(2) = 0;
P(3) = 0;
P(4) = 0;
P(5) = 0;
y0 = feval(f,p0);
for k=1:max1,
df0 = feval(df,p0);
if df0 == 0,
dp = 0;
else
dp = y0/df0;
end
p1 = p0 - dp;
y1 = feval(f,p1);
err = abs(dp);
relerr = err/(abs(p1)+eps);
p0 = p1;
y0 = y1;
P = [P;p1 y0 df0 err relerr];
if (err<delta)|(relerr<delta)|(abs(y1)<epsilon), break, end
end
disp(' Xi          F(Xi)          F(Xi)          ErrorAbs          ErrorRel ');
disp(P);
disp('Error Absoluto: '),disp(err);
disp(' Error Relativo: '), disp(relerr);
end
```

Dada la función $f(x)=x-x^3+4x^2-10$, hallar una raíz por el método de Newton con aprox. inicial 1.5, tolerancia para la raíz de 10^{-4} y 10^{-5} para los valores de f , con 10 iteraciones.

Sea la función x^3+4x^2-10


Su derivada es $3x^2+8x$

```
octave:9> newton(1.5,0.0001,10)
ingresef(x) entre comillas en términos de x 'x^3+4*x^2-10'
'x^3+4*x^2-10'
ingrese la derivada de f(x) entre comillas, en términos de x '3*x^2+8*x'
'3*x^2+8*x'
Xi F(Xi) F(Xi) ErrorAbs ErrorRel
1.50000 0.00000 0.00000 0.00000 0.00000
1.37333 0.13435 18.75000 0.12667 0.09223
1.36526 0.00053 16.64480 0.00807 0.00591
1.36523 0.00000 16.51392 0.00003 0.00002
ErrorAbsoluto:
3.2001e-005
Error Relativo:
2.3440e-005
ans = 1.3652
```

```
function [p1,y1,err,P] = secant(p0,p1,delta,max1)  METODO DE LA SECANTE
# Ejemplo de carga [p1,y1,err,P] = secant(0.5,pi/4,0.01,40)
# función 'cos(x)-x'
f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
epsilon = 1.0842e-19;
P(1) = p0;
P(2) = p1;
P(3) = 0;
P(4) = 0;
P(5) = 0;
P(6) = 0;
y0 = feval(f,p0);
y1 = feval(f,p1);
for k=1:max1,
df = (y1-y0)/(p1-p0);
if df == 0,
dp = 0;
else
dp = y1/df;
end
p2 = p1 - dp;
y2 = feval(f,p2);
err = abs(dp);
relerr = err/(abs(p2)+eps);
p0 = p1;
y0 = y1;
p1 = p2;
y1 = y2;
P = [P;p0 p2 p1 y1 err relerr];
if (err<delta)|(relerr<delta)|(abs(y2)<epsilon), break, end
end
disp('P0    P2    Xi    F(Xi)    ErrAbs    ErrorRelativo');
disp(P);
disp('Error Absoluto:'),disp(err),
disp('Error Relativo:'),disp(relerr);
end
```

Para la función $\cos x - x$, hallar una raíz con aprox. iniciales 0.5 y $\pi/4$, tolerancia para $p_i=0.01$, para f de 0 01 y 10 iteraciones, por el método de la secante

```
octave:10> [p1,y1,err,P]=secant(0.5,pi/4,0.01,10)
ingresef(x) entre comillas en términos de x 'cos(x)-x'
'cos(x)-x'
P0 P2 Xi F(Xi) ErrAbs ErrorRelativo
0.50000 0.78540 0.00000 0.00000 0.00000 0.00000
0.78540 0.73638 0.73638 0.00452 0.04901 0.06656
0.73638 0.73906 0.73906 0.00005 0.00267 0.00362
ErrorAbsoluto:
0.0026740
ErrorRelativo:
0.0036181
p1 = 0.73906
y1 = 4.5177e-005
err = 0.0026740
P =
    0.50000    0.78540    0.00000    0.00000    0.00000    0.00000
    0.78540    0.73638    0.73638    0.00452    0.04901    0.06656
    0.73638    0.73906    0.73906    0.00005    0.00267    0.00362
```

`function [c,yc,err,P] = regula(a,b,delta,maxl)`  **METODO DE LA POSICIÓN FALSA**

```
# Ejemplo carga [c,yc,err,P] = regula(1.38,1.0,0.001,10)
# función 'x^3+4*x^2-10'
f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
epsilon = 1.0842e-19;
P = [a b 0 0 0];
ya = feval(f,a);
yb = feval(f,b);
if ya*yb > 0, break, end
for k=1:maxl,
dx = yb*(b - a)/(yb - ya);
c = b - dx;
ac = c - a;
yc = feval(f,c);
end
if yc == 0,
break;
elseif yb*yc > 0,
b = c;
yb = yc;
else
a = c;
ya = yc;
end
dx = min(abs(dx),ac);
err = abs(dx);
if abs(dx) < delta, break, end
if abs(yc) < epsilon, break, end
end
err = abs(dx);
P = [a b c yc err];
disp(' X0 X1 Raiz FunctValor Error')
disp(P)
end
```

Utilizar la falsa posición para a) x^3+4x^2-10 en $[1.38 \ 1]$, tolerancias de 10^{-3} y 10 iteraciones

```
octave:12> [c,yc,err,P] = regula(1.38,1.0,0.001,10)
ingresef(x) entre comillas en términos de x 'x^3+4*x^2-10'
'x^3+4*x^2-10'
X0X1 Raiz FunctValor Error
1.380000 1.362203 1.362203 -0.049906 0.017797
c = 1.3622
yc = -0.049906
err = 0.017797
P =
    1.380000    1.362203    1.362203   -0.049906    0.017797
```

```
function [p,Q]=steff(p0,delta,epsilon,maxl)
```



METODO DE STEFFENSEN

```
%Ingreso - f es la function ingresada como string 'f'
%          - df es la derivada f ingresada como string 'df'
%          - p0 es la aproximación inicial a un cero de f
%          - delta es la tolerancia para p0
%          - epsilon es la tolerancia para los valores de la función y
%          - maxl es el número máximo de iteraciones
%Salida - p es la aproximación de Steffensen al cero
%         - Q matriz conteniendo la secuencia de Steffensen
%Inicializar la matriz R
f = input('ingrese f(x) entre comillas en términos de x');
f = inline(f,'x');
df = input('ingrese la derivada de f(x) entre comillas, en términos de x');
df = inline(df,'x');
R = zeros(maxl,3);
R(1,1)=p0;
for k=1:maxl
    for j=2:3

        %Cálculo del Denominador en el método de Newton-Raphson
        nrdenom=feval(df,R(k,j-1));

        %Condicional calcula las aproximaciones de Newton-Raphson
        if nrdenom==0
            'división cero en el método de Newton-Raphson '
            break
        else
            R(k,j)=R(k,j-1)-feval(f,R(k,j-1))/nrdenom;
        end

        %Cálculo de Denominador en el proceso de aceleración de Aitkens
        aadenom=R(k,3)-2*R(k,2)+R(k,1);


        % condicional en las aproximaciones de Aitkens
        if aadenom==0
            'división por cero en aceleración de Aitkens '
            break
        else
            R(k+1,1)=R(k,1)-(R(k,2)-R(k,1))^2/aadenom;
        end
    end

    %Corte y salida del programa si ocurre división por cero
    if (nrdenom==0) | (aadenom==0)
        break
    end


    %criterio de parada; se determinan p y matriz Q
    err=abs(R(k,1)-R(k+1,1));
    relerr=err/(abs(R(k+1,1))+delta);
    y=feval(f,R(k+1,1));
    if (err<delta) | (relerr<delta) | (y<epsilon)
        p=R(k+1,1);
        Q=R(1:k+1,:);
        break
    end
end
```

Aplicar el método de Steffensen. Recordando el problema de $f(x) = \cos x - x$ con newton.

```
octave:14> [p,Q]=steff(0.25*pi,0.01,0.01,4)
ingrese f(x) entre comillas en términos de x 'cos(x)-x'
'cos(x)-x'
ingrese la derivada de f(x) entre comillas, en términos de x '(x)-sin(x)-1'
'(x)-sin(x)-1'
p = 0.73819
Q =
    0.78540  0.70046  0.76834
    0.73819  0.00000  0.00000
```

`function [y,b] = horner(a,z)`  [Método de Horner](#)

```
%HORNER Horner algorithm
% Y=HORNER(A,Z) calcula
% Y = A(1)*Z^N + A(2)*Z^(N-1) + ... + A(N)*Z + A(N+1)
% usando el algoritmo de division sintética de Horner.
n = length(a)-1;
b = zeros(n+1,1);
b(1) = a(1);
for j=2:n+1
b(j) = a(j)+b(j-1)*z;
end
y = b(n+1);
b = b(1:end-1);
return
```

`function [roots ,iter]= newtonhorner(a,x0 ,tol ,nmax)`  [Método de Horner](#)

```
%NEWTONHORNER Newton -Horner
% [roots ,ITER ]= NEWTONHORNER(A,X0) calcula las raíces del
% polinomio
% P(X) = A(1)*X^N + A(2)*X^(N-1) + ... + A(N)*X +
% A(N+1)
% usando el método de Newton -Horner partiendo de los datos iniciales X0. El
método para dada raíz
% luego de 100 iteraciones o luego que el valor absoluto de la diferencia
entre dos iteraciones consecutivas es menor que
% 1.e -04.
% [roots ,ITER ]= NEWTONHORNER(A,X0 ,TOL ,NMAX) permite definir la
tolerancia en el criterio de parada y el número máximo de iteraciones.
if nargin == 2
tol = 0.0001; nmax = 100;
elseif nargin == 3
nmax = 100;
end
n=length(a)-1; raiz = zeros(n,1); iter = zeros(n,1);
```

```

for k = 1:n
% iteraciones de
niter = 0; x = x0; diff = tol + 1;
while niter <= nmax & diff >= tol
[pz ,b] = horner(a,x); [dpz ,b] = horner(b,x);
xnew = x - pz/dpz; diff = abs(xnew -x);
niter = niter + 1; x = xnew;
end
if niter >= nmax
fprintf(' falla de convregencia con máximo ' ,...
'número de iteraciones\n');
end
% Deflación
[pz ,a] = horner(a,x); raiz(k) = x; iter(k) = niter;
end
return

```

```

octave:36>[roots ,iter]= newtonhorner([-3 1 0],2,0.001,50)
roots =
    3.3334e-001
   -2.8529e-006
iter =
    6
    2

```

`function [p2,y2,err,P]=muller(p0,p1,p2,delta,max1)`  [Método de Muller](#)

```

# Ejemplo carga [p2,y2,err,P]=muller(0.5,pi/3.5,pi/4,0.01,12)
# función 'cos(x)-x'
f=input('ingrese f(x):');
f=inline(f,'x');
epsilon = 1.0842e-19;
P(1) = p0;
P(2) = p1;
P(3) = p2;
P(4) = 0;
P(5) = 0;
y0 = feval(f,p0);
y1 = feval(f,p1);
y2 = feval(f,p2);
for k=1:max1;
h0 = p0 - p2;
h1 = p1 - p2;
c = y2;
e0 = y0 - c;
e1 = y1 - c;
det1 = h0*h1*(h0-h1);
a = (e0*h1 - h0*e1)/det1;
b = (h0^2*e1 - h1^2*e0)/det1;
if b^2 > 4*a*c,
disc = sqrt(b^2 - 4*a*c);
else

```

```

disc = 0;
end
if b < 0
disc = - disc;
end
z = - 2*c/(b + disc);
p3 = p2 + z;
if abs(p3-p1) < abs(p3-p0)
u = p1;
p1 = p0;
p0 = u;
v = y1;
end
end

```

Emplear el método de Muller para $\cos x - x$ con $p_0=0.5$ $p_1=\pi/3.5$, $p_2= \pi/3.5$

```

octave:17> [p2,y2,err,P]=muller(0.5,pi/3.5,pi/4,0.01,0.001,12)
ingresef(x): 'cos(x)-x '
'cos(x)-x '
p2 = 0.78540
y2 = -0.078291
err = [] (0x0)
P =
0.50000 0.89760 0.78540 0.00000 0.00000

```

`function [p,y,err]=muller2(p0,p1,p2,delta,epsilon,max1)`  [Método de Muller](#)

```

# Ejemplo carga [p2,y2,err,P]=muller2(0.5,pi/3.5,pi/4,0.01,0.001,12)
# funcion 'cos(x)-x'
%Ingresos - f es la function como string 'f'
% - p0, p1, p2 son las aproximaciones iniciales
% - delta tolerancia para p0, p1, p2
% - epsilon la tolerancia para los valores de la función y
% - max1 número máximo de iteraciones
%salida- p aproximación de Muller al cero de f
% - y valor de la function y = f(p)
% - err error en la aproximación de p.
%Inicializar las matrices P e Y
f=input('ingrese f(x):');
f=inline(f,'x');
P=[p0 p1 p2];
Y=feval(f,P);
%Calcula a y b de formula (15)
for k=1:max1
    h0=P(1)-P(3);h1=P(2)-P(3);e0=Y(1)-Y(3);e1=Y(2)-Y(3);c=Y(3);
    denom=h1*h0^2-h0*h1^2;
    a=(e0*h1-e1*h0)/denom;
    b=(e1*h0^2-e0*h1^2)/denom;

    %Suprime las raíces complejas
    if b^2-4*a*c > 0
        disc=sqrt(b^2-4*a*c);
    else
        disc=0;
    end
end

```



```


end
%halla la menor raíz de (17)
if b < 0
    disc=-disc;
end

z=-2*c/(b+disc);
p=P(3)+z;
%clasifica las entradas de p para hallar las dos más cercanas a p
if abs(p-P(2))<abs(p-P(1))
    Q=[P(2) P(1) P(3)];
    P=Q;
    Y=feval(f,P);
end
if abs(p-P(3))<abs(p-P(2))
    R=[P(1) P(3) P(2)];
    P=R;
    Y=feval(f,P);
end
%Reemplaza las entradas de P más alejadas de p con p
P(3)=p;
Y(3) = feval(f,P(3));
y=Y(3);

%Determina criterio de parada
err=abs(z);
relerr=err/(abs(p)+delta);
if (err<delta) | (relerr<delta) | (abs(y)<epsilon)
    break
end
end

[p2,y2,err,P]=muller2(0.5,pi/3.5,pi/4,0.01,0.001,12)
ingresef(x): 'cos(x)-x'
'cos(x)-x'
p2 = 0.73897
y2 = 1.9532e-004
err = 0.046430

```

function [C,L]=lagrange(X,Y)  **POLINOMIO DE LAGRANGE**

```

# Ejemplo carga [C,L]=lagrange([1.0 1.3 1.6 1.9 2.2],[0.7651977 0.6200860
0.4554022 0.2818186 0.1103623])
%Ingreso - X es el vector de abcisas
% - Y es el vector de ordenadas
%salida - C es una matriz que contiene los coeficientes del
% polinomio interpolatorio de Lagrange
% - L es una matriz que contiene los coeficientes polinomiales de
% Lagrange

```

```
w=length(X);
n=w-1;
L=zeros(w,w);

%Forma los coeficientes polinomiales de Lagrange

for k=1:n+1
    V=1;
    for j=1:n+1
        if k~=j
            V=conv(V,poly(X(j)))/(X(k)-X(j));
        end
    end
    L(k,:)=V;
end
%Determina los coeficientes del polinomio interpolatorio de Lagrange
C=Y*L;
```

Dados los siguientes valores x-y:

```
x=[1.0 1.3 1.6 1.9 2.2];
```

```
y=[0.7651977 0.6200860 0.4554022 0.2818186 0.1103623];
```

encontrar el polinomio interpolante de Lagrange


```
octave:19> [C,L]=lagrange([1.0 1.3 1.6 1.9 2.2],[0.7651977 0.6200860
0.4554022
0.2818186 0.1103623])
```

C =

```
0.0018251 0.0552928 -0.3430466 0.0733913 0.9777351
```

L =

```
5.1440 -36.0082 93.3642 -106.2243 44.7243
-20.5761 137.8601 -338.2716 358.6008 -137.6132
30.8642 -197.5309 460.1852 -461.2346 167.7160
-20.5761 125.5144 -279.0123 268.2305 -94.1564
5.1440 -29.8354 63.7346 -59.3724 20.3292
```

`function hp=hermite(x,y,dy)`  **POLINOMIO DE HERMITE**

```
% busca los polinomios interpolantes de Hermite para múltiples subintervalos
%Ingreso : [x,y],dy - puntos y sus derivadas
%salida: H = coeficientes de polinomio interpolante cúbico de Hermite
%
%Polinomio interpolante de Hermite. Considere el problema de hallar
%el polinomio de interpolación para los N + 1 = 4 datos
%{(0, 0), (1, 1), (2, 4), (3, 5)}
%>>x = [0 1 2 3]; y = [0 1 4 5]; dy = [2 0 0 2]; xi = [0:0.01:3];
%>>H = hermite(x,y,dy);
%yi = ppval(mkpp(x,H), xi);
```

```
for n = 1:length(x)-1
H(n,:) = hermit(0,y(n),dy(n),x(n+1)-x(n),y(n+1),dy(n+1));
end
```

```
function H = hermit(x0,y0,dy0,x1,y1,dy1)
% para obtener los coef. de Hermite para un conjunto de múltiples
subintervalos.
A = [x0^3 x0^2 x0 1; x1^3 x1^2 x1 1;
3*x0^2 2*x0 1 0; 3*x1^2 2*x1 1 0];
b = [y0 y1 dy0 dy1]'; %
H = (A\b)';
```

```
octave:20> H = hermite([0 1 2 3],[0 1 4 5],[2 0 0 2])
```

```
H =
```

```
    0.00000   -1.00000   2.00000   0.00000
   -6.00000    9.00000   0.00000   1.00000
    0.00000    1.00000   0.00000   4.00000
```

```
octave:21> yi = ppval(mkpp([0 1 2 3],H), [0:0.1:3])
```

```
yi =
```

```
Columns 1 through 7:
```

```
0.00000 0.19000 0.36000 0.51000 0.64000 0.75000 0.84000
```

```
Columns 8 through 14:
```

```
0.91000 0.96000 0.99000 1.00000 1.08400 1.31200 1.64800
```

```
Columns 15 through 21:
```


```
2.05600 2.50000 2.94400 3.35200 3.68800 3.91600 4.00000
```

```
Columns 22 through 28:
```

```
4.01000 4.04000 4.09000 4.16000 4.25000 4.36000 4.49000
```

```
Columns 29 through 31:
```

```
4.64000 4.81000 5.00000
```

```
function p = neville1(x, f, xdach)  MÉTODO DE NEVILLE
```

```
% Algoritmo Aitken- Neville-
```

```
%
```

```
% usar: p = neville(x, f, xdach)
```

```
%
```

```
% ingreso: vector x - ,x-coordenadas
```

```
%          vector f - ,y-coordenadas
```

```
%          escalar xdach - punto a evaluar
```

```
%
```

```
% salida: escalar p - valor interpolado en xdach
```

```
n = length(x);
```

```

for k = n-1:-1:1
    f(1:k) = f(2:k+1) + ...
                ( xdach - x(n-k+1:n) ) ./ ...
                ( x(n-k+1:n) - x(1:k) ) .* ...
                ( f(2:k+1) - f(1:k) );
end
p = f(1);


```

Dados los siguientes valores x-y, generar el valor de x=1.4 por Neville
 $x=[1.0 \ 1.3 \ 1.6 \ 1.9 \ 2.2]$;
 $y=[0.7651977 \ 0.6200860 \ 0.4554022 \ 0.2818186 \ 0.1103623]$;

```

octave:23> p=neville1([1.0 1.3 1.6 1.9 2.2],[0.7651977 0.6200860 0.4554022
0.2818186 0.1103623],1.4)
p = 0.56685

```

`function y = neville2(xx,n,x,Q)`  [MÉTODO DE NEVILLE](#)

```

% Algoritmo de Neville como una función (guardar como "nev.m")
%
% ingresos:
%   n = orden de interpolación (n+1 = # de puntos)
%   x(1),...,x(n+1)   x coordenadas
%   Q(1),...,Q(n+1)   y coordenadas
%   xx=punto de evaluación para polinomio interpolante p
%
% salida:  p(xx)
% x=1940:10:1990;
% Q=[132165 151326 179323 203302 226542 249633];
% neville2(2000,5,x,Q)

%rpta =      251654
for i = n:-1:1
    for j = 1:i
        Q(j) = (xx-x(j))*Q(j+1) - (xx-x(j+n+1-i))*Q(j);
        Q(j) = Q(j)/(x(j+n+1-i)-x(j));
    end
end
y = Q(1);

```


Recordando el ejercicio de neville1.

Con el algoritmo de neville2, para el mismo punto, grado 4

```

octave:23> p=neville1([1.0 1.3 1.6 1.9 2.2],[0.7651977 0.6200860 0.4554022
0.2818186 0.1103623],1.4)
p = 0.56685
octave:24> y=neville2(1.4,4,[1.0 1.3 1.6 1.9 2.2],[0.7651977 0.6200860
0.4554022 0.2818186 0.1103623])
y = 0.56685

```

```
function nf = divdiff1(xi,fi)  DIFERENCIAS DIVIDIDAS
%DIVDIFF      calcula los the coeficientes para la diferencia hacia
%              adelante de Newton,del polinomio interpolante asociado
%              con un dado conjunto de puntos interpolantes
%
%      secuencias de llamado:
%              nf = divdiff1 ( xi, fi )
%              divdiff1 ( xi, fi )
%
%      ingresos:
%              xi      vector conteniendo los puntos interpolantes
%              fi      vector conteniendo los valores de la función
%                      la entrada I en este vector es el valor de la función
%                      asociado con la entrada i del vector'xi'
%
%      salida:
%              nf      vector conteniendo coeficientes de la diferencia hacia
%                      adelante de Newton,del polinomio interpolante asociado
%                      con un dado conjunto de puntos interpolantes
%
%      NOTA:
%              para evaluar la forma de Newton , aplicar rutina NF_EVAL
%ejemplo:> fi=[0.8;1.6;2.25;2.8;3.35];
% xi=[0.5;0.8;1.0;1.4;1.75];
%divdiff1 ( xi, fi );ans =      0.8000      2.6667      1.1667      -4.7685
%6.6669;

n = length ( xi );
m = length ( fi );

if ( n ~= m )
    disp ( 'error divdiff : número de ordendas y número de valores de la
función deben ser iguales' )
    return
end

nf = fi;
for j = 2:n
    for i = n:-1:j
        nf(i) = ( nf(i) - nf(i-1) ) / ( xi(i) - xi(i-j+1) );
    end
end

Dados los siguientes valores x-y.
x=[1.0 1.3 1.6 1.9 2.2];
y=[0.7651977 0.6200860 0.4554022 0.2818186 0.1103623];
generar las diferencias divididas y el polinomio de Newton

octave:29> xi=[1.0 1.3 1.6 1.9 2.2];
octave:30> fi=[0.7651977 0.6200860 0.4554022 0.2818186 0.1103623];
octave:31> divdiff1 ( xi, fi )
ans =
    0.7651977 -0.4837057 -0.1087339 0.0658784 0.0018251
```

```
function [C,D] = newpoly(X,Y)  DIFERENCIAS DIVIDIDAS

%Ingreso    - X  vector de abcisas
%           - Y  vector de ordenadas
%salida     - C es un vector que contiene los coeficientes
%           del polinomio interpolante de Newton
%           - D es la tabla de diferencias divididas

%

n=length(X);
D=zeros(n,n);
D(:,1)=Y';

%

for j=2:n
    for k=j:n
        D(k,j)=(D(k,j-1)-D(k-1,j-1))/(X(k)-X(k-j+1));
    end
end


%Determina los coeficientes del polinomio interpolante de Newton

C=D(n,n);

for k=(n-1):-1:1
    C=conv(C,poly(X(k)));
    m=length(C);
    C(m)=C(m)+D(k,k);
end

Dados los siguientes valores x-y.
x=[1.0 1.3 1.6 1.9 2.2];
y=[0.7651977 0.6200860 0.4554022 0.2818186 0.1103623];
generar los coeficientes del polinomio interpolante de Newton
y la tabla de diferencias divididas

octave:29> xi=[1.0 1.3 1.6 1.9 2.2];
octave:30> fi=[0.7651977 0.6200860 0.4554022 0.2818186 0.1103623];
octave:32> [C,D] = newpoly(xi,fi)
C =
    0.0018251    0.0552928   -0.3430466    0.0733913    0.9777351
D =
    0.76520    0.00000    0.00000    0.00000    0.00000
    0.62009   -0.48371    0.00000    0.00000    0.00000
    0.45540   -0.54895   -0.10873    0.00000    0.00000
    0.28182   -0.57861   -0.04944    0.06588    0.00000
    0.11036   -0.57152    0.01182    0.06807    0.00183
```

```
function ls = linear_spline ( xi, fi )  SPLINE

%LINEAR_SPLINE    calcula el interpolante lineal a trozos asociados a
```

```
%
%           un conjunto dado de puntos de interpolación y los valores
%           de la función
%
%   forma de la llamada:
%       ls = linear_spline ( xi, fi )
%       linear_spline ( xi, fi )
%
%   Entrada:
%       xi       vector que contiene los puntos de interpolación
%                (orden ascendente)
%       fi       vector que contiene los valores de la función
%
%
%   Salida:
%       ls       matriz de tres columnas
%                - primer columna: puntos de interpolación
%                - segunda columna: valores de la función
%                - tercer columna: piezas lineales
%
%   Nota:
%       para evaluar el interpolante lineal a trozos aplicar el
%       algoritmo SPLINE_EVAL para la salida de esta rutina
%
```

```
n = length ( xi );
m = length ( fi );

if ( n ~= m )
    disp ( 'Número de ordenadas y valores de la función, debe ser igual' )
    return
end

b = zeros (n,1);
b(1:n-1) = ( fi(2:n) - fi(1:n-1) ) ./ ( xi(2:n) - xi(1:n-1) );

if ( nargin == 0 )
    disp ( [ xi' fi' b ] )
else
    ls = [ xi' fi' b ];
end
```

```
Dados los valores de  $x-f(x)$ , encontrar el trazador lineal:
xi=[0.1 0.2 0.3 0.4]; fi=[-0.62049958 -0.2839668 0.00660095 0.24842440];
octave:7> xi=[0.1 0.2 0.3 0.4];
octave:8> fi=[-0.62049958 -0.2839668 0.00660095 0.24842440];
octave:9> linear_spline ( xi, fi )
0.10000 -0.62050 3.36533
0.20000 -0.28397 2.90568
0.30000 0.00660 2.41823
0.40000 0.24842 0.00000
```

La tercera columna da las pendientes de los trozos lineales
 Los valores del pol se obtienen con "**spline_eval**". Ver a continuación

function y = spline_eval (sp, x)  [**ADAPTADOR CÚBICO**](#)

```
%SPLINE_EVAL           evalúa una dada función spline function en un dado
```


```
%
% conjunto de valores de la variable independiente
%
% secuencia de llamado:
%     y = spline_eval ( sp, x )
%     spline_eval ( sp, x )
%
% ingresos:
%     sp      matriz conteniendo la información que define la
%              función spline
%              - típicamente será la la salida de
%                LINEAR_SPLINE, CUBIC_NAK o CUBIC_CLAMPED
%     x      valor(es) de la variable independiente donde se
%             evalúa la función spline
%             - escalar o vector
%
% salida:
%     y      valor(es) de la función spline
%
% NOTA:
%           rutina que acompaña a las CUBIC_NAK,
%           CUBIC_CLAMPED y LINEAR_SPLINE
%
[p c] = size ( sp );
npts = length ( x );

for i = 1 : npts
    piece = max ( find ( sp(1:p-1) < x(i) ) );
    if ( length ( piece ) == 0 )
        piece = 1;
    end

    temp(i) = sp(piece,2) + sp(piece,3) * ( x(i) - sp(piece,1) );
    if ( c == 5 )
        temp(i) = temp(i) + sp(piece,4) * ( x(i) - sp(piece,1) )^2;
        temp(i) = temp(i) + sp(piece,5) * ( x(i) - sp(piece,1) )^3;
    end
end
if ( nargout == 0 )
    disp ( temp )
else
    y = temp;
end
```

Los valores del polinomio obtenidos con *spline_eval*

```
octave:10> sp=[3.3653 2.9057 2.4182 0];
octave:11> x=[0.1 0.2 0.3 0.4];
octave:12> spline_eval( sp, x )
-4.9904 -4.7486 -4.5068 -4.2650
```

`function [A,df]=diffnew(X,Y)`  [DIFERENCIACIÓN NUMÉRICA](#)

```
%Ingreso      - X vector abcisa 1xn
%              - Y vector ordenada 1xn
%salida  - A vector 1xn que contiene los coeficientes del polinomio de grado
%              n de Newton
%              - df derivada aproximada
A=Y;
N=length(X);
for j=2:N
```




```

    for k=N:-1:j
        A(k)=(A(k)-A(k-1))/(X(k)-X(k-j+1));
    end
end

x0=X(1);
df=A(2);
prod=1;
n1=length(A)-1;

for k=2:n1
    prod=prod*(x0-X(k));
    df=df+prod*A(k+1);
end

```

`function [L,n]=difflim(x,toler)`  [DIFERENCIACIÓN NUMÉRICA](#)

```

%Ingreso - x punto de diferenciación
%         - toler tolerancia deseada
%Salida - L=[H' D' E']: H vector de tamaños de etapas
%         D vector de derivadas aproximadas

%         - E vector de cotas de error
%         - n coordenda de la mejor aproximación

f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
max1=15;
h=1;
H(1)=h;
D(1)=(feval(f,x+h)-feval(f,x-h))/(2*h);
E(1)=0;
R(1)=0;

for n=1:2
    h=h/10;
    H(n+1)=h;
    D(n+1)=(feval(f,x+h)-feval(f,x-h))/(2*h);
    E(n+1)=abs(D(n+1)-D(n));
    R(n+1)=2*E(n+1)*(abs(D(n+1))+abs(D(n))+eps);
end

n=2;

while ((E(n)>E(n+1)) & (R(n)>toler)) & n<max1
    h=h/10;
    H(n+2)=h;
    D(n+2)=(feval(f,x+h)-feval(f,x-h))/(2*h);
    E(n+2)=abs(D(n+2)-D(n+1));

```

```

R(n+2)=2*E(n+2)*(abs(D(n+2))+abs(D(n+1))+eps);
n=n+1;
end

n=length(D)-1;
L=[H' D' E'];

```

Dada la función x^2-3x , hallar su derivada en $x=3$ con tolerancia 0.01

difflim(3,0.01)

ingrese $f(x)$ entre comillas en términos de x ' x^2-3*x '

' x^2-3*x '

```

ans =
    1.00000    3.00000    0.00000
    0.10000    3.00000    0.00000
    0.01000    3.00000    0.00000

```

La salida es $L=[H' D' E']$: H vector de tamaños de paso; D el de deriv. aprox, E error

function [D,err,relerr,n]=diffext(f,x,delta,toler)



TÉCNICA DE RICHARDSON

```

%Ingreso      - f es la función
%              - x es el punto de diferenciación
%              - delta es la tolerancia para el error
%              - toler es la tolerancia para el error relativo
%Salida - D es la matriz de derivadas aproximadas
%          - err es la cota de error
%          - relerr es la cota de error relativo
%          - n es la coordenada de la "mejor aproximación"
% Si f es archivo M usar el llamado
%[D,err,relerr,n]=diffext(@f,x,delta,toler).
% Si f es una función anónima llamar como
%[D,err,relerr,n]=diffext(f,x,delta,toler).
%
err=1;
relerr=1;
h=1;
j=1;
D(1,1)=(f(x+h)-f(x-h))/(2*h);

while relerr > toler & err > delta & j < 12
    h=h/2;
    D(j+1,1)=(f(x+h)-f(x-h))/(2*h);
    for k=1:j
        D(j+1,k+1)=D(j+1,k)+(D(j+1,k)-D(j,k))/((4^k)-1);
    end
    err=abs(D(j+1,j+1)-D(j,j));
    relerr=2*err/(abs(D(j+1,j+1))+abs(D(j,j))+eps);
    j=j+1;
end

```

```
[n,n]=size(D);
```

Dada la función x^2-3x , hallar su derivada en $x=3$ con tolerancia 0.0001

```
octave:23> f=@(x)x^2-3*x;
```

```
octave:24> [D,err,relerr,n]=diffext(f,3,0.01,0.0001)
```

```
D =
```

```
3 0
```

```
3 3
```

```
err = 0
```

```
relerr = 0
```

```
n = 2
```

```
function s=traprl(a,b,M)  Regla Compuesta del Trapecio
```

```
%Ingreso - f es el integrando ingresado como string 'f'
```

```
% - a y b son los límites de integración
```

```
% - M es el número de subintervalos
```

```
%Salida - s es la suma de la regla trapezoidal
```

```
f=input('ingrese f(x) entre comillas en términos de x');
```

```
f=inline(f,'x');
```

```
h=(b-a)/M;
```

```
s=0;
```

```
for k=1:(M-1)
```

```
    x=a+h*k;
```

```
    s=s+feval(f,x);
```

```
end
```

```
s=h*(feval(f,a)+feval(f,b))/2+h*s;
```

Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$

Trapezoidal compuesta

```
octave:29> traprl(0,pi/3,6)
```

```
ingrese f(x) entre comillas en términos de x 'sin(x)'
```

```
'sin(x)'
```

```
ans = 0.49873
```

```
function s=simprl(a,b,M)  Regla compuesta de Simpson
```

```
# Ejemplo carga s=simprl(0,pi/3,6)
```

```
# funcion 'sin(x)'
```

```
# - a Limite inferior de integración
```

```
# - b Limite superior de integración
```

```
# - M Numero de subintervalos
```

```
f=input('ingrese f(x) entre comillas en términos de x');
```

```
f=inline(f,'x');
```

```

h=(b-a)/(2*M);
s1=0;
s2=0;
for k=1:M
    x=a+h*(2*k-1);
    s1=s1+feval(f,x);
end
for k=1:(M-1)
    x=a+h*2*k;
    s2=s2+feval(f,x);
end
s=h*(feval(f,a)+feval(f,b)+4*s1+2*s2)/3;

```

Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$
 Simpon's compuesta
 octave:30> **simp1(0,pi/3,6)**
 ingrese f(x) entre comillas en términos de x '**sin(x)**'
 'sin(x)'
 ans = 0.50000

`function [R,quad,err,h]=romber(a,b,n,tol)`  [Integración compuesta de Romberg](#)

```

%Ingreso - f es el integrando entrado como string 'f'
%         - a y b son los límites de integración
%         - n es el número máximo de filas en la tabla
%         - tol is the tolerance
%salida - R es la tabla de Romberg
%         - quad es el valor de cuadratura
%         - err es la estimación de error
%         - h es el menor tamaño de paso usado
f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
M=1;
h=b-a;
err=1;
J=0;
R=zeros(4,4);
R(1,1)=h*(feval(f,a)+feval(f,b))/2;

while ((err>tol)&(J<n)) | (J<4)
    J=J+1;
    h=h/2;
    s=0;
    for p=1:M
        x=a+h*(2*p-1);
        s=s+feval(f,x);
    end
    R(J+1,1)=R(J,1)/2+h*s;
    M=2*M;
    for K=1:J
        R(J+1,K+1)=R(J+1,K)+(R(J+1,K)-R(J,K))/(4^K-1);
    end
end

```

```

end
err=abs(R(J,J)-R(J+1,K+1));
end

```

Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$

Romberg compuesta

```
octave:31> romber(0,pi/3,6,0.001)
```

ingrese $f(x)$ entre comillas en términos de x '**sin(x)**'


```
'sin(x)'
```

```
ans =
```

```

0.45345 0.00000 0.00000 0.00000 0.00000
0.48852 0.50022 0.00000 0.00000 0.00000
0.49714 0.50001 0.50000 0.00000 0.00000
0.49929 0.50000 0.50000 0.50000 0.00000
0.49982 0.50000 0.50000 0.50000 0.50000

```

```
function Ih = trapezoidal(func,a,b,I2h,k)  Regla trapezoidal recursiva
```

```

% regal trapezoidal recursiva.
% USAR: Ih = trapezoid(func,a,b,I2h,k)
% func = handle de function a integrar.
% a,b = límites de integración.
% I2h = integral con 2^(k-1) paneles.
% Ih = integral con 2^k paneles.
if k == 1
fa = feval(func,a); fb = feval(func,b);
Ih = (fa + fb)*(b - a)/2.0;
else
n = 2^(k - 2); % Número de nuevos puntos
h = (b - a)/n; % espaciado de nuevos puntos
x = a + h/2.0; % Coord. Del primer punto nuevo
suma = 0.0;
for i = 1:n
fx = feval(func,x);
suma = suma + fx;
x = x + h;
end
Ih = (I2h + h*suma)/2.0;
end

```


Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$

trapezoidal recursive

```
octave:32> f=@(x) sin(x);
```

```
octave:33> Ih = trapezoidal(f,pi/3,6,1,0)
```

```
Ih = 0.50000
```

`function I = gausslegendre(func,a,b,n)`  [Fórmula de Gauss-Legendre](#)

```
% Cuadratura de Gauss-Legendre.
% USAR I = gaussQuad(func,a,b,n)
% Ingreso:
% func = handle de la function a integrar.
% a,b = límites de integración.
% n = orden de integración.
% Salida:
% I = integral
c1 = (b + a)/2; c2 = (b - a)/2; % constants de mapeo
[x,A] = gaussNodos(n); % pesos y abcisas nodales
sum = 0;
for i = 1:length(x)
y = feval(func,c1 + c2*x(i)); % Función en nodo i
sum = sum + A(i)*y;
end
I = c2*sum;

function [x,A] = gaussNodos(n,tol)
% Calcula abcisas nodales x y pesos A de
% cuadratura del punto n de Gauss-Legendre.
% USAR: [x,A] = gaussNodes(n,epsilon,maxIter)
% tol = tolerancia de error (default = 1.0e4*eps).
if nargin < 2; tol = 1.0e4*eps; end
A = zeros(n,1); x = zeros(n,1);
nRoots = fix(n + 1)/2; % Número de raíces no-neg.
for i = 1:n
t = cos(pi*(i - 0.25)/(n + 0.5)); % raíces aproximadas
for j = i:30
[p,dp] = legendre(t,n); % método de Newton
dt = -p/dp; t = t + dt; % búsqueda de raíz
if abs(dt) < tol %
x(i) = t; x(n-i+1) = -t;
A(i) = 2/(1-t^2)/dp^2; %
A(n-i+1) = A(i);
break
end
end
end

function [p,dp] = legendre(t,n)
% Evalua polinomio de Legendre p de grado n
% y su derivada dp en x = t.
p0 = 1.0; p1 = t;
for k = 1:n-1
p = ((2*k + 1)*t*p1 - k*p0)/(k + 1); % Eq. (6.19)
p0 = p1;p1 = p;
end
dp = n *(p0 - t*p1)/(1 - t^2);
```

Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$

```
Gauss-Legendre
octave:35> f=@(x) sin(x);
octave:36> I = gausslegendre(f,0,pi/3,25)
I = 0.50000
```


```

function [SRmat,quad,err]=adapt(a,b,tol)  Cuadratura adaptativa
%Ingresos - f es el integrando como string 'f'
% - a y b límites inferior y superior de integración
% - tol es la tolerancia
%salida - SRmat tabla de valores
% - quad valor de cuadratura
% - err error estimado
%Initialize values
f=input('ingrese f(x) entre comillas en términos de x');
f=inline(f,'x');
SRmat = zeros(30,6);
iterating=0;
done=1;
SRvec=zeros(1,6);
SRvec=srule(f,a,b,tol);
SRmat(1,1:6)=SRvec;
m=1;
state=iterating;

while(state==iterating)
    n=m;
    for j=n:-1:1
        p=j;
        SR0vec=SRmat(p,:);
        err=SR0vec(5);
        tol=SR0vec(6);
        if (tol<=err)

            %intervalo Biseccionado,aplica regla de Simpson
            %recursivamente, y determina error
            state=done;
            SR1vec=SR0vec;
            SR2vec=SR0vec;
            a=SR0vec(1);
            b=SR0vec(2);
            c=(a+b)/2;
            err=SR0vec(5);
            tol=SR0vec(6);
            tol2=tol/2;
            SR1vec=srule(f,a,c,tol2);
            SR2vec=srule(f,c,b,tol2);
            err=abs(SR0vec(3)-SR1vec(3)-SR2vec(3))/10;

            %prueba de exactitud
            if (err<tol)
                SRmat(p,:)=SR0vec;
                SRmat(p,4)=SR1vec(3)+SR2vec(3);
                SRmat(p,5)=err;
            else
                SRmat(p+1:m+1,:)=SRmat(p:m,:);
                m=m+1;
                SRmat(p,:)=SR1vec;
                SRmat(p+1,:)=SR2vec;
                state=iterating;
            end
        end
    end
end
quad=sum(SRmat(:,4));
err=sum(abs(SRmat(:,5)));
SRmat=SRmat(1:m,1:6);
    
```

function Z=srule(f,a0,b0,tol0)  [Regla de Simpson's](#)

```
%Ingreso - f es el integrando como string 'f'
% - a0 y b0 límites inferior y superior de integración
% - tol0 es la tolerancia
% salida - Z es un vector 1 x 6 [a0 b0 S S2 err toll]
h=(b0-a0)/2;
C=zeros(1,3);
C=feval(f,[a0 (a0+b0)/2 b0]);
S=h*(C(1)+4*C(2)+C(3))/3;
S2=S;
toll=tol0;
err=tol0;
Z=[a0 b0 S S2 err toll];
```

Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$
 Cuadratura adaptativa de Simpson's
 octave:37> **adapt(0,pi/3,0.0001)**
 ingrese f(x) entre comillas en términos de x '**sin(x)**'
 'sin(x)'
 ans = 0.00000 1.04720 0.50022 0.50001 0.00002 0.00010

function I= gauss_q(f,a,b,n)  [Cuadratura de Gauss-Legendre](#)

```
p=Legen_pw(n);
x=roots(p)';
x=sort(x);
for j=1:n
y=zeros(1,n);
y(j)=1;
p=polyfit(x,y,n-1);
P=poly_itg(p);
w(j)=polyval(P,1)- polyval(P,-1);
end
x=0.5*((b-a)*x+a+b);
y= feval(f,x);
I=sum(w.*y)*(b-a)/2;
fprintf('\n x          y          w\n')
for j=1:n
fprintf('%e %e %e\n', x(j) , y(j) , w(j))
end
```

```
function pn=legen_pw(n)
pbb=[1]; if n==0, pn=pbb; break; end
pb=[1 0]; if n==1, pn=pb; break; end
for i=2:n;
pn=((2*i-1)*[pb,0]-(i-1)*[0, 0, pbb])/i;
pbb=pb; pb=pn;
end
```




```
function py = poly_itg(p)
% poly_itg(p) integra un polinomio p que es una
% serie de potencias. El resultado también es una serie de potencias.
n=length(p);
py = [p.*[n:-1:1].^(-1),0];
```

Hallar la integral de la función $f(x)=\sin x$ entre 0 y $\pi/3$
 Cuadratura de Gauss-Legendre
 octave:37> I= gauss_q(@(x) sin(x),0,pi/3,20)

x	y	w
3.597857e-003	3.597849e-003	1.761401e-002
1.886425e-002	1.886314e-002	4.060143e-002
4.595395e-002	4.593777e-002	6.267205e-002
8.423816e-002	8.413857e-002	8.327674e-002
1.328203e-001	1.324301e-001	1.019301e-001
1.905618e-001	1.894106e-001	1.181945e-001
2.561094e-001	2.533188e-001	1.316886e-001
3.279267e-001	3.220809e-001	1.420961e-001
4.043304e-001	3.934032e-001	1.491730e-001
4.835296e-001	4.649070e-001	1.527534e-001
5.636680e-001	5.342903e-001	1.527534e-001
6.428672e-001	5.994927e-001	1.491730e-001
7.192708e-001	6.588363e-001	1.420961e-001
7.910881e-001	7.111187e-001	1.316886e-001
8.566357e-001	7.556433e-001	1.181945e-001
9.143772e-001	7.921827e-001	1.019301e-001
9.629594e-001	8.208853e-001	8.327674e-002
1.001244e+000	8.421423e-001	6.267205e-002
1.028333e+000	8.564397e-001	4.060143e-002
1.043600e+000	8.642209e-001	1.761401e-002

I =
 0.5000

function R=rk4(a,b,ya,M)  [Métodos de Runge-Kutta](#)

```
%Ingresos      - f es la función como string 'f'
%              - a y b extremos derecho e izquierdo
%              - ya condición inicial y(a)
%              - M es el número de etapas
%salida        - R = [T' Y'] con T vector de abcisas
%              e Y vector de ordenadas
```

```
f=input('ingrese f entre comillas en términos de t,y');
f=inline(f,'t','y');
h=(b-a)/M;
T=zeros(1,M+1);
Y=zeros(1,M+1);
T=a:h:b;
```

```
Y(1)=ya;
for j=1:M
    k1=h*feval(f,T(j),Y(j));
    k2=h*feval(f,T(j)+h/2,Y(j)+k1/2);
    k3=h*feval(f,T(j)+h/2,Y(j)+k2/2);
    k4=h*feval(f,T(j)+h,Y(j)+k3);
    Y(j+1)=Y(j)+(k1+2*k2+2*k3+k4)/6;
end

R=[T' Y'];
```

Sea la ecuación $y' = 0.5 \cdot (t - y)$ con $y(0) = 1$ en $[0, 3]$, se plantean los métodos de Adams Basforth-Moulton (4 pasos); Milne- Simpson (3 pasos), Hamming(3 pasos); todos requieren el cálculo previo de coordenadas T e Y (por ej con Runge Kutta de cuarto orden)

Seleccionando un paso de 0.25

octave:10> **R=rk4(0,3,1,12)**

ingrese f entre comillas en términos de t,y '0.5*(t-y) '
'0.5*(t-y) '

```
R =
    0.00000    1.00000
    0.25000    0.89749
    0.50000    0.83640
    0.75000    0.81187
    1.00000    0.81959
    1.25000    0.85579
    1.50000    0.91710
    1.75000    1.00059
    2.00000    1.10364
    2.25000    1.22396
    2.50000    1.35952
    2.75000    1.50852
    3.00000    1.66939
```

function A=abm(T,Y)  **Métodos de Adams-Moulton**

```
%Ingreso    - f función como string 'f'
%           - T vector de abcisas
%           - Y vector de ordenadas
% Las primeras cuatro coordendaas de T e Y deben
%           tener valores iniciales obtenidos con RK4
%salida - A=[T' Y'] con T vector de abcisas e
%           Y vector de ordenadas
```

```
f=input('ingrese f entre comillas en términos de t,y');
f=inline(f,'t','y');
n=length(T);
```

```

if n<5, return, end;

F=zeros(1,4);
F=feval(f,T(1:4),Y(1:4));
h=T(2)-T(1);

for k=4:n-1
    %Predictor
    p=Y(k)+(h/24)*(F*[-9 37 -59 55]');
    T(k+1)=T(1)+h*k;
    F=[F(2) F(3) F(4) feval(f,T(k+1),p)];
    %Corrector
    Y(k+1)=Y(k)+(h/24)*(F*[1 -5 19 9]');
    F(4)=feval(f,T(k+1),Y(k+1));
end

A=[T' Y'];

```


Empleando los datos obtenidos con el algoritmo rk4.

Desde la ventana de comandos hacemos:

```

octave:11> T=zeros(1,13);
octave:12> Y=zeros(1,13);
octave:13> T=0:1/4:3;
octave:14> Y(1:4)=[1 0.8975 0.8364 0.8119];
octave:15> abm(T,Y)
ingrese f entre comillas en términos de t,y '0.5*(t-y) '
'0.5*(t-y) '
ans =
    0.00000    1.00000
    0.25000    0.89750
    0.50000    0.83640
    0.75000    0.81190
    1.00000    0.81962
    1.25000    0.85580
    1.50000    0.91711
    1.75000    1.00060
    2.00000    1.10365
    2.25000    1.22396
    2.50000    1.35952
    2.75000    1.50852
    3.00000    1.66939

```

`function M=milne(T,Y)`  [**Método de Milne**](#)

```

%Ingreso - f función como 'f'
%          - T vector de abcisas
%          - Y vector de ordenadas
%. Las primeras cuatro coordendaas de T e Y deben
%          tener valores iniciales obtenidos con RK4

%salida - M=[T' Y'] con T vector de abcisas e

```

```
%           Y vector de ordenadas

f=input('ingrese f entre comillas en términos de t,y');
f=inline(f,'t','y');
n=length(T);
if n<5, break, end;

F=zeros(1,4);
F=feval(f,T(1:4),Y(1:4));
h=T(2)-T(1);
pold=0;
yold=0;


for k=4:n-1
    %Predictor
    pnew=Y(k-3)+(4*h/3)*(F(2:4)*[2 -1 2]');
    %Modifier
    pmod=pnew+28*(yold-pold)/29;
    T(k+1)=T(1)+h*k;
    F=[F(2) F(3) F(4) feval(f,T(k+1),pmod)];
    %Corrector
    Y(k+1)=Y(k-1)+(h/3)*(F(2:4)*[1 4 1]');
    pold=pnew;
    yold=Y(k+1);
    F(4)=feval(f,T(k+1),Y(k+1));
end
```

Empleando los datos obtenidos con el algoritmo rk4.
Desde la ventana de comandos hacemos:

```
octave:11> T=zeros(1,13);
octave:12> Y=zeros(1,13);
octave:13> T=0:1/4:3;
octave:14> Y(1:4)=[1 0.8975 0.8364 0.8119];
```

```
octave:15> milne(T,Y)
ingrese f entre comillas en términos de t,y '0.5*(t-y) '
'0.5*(t-y) '
ans =
```

```
0.00000 1.00000
0.25000 0.89750
0.50000 0.83640
0.75000 0.81190
1.00000 0.81958
1.25000 0.85582
1.50000 0.91709
1.75000 1.00062
2.00000 1.10362
2.25000 1.22399
2.50000 1.35950
2.75000 1.50855
3.00000 1.66937
```

`function H=heun(a,b,ya,M)`  **Método Predictor-Corrector**

```
# - a y b son los puntos finales de izquierda y derecha
# - ya condicion inicial y(a)
# - M Tamaño de paso
#Salidas - H=[T' Y'] T: vector de abcisas Y: vector de ordenadas
# Ejemplo carga H=heun(0,1,1,5)
# funcion '3*y+3*t'
f=input('ingrese f entre comillas en términos de t,y');
f=inline(f,'t','y');
h=(b-a)/M;
T=zeros(1,M+1);
Y=zeros(1,M+1);
T=a:h:b;
Y(1)=ya;
for j=1:M
    k1=feval(f,T(j),Y(j));
    k2=feval(f,T(j+1),Y(j)+h*k1);
    Y(j+1)=Y(j)+(h/2)*(k1+k2);
end
H=[T' Y'];
```

Dada la ecuación diferencial en PVI

$$y' = 3t + 3y$$

$y(0) = 1$, en $[0,1]$

Solución Exacta: $-1/3 - t + 4/3 \cdot \exp(3 \cdot t)$

H1 = heun(0,1,1,5)

ingrese f entre comillas en términos de t,y '3*y+3*t'

'3*y+3*t'

H1 =

```
0.00000 1.00000
0.20000 1.84000
0.40000 3.49120
0.60000 6.58634
0.80000 12.25168
1.00000 22.49199
```

`function X = backsub(A,B)`  **SUSTITUCIÓN HACIA ATRÁS**

```
# -----
# Llamada
# X = backsub(A,B)
# Parametros
# A Matriz de coeficientes triangular
# superior obtenida de gauss(A)
# B Vector lado derecho de la ecuacion
# Devuelve
# X Vector de Solucion
#Ejemplo carga X = backsub([4 -1 2 3;0 -2 7 -4;0 0 6 5;0 0 0 3],[20;-
7;4;6])
n = length(B);
det1 = A(n,n);
X = zeros(n,1);
X(n) = B(n)/A(n,n);
```

```
for r = n-1:-1:1,
    det1 = det1*A(r,r);
    if det1 == 0, break, end
    X(r) = (B(r) - A(r,r+1:n)*X(r+1:n))/A(r,r);
end
end
```

Resolver un sistema triangular (cuadrada), dada la matriz de coeficientes A y el vector independiente B

```
octave:17> A=[4 -1 2 3;0 -2 7 -4;0 0 6 5;0 0 0 3];
```

```
octave:18> B=[20;-7;4;6];
```

```
octave:19> backsub(A,B)
```

```
ans =
     3
    -4
    -1
     2
```

`function X = uptrbk(A,B)`  [SUSTITUCIÓN HACIA ATRÁS](#)

%Ingreso - A matriz N x N no singular

% - B N x 1 matriz

%salida - X es matriz N x 1 con la solución de AX=B.

%Inicia a lizar X y la matriz temporario de almacenaje C

[N N]=size(A);

X=zeros(N,1);

C=zeros(1,N+1);

%Forma la matriz aumentada: Aug=[A|B]

Aug=[A B];

for p=1:N-1

 %Pivoteo parcial para columna p

 [Y,j]=max(abs(Aug(p:N,p)));

 %Intercambio fila p y j

 C=Aug(p,:);

 Aug(p,:)=Aug(j+p-1,:);

 Aug(j+p-1,:)=C;

 if Aug(p,p)==0

 'A es singular. No hay solución única'

 break

 end

 %etapa de eliminación para columna p

 for k=p+1:N

 m=Aug(k,p)/Aug(p,p);

 Aug(k,p:N+1)=Aug(k,p:N+1)-m*Aug(p,p:N+1);

 end

end

%sustitución hacia atrás en [U|Y]

X=backsub(Aug(1:N,1:N),Aug(1:N,N+1));

Resolver un sistema triangular (cuadrada), dada la matriz de coeficientes A y el vector independiente B , empleando triangularización superior más sustitución hacia atrás

```
octave:22> A=[1 2 2 4;2 0 4 3;4 2 2 1;-3 1 3 2]; % no singular
```

```
octave:23> B=[13 28 20 6]';
```

```
octave:24> uptrbk(A,B)
```

```
ans =  
    3.00000  
   -1.33333  
    5.00000  
    0.66667
```

```
function Gjelim(P)
```



TÉCNICA DE GAUSS-JORDAN

```
%Gjelim  
%eliminación de Gauss-Jordan.  
%Opciones: formato en numero racional  
%           conteo de operaciones  
%           todas las etapas  
%forma de llamado: Gjelim(A)  
  
%puede usarse tolerancia de 1e-20  
%o cambiar a una propia  
%MATLAB calcula hasta 16 dígitos decimales  
%  
%valores iniciales  
adds=0;totadds=0;mults=0;totmults=0;swaps=0;totswaps=0;  
ops=[0];  
  
%hold off  
%  
tol=1e-20;  
[n,m]=size(P);  
format compact  
  
disp(' ')  
h=input('Números racionales? y/n: ','s');  
q=input('Conteo de operaciones? y/n: ','s');  
g=input('todas las etapas? y/n: ','s');  
  
disp(' ')  
disp('matriz inicial')  
if h=='y'  
    disp(rats(P))  
else  
    disp(P)  
end  
  
if g=='y';  
    disp('[presione Enter en cada paso para seguir]')  
    disp(' ')  
    pause  
end
```

```

%busca un pivot
j=1;
for i=1:n,
    if j <= m
        found=0;
        if abs(P(i, j)) <= tol %fin está en línea 101
            while (found == 0) %
%busca de unos e intercambio de filas si hace falta
                for s=i:n,
                    if (abs(P(s, j)) > tol)
                        if (found == 0)
                            found=1;
                            if s~=i
                                for r=j:m,
                                    temp=P(i, r);
                                    P(i, r)=P(s, r);
                                    P(s, r) = temp;
                                end
                                swaps = m-j+1;
                                totswaps = totswaps + swaps;
                                if g=='y'; %todas las etapas
                                    disp('intercambio filas')
                                if h=='y'
                                    disp(rats(P))
                                else
                                    disp(P)
                                end
                                if q=='y'
                                    disp('intercambio de elementos:')
                                    disp(swaps)
                                end
                                disp('-----')
                                pause
                                end %todas las etapas
                            end
                        end
                    end
                end
            end
            if (found==0)
                if (j <= m)
                    j = j + 1;
                end
            end
            if j>m
                found=1;
            end
            end %
            if j > m
                found = 0;
            end
            else
                found = 1;
            end %parte línea 51

%normaliza elemento lider en la fika ajustando el resto
if found == 1
    k=i;
    if (P(k, j) ~= 1)

```



```

if (abs(P(k, j)) > tol)
    y = P(i, j);
    for l=j:m,
        P(k, l) = P(k, l)/y ;
    end
    mults = m-j;
    totmults = totmults + mults;
    if g=='y'; %todas las etapas
        disp('normaliza')
        if h=='y'
            disp(rats(P))
        else
            disp(P)
        end
        if q=='y'
            disp('multiplicaciones:')
            disp(mults)
        end
        disp('-----')
        pause
    end %todas las etapas
end
end
for r=1:n,
    if (abs(P(r, j)) > tol)
        if (r ~= i)
            z=P(r, j);
            for c=j:m,
                P(r, c)=P(r, c) - z * P(i, c);
            end
            adds = m-j;
            mults = m-j;
            totadds = totadds + adds;
            totmults = totmults + mults;
            if g=='y'; %todas las etapas
                disp('crea cero')
                if h=='y'
                    disp(rats(P))
                else
                    disp(P)
                end
                if q=='y'
                    disp('sumas, multiplicaciones:')
                    ops=[adds mults];
                    disp(ops)
                end
                disp('-----')
                pause
            end %todas las etapas
        end
    end
end
end
end

j = j + 1;

end
end %fin loop i

%muestra matriz final
disp('-forma escalonada reducida-')


```

```

if h=='y'
    disp(rats(P))
else
    disp(P)
end

if q=='y'
    disp('Total de sumas, multiplicaciones, intercambio elementos:')
    ops=[totadds totmults totswaps];
    disp(ops)
    disp('-----')
end

disp(' ')
format loose


function x = gaussPiv(A,b)  PIVOTEO
% resuelve A*x = b por eliminación de Gauss con pivoteo de filas
% USAR: x = gaussPiv(A,b)
if size(b,2) > 1; b = b'; end
n = length(b); s = zeros(n,1);
%-----establce arreglo factor escala-----
for i = 1:n; s(i) = max(abs(A(i,1:n))); end
%-----intercambio filas si es necesario-----
for k = 1:n-1
    [Amax,p] = max(abs(A(k:n,k))./s(k:n));
    p = p + k - 1;
    if Amax < eps; error('Matriz es singular'); end
    if p ~= k b = swapRows(b,k,p); s = swapRows(s,k,p); A = swapRows(A,k,p);
end
%-----Eliminación-----
for i = k+1:n
    if A(i,k) ~= 0
        lambda = A(i,k)/A(k,k);
        A(i,k+1:n) = A(i,k+1:n) - lambda*A(k,k+1:n);
        b(i) = b(i) - lambda*b(k);
    end
end
%-----fase de sustitución hacia atrás-----
for k = n:-1:1
    b(k) = (b(k) - A(k,k+1:n)*b(k+1:n))/A(k,k);
end
x = b;

function v = swapRows(v,i,j)
% Swap rows i and j of vector or matrix v.
% USAGE: v = swapRows(v,i,j)
temp = v(i,:);
v(i,:) = v(j,:);
v(j,:) = temp;

```

resuelve $A \cdot x = b$ por eliminación de Gauss con pivoteo de filas, dada la matriz de coeficientes A y el vector independiente B , empleando triangularización superior más sustitución hacia atrás

```
octave:49> A=[4 -1 2 3;0 -2 7 -4;0 0 6 5;0 0 0 3];
octave:50> B=[20;-7;4;6];
octave:51> x = gaussPiv(A,B)
x =
     3
    -4
    -1
     2
```

`function X = lufact(A,B)`  [Factorización LU](#)

```
%Ingreso - A es matriz N x N
%          - B matriz N x 1
%salida - X matriz N x 1 conteniendo solución de AX = B.
```

```
%Inicializar X, Y,matriz temporario de almacenaje C, y la matriz de
información de permutación fila R
```

```
[N,N]=size(A);
X=zeros(N,1);
Y=zeros(N,1);
C=zeros(1,N);
R=1:N;
```

```
for p=1:N-1
```

```
    %Busca fila pivot para columna p
    [max1,j]=max(abs(A(p:N,p)));
```

```
    %Intercambia fila p y j
    C=A(p,:);
    A(p,:)=A(j+p-1,:);
    A(j+p-1,:)=C;
    d=R(p);
    R(p)=R(j+p-1);
    R(j+p-1)=d;
```

```
    if A(p,p)==0
        'A es singular. Sin solución única'
        break
    end
```

```
    %Calcula multiplicadores y ubica en la subdiagonal de A
    for k=p+1:N
        mult=A(k,p)/A(p,p);
```

```

    A(k,p) = mult;
    A(k,p+1:N)=A(k,p+1:N)-mult*A(p,p+1:N);
end
end

%resuelve para Y
Y(1) = B(R(1));
for k=2:N
    Y(k)= B(R(k))-A(k,1:k-1)*Y(1:k-1);
end

%resuelve para X
X(N)=Y(N)/A(N,N);
for k=N-1:-1:1
    X(k)=(Y(k)-A(k,k+1:N)*X(k+1:N))/A(k,k);
end

```

Resolver un sistema triangular (cuadrada), dada la matriz de coeficientes A y el vector independiente B, con factorización con pivoteo

```
octave:49> A=[4 -1 2 3;0 -2 7 -4;0 0 6 5;0 0 0 3];
```


```
octave:50> B=[20;-7;4;6];
```

```
octave:55> X = lufact(A,B)
```

```

X =
    3.00000
   -1.33333
    5.00000
    0.66667

```

```
function [P,dP,Z] = jacobi(A,B,P,delta,max1)  Método de Jacobi
```

```

%JACOBI iteración de Jacobi para resolver un sistema lineal.
% forma de llamado
% [X,dX] = jacobi(A,B,P,delta,max1)
% [X,dX,Z] = jacobi(A,B,P,delta,max1)
% Ingresos
% A matriz de coeficientes
% B vector lado derecho
% P vector de arranque
% delta tolerancia de convergencia
% max1 maximo número de iteraciones
% devuelve
% X vector solución
% dX vector de estimación error
% Z matriz de historia de iteraciones
%
Z = P';
n = length(B);
Pnew = P;
for k=1:max1,
    for r = 1:n,
        Sum1 = B(r) - A(r,[1:r-1,r+1:n])*P([1:r-1,r+1:n]);
        Pnew(r) = Sum1/A(r,r);
    end
end

```

```

end
dP = abs(Pnew-P);
err = norm(dP);
relerr = err/(norm(Pnew)+eps);
P = Pnew;
Z = [Z;P'];
if (err<delta)|(relerr<delta), break, end
end


```

Resolución por técnicas iterativas , dada la matriz de coeficientes y el vector independiente

```

octave:35> A=[1 -5 -1;4 1 -1;2 -1 -6];
octave:36> B=[-8 13 -2]';
octave:37> P=[0 0 0]';
octave:38> jacobi(A,B,P,0.001, 10) con 10 iteraciones
ans =
    9.1711e+006
    5.8774e+006
   -8.0714e+005
octave:39> jacobi(A,B,P,0.001, 20) con 20 iteraciones
ans =
   -3.0646e+013
   -1.4564e+013
    2.2794e+012

```

`function [P,dP,Z] = gseid(A,B,P,delta,max1)`  [Método de Gauss-Seidel](#)

```

# GSEID Iteracion para resolver sistemas lineales Gauss-Seidel
# Llamado de la funcion
# [X,dX] = gseid(A,B,P,delta,max1)
# [X,dX,Z] = gseid(A,B,P,delta,max1)
# Entradas
# A Matriz de coeficientes
# B vector de soluciones
# P vector de inicio
# delta tolerancia
# max1 maximo numero de iteraciones
# Devuelve
# P vector solución
# dP vector de estimación error
# Z matriz de historia de las iteraciones
# Ejemplo carga [P,dP,Z] = gseid([1 2 3;2 4 3;3 6 4],[-
1;3;0],[0;0;0],0.0001,20)
Z = P';
n = length(B);
Pold = P;
for k=1:max1,
for r = 1:n,
Sum1 = B(r) - A(r,[1:r-1,r+1:n])*P([1:r-1,r+1:n]);
P(r) = Sum1/A(r,r);

```

```

end
dP = abs(Pold-P);
err = norm(dP);
relerr = err/(norm(P)+eps);
Pold = P;
Z = [Z;P'];
if (err<delta)|(relerr<delta), break, end
end
end


```

Resolución por técnicas iterativas , dada la matriz de coeficientes y el vector independiente

```

octave:40> A=[1 -5 -1;4 1 -1;2 -1 -6];
octave:41> B=[-8 13 -2]';
octave:42> P=[0 0 0]';
octave:43> gseid(A,B,P,0.001,10)
ans =
    3.8556e+012
   -1.5624e+013
    3.8892e+012

```

`function x = conjgrad(A,b,tol)`  [GRADIENTE CONJUGADO](#)

```

% CONJGRAD Metodo del gradiente conjugado.
% X = CONJGRAD(A,B) busca resolver el sistema lineal A*X=B
% para X. La matriz de coeficientes ,NxN,A debe ser simétrica y el vector
% column B de longitud N.
%
% X = CONJGRAD(A,B,TOL) especifica la tolerancia del método, por
% default es 1e-10.
%
% Ejemplo:
%{
n = 6000;
m = 8000;
A = randn(n,m);
A = A * A';
b = randn(n,1);
tic, x = conjgrad(A,b); toc
norm(A*x-b)
%}

if nargin<3
    tol=1e-10;
end
r = b + A*b;
y = -r;
z = A*y;
s = y'*z;
t = (r'*y)/s;
x = -b + t*y;

for k = 1:numel(b);
    r = r - t*z;
    if( norm(r) < tol )
        return;
    end
    B = (r'*z)/s;
    y = -r + B*y;
    z = A*y;
    s = y'*z;

```

```
t = (r'*y)/s;  
x = x + t*y;  
end  
end
```

Utilizar el método del Gradiente conjugado(directo no es tan bueno como elim. Gauss con pivoteo, sí para iterativos)


```
octave:44> A=[4 3 0;3 4 -1;0 -1 4]; % debe ser simétrica  
octave:45> b=[24 30 -24]';  
octave:46> conjgrad(A,b,0.001)  
ans =  
    3.0000  
    4.0000  
   -5.0000
```

%El método del gradiente conjugado ayuda a resolver el sistema $Ax=b$, con A simétrica, sin calcula inversa de A. Sólo requiere poca memoria, siendo adecuado para sistemas de gran escala.

%es más rápido que otros como eliminación Gaussiana, si A está bien condicionada. Por ejemplo,

```
%n=1000;  
%[U,S,V]=svd(randn(n));  
%s=diag(S);  
%A=U*diag(s+max(s))*U'; % A es simétrica, bien condicionada  
%b=randn(1000,1);  
%tic,x=conjgrad(A,b);toc  
%tic,x1=A\b;toc  
%norm(x-x1)  
%norm(x-A*b)
```

%El GC es de dos a tres veces más rápido que $A\b$, que emplea eliminación Gaussiana

`function [C]=gerschgorin(A)`  [VALORES PROPIOS](#)

% Gershgorin's circles C of the matrix A.

```
d = diag(A);  
cx = real(d);  
cy = imag(d);  
B = A - diag(d);  
[m, n] = size(A);  
r = sum(abs(B'));  
C = [cx cy r(:)];  
t = 0:pi/100:2*pi;  
c = cos(t);  
s = sin(t);  
[v,d] = eig(A);  
d = diag(d);  
u1 = real(d);  
v1 = imag(d);  
hold on
```

```

grid on
axis equal
xlabel('Re')
ylabel('Im')
h1_line = plot(u1,v1,'or');
set(h1_line,'LineWidth',1.5)
for i=1:n
x = zeros(1,length(t));
y = zeros(1,length(t));
x = cx(i) + r(i)*c;
y = cy(i) + r(i)*s;
h2_line = plot(x,y);
set(h2_line,'LineWidth',1.2)
end
hold off
title('Gershgorin circles and the eigenvalues of a')

```

Para A, acotar los autovalores empleando los círculos de Gerschgorin

```
octave:59> [C] = gerschgorin([4 1 1;0 2 1;-2 0 9])
```

```

C =
    4    0    2
    2    0    1
    9    0    2

```

```
function [lambda,V]= power1(A,X,epsilon,max1)
```



[Métodos de Potencias](#)

```

%Ingreso - A matriz nxn
%         - X vector de arranque nx1
%         - epsilon es la tolerancia
%         - max1 número máximo de iteraciones
%salida - lambda eigenvalue dominante
%         - V eigenvector dominante

```

```
%Inicializar parámetros
```

```

lambda=0;
cnt=0;
err=1;
state=1;

```

```

while ((cnt<=max1)&(state==1))
    Y=A*X;

```

```

    %Normaliza Y
    [m j]=max(abs(Y));
    c1=m;
    dc=abs(lambda-c1);
    Y=(1/c1)*Y;

```

```

    %actualiza X y lambda y chequea para convergencia
    dv=norm(X-Y);

```



```

err=max(dc,dv);
X=Y;
lambda=c1;
state=0;
if (err>epsilon)
    state=1;
end
cnt=cnt+1;
end

```

V=X;

Dada A=[0 11 -5;-2 17 -7;-4 26 -10]; hallar el autovalor y autovector dominante

octave:58> **A=[0 11 -5;-2 17 -7;-4 26 -10];**

octave:59> **X=[1 1 1]';**

octave:60> **power1(A,X,0.001,10)**

ans = 4.0016

function [lambda,V]=invpow(A,X,alpha,epsilon,max1)  [Método de la potencia inversa](#)

```

%Ingreso - A matriz nxn
%         - X vector de arranque nx1
%         - alpha dado
%         - epsilon tolerancia
%         - max1 número máximo de iteraciones
%salida - lambda es el eigenvalue dominante
%         - V eigenvector dominante

```

%Inicializar matriz A-alphaI y parámetros

```

[n n]=size(A);
A=A-alpha*eye(n);
lambda=0;
cnt=0;
err=1;
state=1;

```

```

while ((cnt<=max1) & (state==1))

```

```

    %resuelve sistema AY=X
    Y=A\X;

```

```

    %Normaliza Y
    [m j]=max(abs(Y));
    c1=m;
    dc=abs(lambda-c1);
    Y=(1/c1)*Y;

```


```

%actualiza X y lambda y chequea la convergencia
dv=norm(X-Y);
err=max(dc,dv);
X=Y;
lambda=c1;
state=0;
if (err>epsilon)
    state=1;
end
cnt=cnt+1;
end

lambda=alpha+1/c1;
V=X;

```

Dada $A = \begin{bmatrix} 0 & 11 & -5 \\ -2 & 17 & -7 \\ -4 & 26 & -10 \end{bmatrix}$; hallar el autovalor y autovector dominante
 Por el método de inverso de potencias
 octave:61> **A=[0 11 -5;-2 17 -7;-4 26 -10];**
 octave:62> **X=[1 1 1]';**
 octave:63> **alpha=4.1;**
 octave:64> **invpow(A,X,alpha,0.001,10)**
 ans = 4.2000

function [V,D]=jacobil(A,epsilon)  **METODO CLASICO DE JACOBI**

```

%Ingreso - A matriz nxn
%      - epsilon tolerancia
%salida - V matriz nxn de eigenvectors
%      - D matriz diagonal nxn de eigenvalues
%Inicializar V, D, y parámetros
D=A;
[n,n]=size(A);
V=eye(n);
state=1;

%Calcula fila p y columna q del elementodiagonal más grande deelement
% A

[m1 p]=max(abs(D-diag(diag(D))));
[m2 q]=max(m1);
p=p(q);

while (state==1)
    %elim. cero Dpq y Dqp
    t=D(p,q)/(D(q,q)-D(p,p));
    c=1/sqrt(t^2+1);
    s=c*t;
    R=[c s;-s c];
    D([p q],:)=R'*D([p q],:);
    D(:,[p q])=D(:,[p q])*R;
    V(:,[p q])=V(:,[p q])*R;
    [m1 p]=max(abs(D-diag(diag(D))));
    [m2 q]=max(m1);
    p=p(q);
end

```

```

    if (abs(D(p,q)) < epsilon * sqrt(sum(diag(D).^2)/n))
        state=0;
    end
end
D=diag(diag(D));

```

Para A simétrica, hallar sus autovalores y autovectores

```
octave:65> A=[8 -1 3 -1; -1 6 2 0; 3 2 9 1; -1 0 1 7];
```

```
octave:66> jacob1(A,0.001)
```

```
ans =
    0.528779 -0.572958 0.582194 0.230569
    0.591968 0.472072 0.176060 -0.629067
   -0.536039 0.282024 0.792519 -0.070981
    0.287453 0.607725 0.044347 0.738968
```

La salida es una matriz 4x4 de eigenvectores y diagonal de eigenvalores

`function T=house(A)`  [Técnica de Householder](#)

```
%Ingreso - A matriz simétrica nxn
%salida - T matriz tridiagonal
```

```
[n,n]=size(A);
```

```

for k=1:n-2
    s=norm(A(k+1:n,k));
    if (A(k+1,k)<0)
        s=-s;
    end
    r=sqrt(2*s*(A(k+1,k)+s));
    W(1:k)=zeros(1,k);
    W(k+1)=(A(k+1,k)+s)/r;
    W(k+2:n)=A(k+2:n,k)'/r;
    V(1:k)=zeros(1,k);
    V(k+1:n)=A(k+1:n,k+1:n)*W(k+1:n)';
    c=W(k+1:n)*V(k+1:n)';
    Q(1:k)=zeros(1,k);
    Q(k+1:n)=V(k+1:n)-c*W(k+1:n);
    A(k+2:n,k)=zeros(n-k-1,1);
    A(k,k+2:n)=zeros(1,n-k-1);
    A(k+1,k)=-s;
    A(k,k+1)=-s;
    A(k+1:n,k+1:n)=A(k+1:n,k+1:n) ...
        -2*W(k+1:n)'*Q(k+1:n)-2*Q(k+1:n)'*W(k+1:n);
end
T=A;

```


Reducir una matriz simétrica a la forma tridiagonal (por transformación de Householder)

```
octave:67> A=[8 -1 3 -1; -1 6 2 0; 3 2 9 1; -1 0 1 7];
```

```
octave:68> house (A)
```

```
ans =
```

```
      8.00000    3.31662    0.00000    0.00000
      3.31662    6.90909   -2.46630    0.00000
      0.00000   -2.46630    8.24308   -0.79311
      0.00000    0.00000   -0.79311    6.84783
```

```
function D=qr1(A,epsilon)  Algoritmo QR
```

```
%Ingreso - A matriz tridiagonal simétrica nxn
%          - epsilon tolerancia
%salida - D vector de eigenvalues
```

```
%Inicializar parámetros
```

```
[n,n]=size(A);
```

```
m=n;
```

```
while (m>1)
```

```
    S=A(m-1:m,m-1:m);
```

```
    if abs(S(2,1))<epsilon
```

```
        A(m,m-1)=0;
```

```
        A(m-1,m)=0;
```

```
    else
```

```
        shift=eig(S);
```

```
        [j,k]=min([abs(A(m,m)-shift(1)) abs(A(m,m)-shift(2))]);
```

```
    end
```

```
    [Q,U]=qr(A-shift(k)*eye(n));
```

```
    A=U*Q+shift(k)*eye(n);
```

```
    m=m-1;
```

```
end
```

```
D=diag(A);
```

Aproximar los autovalores con el método QR

```
octave:69> A=[8 -1 3 -1; -1 6 2 0; 3 2 9 1; -1 0 1 7];
```

```
octave:70> qr1(A,0.001)
```


```
ans =
```

```
      3.2976
```

```
      8.4547
```

```
     11.6541
```

```
      6.5936
```

`function D=qr2(A,epsilon)`  **Algoritmo QR**

%Ingreso- A matriz simétrica tridiagonal nxn
% - epsilon tolerancia
%salida - D nx1 vector de eigenvalues

%Inicializar parametros

[n,n]=size(A);
m=n;
D=zeros(n,1);
B=A;

while (m>1)
 while (abs(B(m,m-1))>=epsilon)

 %Calcula shift
 S=eig(B(m-1:m,m-1:m));
 [j,k]=min(abs(B(m,m)*[1 1]'-S));

 %factorización QR de B
 [Q,U]=qr(B-S(k)*eye(m));

 %Calcula siguiente B
 B=U*Q+S(k)*eye(m);
 end

 %ubica el m-simo eigenvalue en A(m,m)
 A(1:m,1:m)=B;

 %Repite proceso en m-1 x m-1 submatriz de A
 m=m-1;
 B=A(1:m,1:m);

end
D=diag(A);

Aproximar los autovalores con el método QR

octave:71> **A=[8 -1 3 -1; -1 6 2 0; 3 2 9 1; -1 0 1 7];**

octave:72> **qr2(A,0.001)**

ans =
 11.7043
 3.2957
 8.4077
 6.5923

Con A tridiagonal simétrica cuadrada

`function [Q, R] = grams(A)`  [Ortogonalización de Gram-Schmidt](#)

```
% gschmidt.m: Ortogonalización de las columnas de A
% vía el proceso de Gram-Schmidt.
% Se asumen que las columnas de A son LI
%
% Forma de uso:
% Q = grams(A) regresa una matriz Q (de orden m x n) cuyas
% columnas forman una base ortonormal para el espacio de A.
%
% [Q, R] = grams(A) regresa una matriz Q con columnas ortonormales
% y R como una matriz triangular superior de tal forma que A = Q*R.
%
[m, n] = size(A);
Asave = A;
for j = 1:n
    for k = 1:j-1
        mult = (A(:, j)'*A(:, k)) / (A(:, k)'*A(:, k));
        A(:, j) = A(:, j) - mult*A(:, k);
    end
end
for j = 1:n
    if norm(A(:, j)) < sqrt(eps)
        error('Las columnas de A son linalmente independientes.')
    end
    Q(:, j) = A(:, j) / norm(A(:, j));
end
R = Q'*Asave;
```

Dados la matriz A aplicar la ortogonalización de Gram Schdmitt
Grams Schmidt

A=[8 -1 3 -1; -1 6 2 0; 3 2 9 1; -1 0 1 7];

octave:75> **[Q, R] = grams(A)**


Q =


```
0.923760 -0.023148 -0.322415 0.205377
-0.115470 0.930114 -0.334071 0.099754
0.346410 0.366154 0.841005 -0.196575
-0.115470 -0.016835 0.277769 0.953534
```

R =

```
8.6603e+000 -9.2376e-001 5.5426e+000 -1.3856e+000
2.0817e-017 6.3361e+000 5.0693e+000 2.7146e-001
5.5511e-017 6.6613e-016 6.2114e+000 3.1078e+000
-1.1102e-016 -2.7756e-016 -2.2204e-016 6.2728e+000
```

OBSERVACION: La versión de octave a emplear para el algoritmo de broyden deberá contar con el paquete "simbols", dado que sin el mismo no puede operarse con variables simbólicas.

```
function [x,k] = broyden(fcn1,fcn2,x0,maxits,tol)  CUASI-NEWTON
% sintaxis: [X] = BROYDEN(FCN1,FCN2,X0,MAXITS)
% Una función para calcular un sistema 2x2 de ec. no lineales
% los ingresos mínimos requeridos son fcn1 y fcn2
% tener cuidado al ingresar fcn1,fcn2 please usar sólo x1 y x2
%
%
% ingresos - FCN1: primera ecuación como string
%           - FCN2: segunda ecuación como string
%           - X0: aproximación inicial de la solución, default es [1 1]
%           - MAXITS: máximo número de iteraciones, default es 50
%           - TOL: tolerancia, default es 1e-8
% salida - X: the solución del sistema
%         - K: número de iteraciones para llegar a la solución
%
%
if nargin < 5
tol = 1e-8;
end
if nargin < 4
maxits = 50;
end
if nargin < 3
x0 = [1 1]';
end
if nargin < 2
help broyden
error('dos entradas como mínimo')
end
syms x1 x2
B = [diff(fcn1,x1) diff(fcn1,x2);diff(fcn2,x1) diff(fcn2,x2)];
x1 = x0(1);
x2 = x0(2);
h = inline(fcn1,'x1','x2');
g = inline(fcn2,'x1','x2');
f(1:2,1) = [h(x1,x2);g(x1,x2)];
B = eval(B);
x = [x1 x2]';
for k=1:maxits
s = B\(-f);
x = x + s;
fnew = [h(x(1),x(2));g(x(1),x(2))];
y = fnew-f;
if abs(fnew-f) < tol
break
end
f = fnew;
B = B + ((y-B*s)*s')/(s'*s);
end
```

`function [x,fx,xx]=newtons(f,x0,TolX,MaxIter,varargin)`  **MÉTODO DE NEWTON**

```
%newtons.m resuelve conjunto de ec. no lineales f1(x)=0, f2(x)=0,..
%ingreso: f = vector 1^st-orden ftn equivalente a un conjunto de ecuaciones
% x0 = suposición inicial de solución
% TolX = límite superior de |x(k) - x(k - 1)|
% MaxIter = máximo número de iteraciones
% salida: x = punto donde el algoritmo llegó
% fx = f(x(última))
% xx = historia de x
h = 1e-4; TolFun = eps; EPS = 1e-6;
fx = feval(f,x0,varargin{:});
Nf = length(fx); Nx = length(x0);
if Nf ~= Nx; error('dimensiones incompatible de f y x0!'); end
if nargin < 4, MaxIter = 100; end
if nargin < 3, TolX = EPS; end
xx(1,:) = x0(:).'; %Inicializar la solución como vector fila inicial
%fx0 = norm(fx); % (1)
for k = 1: MaxIter dx = -jacob(f,xx(k,:),h,varargin{:})\fx(:); %-[dfdx]^-1*fx
%para l = 1: 3 %damping para eviatar divergencia % (2)
%dx = dx/2; % (3)
xx(k + 1,:) = xx(k,:) + dx.';
fx = feval(f,xx(k + 1,:),varargin{:}); fxn = norm(fx);
% if fxn < fx0, break; end % (4)
%end % (5)
if fxn < TolFun | norm(dx) < TolX, break; end
%fx0 = fxn; % (6)
end
x = xx(k + 1,:);
if k == MaxIter;
fprintf('la mejor en %d iteraciones\n',MaxIter);
end

function g = jacob(f,x,h,varargin) %Jacobiano de f(x)
if nargin < 3; h = 1e-4; end
h2 = 2*h; N = length(x); x = x(:).'; I = eye(N);
for n = 1:N
g(:,n) = (feval(f,x + I(n,:)*h,varargin{:}) ...
-feval(f,x - I(n,:)*h,varargin{:}))/h2;
end
```

%Método del disparo Lineal  **TECNICA DE DISPARO PARA EL PROBLEMA LINEAL**

```
% - - - - -
%
% este programa implementa el método de disparo lineal
%
% para resolver el PVF
%
%      x'' = p(t)x'(t) + q(t)x(t) + r(t)
%
% sobre [a,b] con x(a) = alpha y x(b) = beta
%
```



```
% Usa el cambio de variable  $y = x'$  y las
%
% funciones  $p(t)x'(t) + q(t)x(t) + r(t)$ 
%
%  $p(t)x'(t) + q(t)x(t)$ 
%
% para formar fn1.m y fn2.m
% -----
% -----
%
% Aplica el método de Runge-Kutta para un orden más alto para resolver:
%
%  $u'' = p(t)u'(t) + q(t)u(t) + r(t)$ 
% con  $u(a) = \alpha$  y  $u'(a) = 0$ 
%
%  $v'' = p(t)v'(t) + q(t)v(t)$ 
% con  $v(a) = 0$  y  $v'(a) = 1$ 
%
% Define y almacena la función fn1(t,Z) y fn2(t,Z)
% en los archivos m fn1.m y fn2.m
% function W = fn1(t,Z)
% x = Z(1); y = Z(2);
% W = [y, (2*t*y/(1 + t^2) - 2*x/(1 + t^2) + 1)];
% function W = fn2(t,Z)
% x = Z(1); y = Z(2);
% W = [y, (2*t*y/(1 + t^2) - 2*x/(1 + t^2))];
%.....
% inicia una sección que entra la/s funcion(es) necesarias para el ejemplo
% en archivos m(s) ejecutando el comando diary command en este script.
% el método de programación preferido no usa estas etapas.
% uno debe entrar la función(es) en los M con un editor.
delete output;
delete fn1.m;
diary fn1.m; disp('funcion W = fn1(t,Z)');...
disp('x = Z(1); y = Z(2);');...
disp('W = [y, (2*t*y/(1 + t^2) - 2*x/(1 + t^2) + 1)];');...
diary off;
delete fn2.m;
diary fn2.m; disp('funcion W = fn2(t,Z)');...
disp('x = Z(1); y = Z(2);');...
disp('W = [y, (2*t*y/(1 + t^2) - 2*x/(1 + t^2))];');...
diary off;
fn1(0,[0,0]);
fn2(0,[0,0]);
% . fn1.m fn2.m rks4.m se usan

% -----
%
% Usa el método disparo lineal para resolver
% el PVF  $x'' = 2t/(1+t^2) x' - 2/(1+t^2) + 1$ 
% con  $x(0) = 1.25$  y  $x(4) = -0.95$ .
%
% Entrar los extremos de [a,b].
% Entrar valor inicial  $x(a)$  en alpha.
% Entrar valor terminal  $x(b)$  en beta.
% Entrar número de subintervalos en m.

a = 0;
b = 4;
alpha = 1.25;
beta = -0.95;
```

```

m = 20;
% -----
%
% resolver u' = p(t)u'(t) + q(t)u(t) + r(t)
% con u(a) = alpha
% y u'(a) = 0

Za = [alpha 0];
[T,Z] = rks4('fn1',a,b,Za,m);
U = Z(:,1)';

% resolver v' = p(t)v'(t) + q(t)v(t)
% con v(a) = 0
% y v'(a) = 1
Za = [0 1];
[T,Z] = rks4('fn2',a,b,Za,m);
V = Z(:,1)';

% Forma la combinación lineal para x(t).
X = U + (beta - U(m+1)) * V / V(m+1);
points = [T;X];

clc; figure(1); clf;

%~~~~~h
% inicio sección gráfica
%~~~~~
a = 0;
b = 4;
c = -1.25;
d = 1.75;
plot([a b],[0 0],'b',[0 0],[c d],'b');
axis([a b c d]);
axis(axis);
hold on;
plot(T,X,'-g');
if m<=30,
    plot(T,X,'or');
end;
xlabel('t');
ylabel('x');
Mx1 = 'La solución del disparo lineal a x` = f(t,x,x`).' ;
title(Mx1);
grid;
hold off;
figure(gcf);

clc;
%.....
% inicio sección impresión resultados.
%.....
Mx2 = '          t(k)          x(k)';
clc,...
disp(''),disp(Mx1),disp(''),disp(Mx2),disp(points')

```

```
function [T,Z]=rks4(F,a,b,Za, M)
```

```
%entrada    - F es el sistema ingresado como string 'F'
%            - a y b son los extremos del intervalo
%            - Za=[x(a) y(a)] condiciones iniciales
%            - M número de pasos
%salida     - T vector de pasos
%            - Z=[x1(t) . . . xn(t)] con xk(t) la aproximación a la
%              kth variable dependiente
```

```
h=(b-a)/M;
T=zeros(1,M+1);
Z=zeros(M+1,length(Za));
T=a:h:b;
Z(1,:)=Za;
```

```
for j=1:M
    k1=h*feval(F,T(j),Z(j,:));
    k2=h*feval(F,T(j)+h/2,Z(j,:)+k1/2);
    k3=h*feval(F,T(j)+h/2,Z(j,:)+k2/2);
    k4=h*feval(F,T(j)+h,Z(j,:)+k3);
    Z(j+1,:)=Z(j,:)+(k1+2*k2+2*k3+k4)/6;
end
```

Ejemplo de la ejecución del método del disparo lineal para resolver el PVF $x'' = 2t/(1+t^2) x' - 2/(1+t^2) + 1$ con $x(0) = 1.25$ y $x(4) = -0.95$.

```
octave:4> disparo_lineal
```

```
function W = fn1(t,Z)
x = Z(1); y = Z(2);
W = [y, (2*t*y/(1 + t^2) - 2*x/(1 + t^2) + 1)];
function W = fn2(t,Z)
x = Z(1); y = Z(2);
W = [y, (2*t*y/(1 + t^2) - 2*x/(1 + t^2))];
La solución del disparo lineal a  $x'' = f(t,x,x')$ .
```

t(k)	x(k)
0.00000	1.25000
0.20000	1.31731
0.40000	1.32643
0.60000	1.28165
0.80000	1.18928
1.00000	1.05673
1.20000	0.89191
1.40000	0.70276
1.60000	0.49699
1.80000	0.28198
2.00000	0.06473
2.20000	-0.14818
2.40000	-0.35052
2.60000	-0.53644
2.80000	-0.70043
3.00000	-0.83726
3.20000	-0.94201
3.40000	-1.01000
3.60000	-1.03678
3.80000	-1.01812
4.00000	-0.95000

```
% Método del Disparo no lineal  TECNICA DE DISPARO PARA EL PROBLEMA NO LINEAL

%
% Aproxima la solución del problema no lineal del valor de frontera
%
%      Y'' = F(X,Y,Y'), A<=X<=B, Y(A) = ALPHA, Y(B) = BETA:
%
%
% ENTRADA: Intervalo A,B; condiciones de contorno ALPHA, BETA; número de
% subintervalos N; tolerancia TOL; máximo número de iteraciones M.
% SALIDA: Aproximaciones W(1,I) a Y(X(I)); W(2,I) TO Y'(X(I))
% por cada I=0,1,...,N o un mensaje de que el número de iteraciones
% ha sido excedido.
% EJEMPLO
% Resolver por diferencias finitas la
% EDO con valor de frontera
% y ''= (32+2*x^3-y*y')/8, 1<=x<=3,
% y(1) = 17 , y(3) = 43 / 3 .
% Usando h = 0.5; 0.2; 0.05; 0.025.
% Solución: (La solución exacta es y(x) = x^2 +16 / x ).
TRUE = 1;
FALSE = 0;
fprintf(1,'este es el método de disparo no lineal.\n');
fprintf(1,'Ingrese la funcion F(X,Y,Z) en términos de x, y, z.\n');
fprintf(1,'seguida de la deriv. parcial de F respecto a y en \n');
fprintf(1,'la siguiente línea seguida de la de F respecto a \n');
fprintf(1,'z o y-prima en la línea siguiente. \n');
fprintf(1,'ejemplp:   (32+2*x^3-y*z)/8 \n'); %z=y' {f(x,y,y')}
fprintf(1,'           -z/8 \n'); %fy=-z/8 {df/dy}
fprintf(1,'           -y/8 \n'); %fyl=-y/8 {df/dy}
s = input('');
F = inline(vectorize(s),'x','y','z');
s = input('');
FY = inline(vectorize(s),'x','y','z');
s = input('');
FYP = inline(vectorize(s),'x','y','z');
OK=0;
while OK == 0
fprintf(1,'Ingrese extremos derecho e izquierdo en líneas separadas.\n');
A = input(' ');
B = input(' ');
if A >= B
fprintf(1,'extremo izquierdo debe ser menor que el derecho.\n');
else OK = 1;
end;
end;
fprintf(1,'Ingrese Y(%.10e).\n', A);
ALPHA = input(' ');
fprintf(1,'Ingrese Y(%.10e).\n', B);
BETA = input(' ');
TK = (BETA-ALPHA)/(B-A);
fprintf(1,'TK = %.8e\n', TK);
fprintf(1,'Ingresa nuevo TK? Entre Y o N.\n');
AA = input(' ','s');
if AA == 'Y' | AA == 'y'
fprintf(1,'ingrese nuevo TK\n');
TK = input(' ');
end;
OK = 0;
while OK == 0
fprintf(1,'Ingrese un entero > 1 par el número de subintervalos.\n');
```

```

N = input(' ');
if N <= 1
    fprintf(1, 'Número debe ser mayor que 1.\n');
else
    OK = 1;
end;
end;
OK = 0;
while OK == 0
    fprintf(1, 'Ingrese Tolerancia.\n');
    TOL = input(' ');
    if TOL <= 0
        fprintf(1, 'Tolerancia debe ser positiva.\n');
    else
        OK = 1;
    end;
end;
OK = 0;
while OK == 0
    fprintf(1, 'Ingrese número máximo de iteraciones.\n');
    NN = input(' ');
    if NN <= 0
        fprintf(1, 'debe ser entero positivo .\n');
    else
        OK = 1;
    end;
end;
if OK == 1
    FLAG = 1;
    if FLAG == 2
        fprintf(1, 'Ingrese nombrearchivo en la forma - drive:\\name.ext\n');
        fprintf(1, 'ejemplo A:\\OUTPUT.DTA\n');
        NAME = input(' ', 's');
        OUP = fopen(NAME, 'wt');
    else
        OUP = 1;
    end;
    fprintf(OUP, 'METODO DISPARO NO LINEAL\n\n');
    fprintf(OUP, '  I      X(I)          W1(I)          W2(I)\n');
% STEP 1
    W1 = zeros(1, N+1);
    W2 = zeros(1, N+1);
    H = (B-A)/N;
    K = 1;
% TK CALCULADA
    OK = 0;
% etapa 2
    while K <= NN & OK == 0
% etapa 3
        W1(1) = ALPHA;
        W2(1) = TK;
        U1 = 0 ;
        U2 = 1;
% etapa 4
% Método Runge-Kutta para sistemas se usa en etapas 5 y 6
        for I = 1 : N
% etapa 5
            X = A+(I-1)*H;
            T = X+0.5*H;
% etapa 6
            K11 = H*W2(I);
            K12 = H*F(X, W1(I), W2(I));

```

```

K21 = H*(W2(I)+0.5*K12);
K22 = H*(F(T,W1(I)+0.5*K11,W2(I)+0.5*K12));
K31 = H*(W2(I)+0.5*K22);
K32 = H*(F(T,W1(I)+0.5*K21,W2(I)+0.5*K22));
K41 = H*(W2(I)+K32);
K42 = H*(F(X+H,W1(I)+K31,W2(I)+K32));
W1(I+1) = W1(I)+(K11+2*(K21+K31)+K41)/6;
W2(I+1) = W2(I)+(K12+2*(K22+K32)+K42)/6;
K11 = H*U2;
K12 = H*(FY(X,W1(I),W2(I))*U1+FYP(X,W1(I),W2(I))*U2);
K21 = H*(U2+0.5*K12);
K22 = H*(FY(T,W1(I),W2(I))*(U1+0.5*K11)+FYP(T,W1(I),W2(I))*(U2+0.5*K21));
K31 = H*(U2+0.5*K22);
K32 = H*(FY(T,W1(I),W2(I))*(U1+0.5*K21)+FYP(T,W1(I),W2(I))*(U2+0.5*K22));
K41 = H*(U2+K32);
K42 = H*(FY(X+H,W1(I),W2(I))*(U1+K31)+FYP(X+H,W1(I),W2(I))*(U2+K32));
U1 = U1+(K11+2*(K21+K31)+K41)/6;
U2 = U2+(K12+2*(K22+K32)+K42)/6;
end;
% etapa 7
% prueba de exactitud
if abs(W1(N+1)-BETA) < TOL
% etapa 8
I = 0;
fprintf(OUP, '%3d %13.8f %13.8f %13.8f\n', I, A, ALPHA, TK);
for I = 1 : N
J = I+1;
X = A+I*H;
fprintf(OUP, '%3d %13.8f %13.8f %13.8f\n', I, X, W1(J), W2(J));
end;
fprintf(OUP, 'Convergencia en %d iteraciones\n', K);
fprintf(OUP, ' t = %14.7e\n', TK);
% etapa 9
OK = TRUE;
else
% etapa 10
% SE aplica método de Newton para mejorar TK
TK = TK-(W1(N+1)-BETA)/U1;
K = K+1;
end;
end;
% etapa 11
% método falló
if OK == 0
fprintf(OUP, 'Método falló luego de %d iteraciones\n', NN);
end;
end;
if OUP ~= 1
fclose(OUP);
fprintf(1, 'archivo de salida %s creado con éxito \n', NAME);
end;

```

```
%Entrada - f1,f2,f3,f4 son las funciones de contorno ingresadas como string
%          - a y b extremos derechos como [0,a] y [0,b]
%          - h tamaño de paso
%          - tol es la tolerancia
%SALIDA - U la matriz solución del problema

% EJEMPLO DE CARGA
% f1=@(x)0*x+10
% f2=@(x)0*x+100
% f3=@(x)0*x+50
% f4=@(x)0*x
% U=dirich(f1,f2,f3,f4,4,4,0.5,0.001,20)

%Inicializa parámetros y U

n=fix(a/h)+1;
m=fix(b/h)+1;
ave=(a*(feval(f1,0)+feval(f2,0)) ...
    +b*(feval(f3,0)+feval(f4,0)))/(2*a+2*b);
U=ave*ones(n,m);

%Condiciones de borde

U(1,1:m)=feval(f3,0:h:(m-1)*h)';
U(n,1:m)=feval(f4,0:h:(m-1)*h)';
U(1,n,1)=feval(f1,0:h:(n-1)*h);
U(1:n,m)=feval(f2,0:h:(n-1)*h);
U(1,1)=(U(1,2)+U(2,1))/2;
U(1,m)=(U(1,m-1)+U(2,m))/2;
U(n,1)=(U(n-1,1)+U(n,2))/2;
U(n,m)=(U(n-1,m)+U(n,m-1))/2;

%parametros SOR

w=4/(2+sqrt(4-(cos(pi/(n-1))+cos(pi/(m-1)))^2));

%Refinando aproximaciones
err=1;
cnt=0;
while((err>tol)&(cnt<=max1))
    err=0;
    for j=2:m-1
        for i=2:n-1
            relx=w*(U(i,j+1)+U(i,j-1)+U(i+1,j)+U(i-1,j)-4*U(i,j))/4;
            U(i,j)=U(i,j)+relx;
            if (err<=abs(relx))
                err=abs(relx);
            end
        end
    end
    cnt=cnt+1;
end

U=flipud(U');
```

Se plantea la ecuación de Laplace, $\text{div}(\text{div})=0$ en $R=\{(x,y):0\leq x\leq 4, 0\leq y\leq 4\}$ con las CF dadas por:

$u(x,0)=10$ y $u(x,4)=100$ para $0<x<4$ y

$u(0,y)=50$ y $u(4,y)=0$ para $0<y<4$

se toma $h=0.5$ para x e y , o sea 64 cuadrados en la malla

```
octave:13> f1=@(x)0*x+10;
octave:14> f2=@(x)0*x+100;
octave:15> f3=@(x)0*x+50;
octave:16> f4=@(x)0*x;
octave:17> U=dirich(f1,f2,f3,f4,4,4,0.5,0.001,20)
```


U =

Columns 1 through 6:

75.00000	100.00000	100.00000	100.00000	100.00000	100.00000
50.00000	72.56216	79.88936	81.64786	80.52445	76.68815
50.00000	60.35926	65.34734	66.17761	63.76179	57.91969
50.00000	53.52760	54.96319	53.95345	50.42541	43.98667
50.00000	48.78785	47.02429	44.24762	39.99980	33.76675
50.00000	44.59979	40.09850	36.01288	31.55942	26.04620
50.00000	39.51325	32.75749	28.14616	24.17901	19.88825
50.00000	30.69626	23.27224	19.63521	17.12241	14.67547
30.00000	10.00000	10.00000	10.00000	10.00000	10.00000

Columns 7 through 9:

100.00000	100.00000	50.00000
68.30843	49.30347	0.00000
47.24210	28.90547	0.00000
33.83481	19.07627	0.00000
25.03420	13.56482	0.00000
18.97039	10.14874	0.00000
14.65245	8.05973	0.00000
11.69142	7.43778	0.00000
10.00000	10.00000	5.00000

`function U=crnich(f,c1,c2,a,b,c,n,m)`  [ECUACIÓN PARABÓLICA](#)

```
%Entrada - f=u(x,0) 'f' como un string
%          - c1=u(0,t) y c2=u(a,t)
%          - a y b extremos derechos como [0,a] y [0,b]
%          - c es la constante en la ecuación de calor

%          - n y m numeros de puntos de malla [0,a] y [0,b]
%SALIDA - U la matriz solución del problema
```

```
% Ejemplo de carga
% f=@(x)sin(pi*x)+sin(3*pi*x)
% crnich(f,0,0,1,0.5,1,11,11)
```

```
%Inicializa parámetros y U
```

```
h=a/(n-1);
k=b/(m-1);
r=c^2*k/h^2;
s1=2+2/r;
s2=2/r-2;
```



```
U=zeros (n,m) ;

%Condiciones de borde

U (1,1:m)=c1;
U (n,1:m)=c2;

%Generación de la primera fila

U (2:n-1,1)=feval (f,h:h:(n-2)*h)';

%Se forma la diagonal y los elementos no diagonales de A
%el vector constante B y resuelve el sistema tridiagonal AX=B

Vd (1,1:n)=s1*ones (1,n) ;
Vd (1)=1;
Vd (n)=1;
Va=-ones (1,n-1) ;
Va (n-1)=0;
Vc=-ones (1,n-1) ;
Vc (1)=0;
Vb (1)=c1;
Vb (n)=c2;
for j=2:m
    for i=2:n-1
        Vb (i)=U (i-1,j-1)+U (i+1,j-1)+s2*U (i,j-1);
    end
    X=trisys (Va,Vd,Vc,Vb);
    U (1:n,j)=X';
end

U=U'

function X=trisys (A,D,C,B)
%Entradas - A es la subdiagonal de la matriz de coeficientes
%          - D es la diagonal principal de la matriz de coeficientes
%          - C es la diagonal superior de la matriz de coeficientes
%          - B es el vector constante del sistema lineal
%SALIDA   - X es el vector solución

N=length (B) ;

for k=2:N
    mult=A (k-1) /D (k-1) ;
    D (k)=D (k) -mult*C (k-1) ;
    B (k)=B (k) -mult*B (k-1) ;
end

X (N)=B (N) /D (N) ;
for k= N-1:-1:1
    X (k)= (B (k) -C (k) *X (k+1) ) /D (k) ;
end
```

Ecuación de calor

```
(∂u(x,t)/∂t)=c²(∂²u(x,t)/∂x²) en 0<x<1 y 0<t<0.1
u(0,t)=0,u(1,t)=0 son las CF para 0≤t≤0.1
u(x,0)=sin(πx)+sin(3πx) a t=0 y 0≤x≤1
h=0.1,k=0.01,r=1, generando n=11 y m=11
octave:25> f=@(x)1.5-1.5*x;
octave:26> g=@(x)0*x;a=1;b=0.5;c=2;n=10;m=10
m = 10
octave:27> f=@(x)sin(pi*x)+sin(3*pi*x);c1=0;c2=0;c=1;n=11;m=11;
octave:28> forwddif(f,c1,c2,a,b,c,n,m)
ans =
Columns 1 through 6:
0.0000e+000 1.1180e+000 1.5388e+000 1.1180e+000 3.6327e-001 0.0000e+000
0.0000e+000 -2.3681e+000 -2.6692e+000 -5.5174e-001 2.3207e+000 3.6327e+000
0.0000e+000 7.9667e+000 9.4239e+000 3.2231e+000 -5.4817e+000 -9.4871e+000
0.0000e+000 -2.4581e+001 -2.8866e+001 -9.2970e+000 1.8015e+001 3.0567e+001
0.0000e+000 7.6894e+001 9.0409e+001 2.9418e+001 -5.5787e+001 -9.4952e+001
0.0000e+000 -2.4000e+002 -2.8213e+002 -9.1647e+001 1.7441e+002 2.9670e+002
0.0000e+000 7.4934e+002 8.8091e+002 2.8624e+002 -5.4441e+002 -9.2622e+002
0.0000e+000 -2.3395e+003 -2.7503e+003 -8.9362e+002 1.6998e+003 2.8918e+003
0.0000e+000 7.3044e+003 8.5868e+003 2.7900e+003 -5.3070e+003 -9.0287e+003
0.0000e+000 -2.2805e+004 -2.6809e+004 -8.7109e+003 1.6569e+004 2.8189e+004
0.0000e+000 7.1202e+004 8.3703e+004 2.7197e+004 -5.1731e+004 -8.8010e+004
Columns 7 through 11:
3.6327e-001 1.1180e+000 1.5388e+000 1.1180e+000 0.0000e+000
2.3207e+000 -5.5174e-001 -2.6692e+000 -2.3681e+000 0.0000e+000
-5.4817e+000 3.2231e+000 9.4239e+000 7.9667e+000 0.0000e+000
1.8015e+001 -9.2970e+000 -2.8866e+001 -2.4581e+001 0.0000e+000
-5.5787e+001 2.9418e+001 9.0409e+001 7.6894e+001 0.0000e+000
1.7441e+002 -9.1647e+001 -2.8213e+002 -2.4000e+002 0.0000e+000
-5.4441e+002 2.8624e+002 8.8091e+002 7.4934e+002 0.0000e+000
1.6998e+003 -8.9362e+002 -2.7503e+003 -2.3395e+003 0.0000e+000
-5.3070e+003 2.7900e+003 8.5868e+003 7.3044e+003 0.0000e+000
1.6569e+004 -8.7109e+003 -2.6809e+004 -2.2805e+004 0.0000e+000
-5.1731e+004 2.7197e+004 8.3703e+004 7.1202e+004 0.0000e+000
```

Otro ejemplo:

```
octave:20> f=@(x)sin(pi*x)+sin(3*pi*x);
octave:21> U=crnich(f,0,0,1,0.5,1,11,11)
U =
Columns 1 through 7:
0.00000 1.11803 1.53884 1.11803 0.36327 0.00000 0.36327
0.00000 -0.09292 0.02699 0.38379 0.78084 0.95342 0.78084
0.00000 0.21099 0.33069 0.33501 0.27955 0.24804 0.27955
0.00000 0.03534 0.09171 0.16788 0.23696 0.26507 0.23696
0.00000 0.05357 0.09342 0.11414 0.12045 0.12113 0.12045
0.00000 0.02137 0.04359 0.06500 0.08118 0.08727 0.08118
0.00000 0.01683 0.03099 0.04092 0.04645 0.04818 0.04645
0.00000 0.00887 0.01723 0.02432 0.02916 0.03089 0.02916
0.00000 0.00585 0.01100 0.01493 0.01736 0.01817 0.01736
0.00000 0.00339 0.00649 0.00900 0.01065 0.01122 0.01065
0.00000 0.00211 0.00400 0.00548 0.00642 0.00674 0.00642
Columns 8 through 11:
1.11803 1.53884 1.11803 0.00000
0.38379 0.02699 -0.09292 0.00000
0.33501 0.33069 0.21099 0.00000
0.16788 0.09171 0.03534 0.00000
0.11414 0.09342 0.05357 0.00000
0.06500 0.04359 0.02137 0.00000
```

```

0.04092 0.03099 0.01683 0.00000
0.02432 0.01723 0.00887 0.00000
0.01493 0.01100 0.00585 0.00000
0.00900 0.00649 0.00339 0.00000
0.00548 0.00400 0.00211 0.00000
U =
Columns 1 through 7:
0.00000 1.11803 1.53884 1.11803 0.36327 0.00000 0.36327
0.00000 -0.09292 0.02699 0.38379 0.78084 0.95342 0.78084
0.00000 0.21099 0.33069 0.33501 0.27955 0.24804 0.27955
0.00000 0.03534 0.09171 0.16788 0.23696 0.26507 0.23696
0.00000 0.05357 0.09342 0.11414 0.12045 0.12113 0.12045
0.00000 0.02137 0.04359 0.06500 0.08118 0.08727 0.08118
0.00000 0.01683 0.03099 0.04092 0.04645 0.04818 0.04645
0.00000 0.00887 0.01723 0.02432 0.02916 0.03089 0.02916
0.00000 0.00585 0.01100 0.01493 0.01736 0.01817 0.01736
0.00000 0.00339 0.00649 0.00900 0.01065 0.01122 0.01065
0.00000 0.00211 0.00400 0.00548 0.00642 0.00674 0.00642
Columns 8 through 11:
1.11803 1.53884 1.11803 0.00000
0.38379 0.02699 -0.09292 0.00000
0.33501 0.33069 0.21099 0.00000
0.16788 0.09171 0.03534 0.00000
0.11414 0.09342 0.05357 0.00000
0.06500 0.04359 0.02137 0.00000
0.04092 0.03099 0.01683 0.00000
0.02432 0.01723 0.00887 0.00000
0.01493 0.01100 0.00585 0.00000
0.00900 0.00649 0.00339 0.00000
0.00548 0.00400 0.00211 0.00000

```

`function F=finedif(f,g,a,b,c,n,m)`  [ECUACIÓN HIPERBÓLICA](#)

```

%Entrada - f*u(x,0) como un string
%          g*ut(x,0) como un string
%          - a y b extremos derechos como [0,a] y [0,b]
%          - c es la constant de la ecuación de onda
%          - n y m numeros de puntos de malla [0,a] y [0,b]
%SALIDA - U la matriz solución del problema

%Inicializa parámetros y U
h=a/(n-1);
k=b/(m-1);
r=c+k/h;
r2=r^2;
r22=r^2/2;
s1=1-r^2;
s2=2-2*r^2;
U=zeros(n,m);

%Calcular la primera y segunda fila
for i=2:(n-1)
    U(i,1) = feval(f,h*(i-1));

```

```

U(i,2) = s1*feval(f,h*(i-1)) + k*feval(g,h*(i-1)) ...
        + r22*(feval(f,h*(i)) + feval(f,h*(i-2)));
end

%Calcular las filas restantes de U
for j=3:m
    for i=2:(n-1),
        U(i,j) = s2*U(i,j-1) + r2*(U(i-1,j-1) + U(i+1,j-1)) - U(i,j-2);
    end
end
U=U';

```

Ecuación de onda

$(\partial^2 u(x,t)/\partial t^2) = c^2 (\partial^2 u(x,t)/\partial x^2)$ en $0 < x < 1$ y $0 < t < 0.5$
 $u(0,t)=0, u(1,t)=0$ son las CF
 $u(x,0) = u(x,0) = 1.5 - 1.5x$ en $0 \leq x \leq 1$
 $u_t(x,0) = 0$ en $0 < x < 1$

```

octave:22> f=@(x)1.5-1.5*x;
octave:23> g=@(x)0*x;a=1;b=0.5;c=2;n=10;m=10;
octave:24> finedif(f,g,a,b,c,n,m)
ans =
Columns 1 through 7:
0.00000  1.33333  1.16667  1.00000  0.83333  0.66667  0.50000
0.00000  1.33333  1.16667  1.00000  0.83333  0.66667  0.50000
0.00000 -0.16667  1.16667  1.00000  0.83333  0.66667  0.50000
0.00000 -0.16667 -0.33333  1.00000  0.83333  0.66667  0.50000
0.00000 -0.16667 -0.33333 -0.50000  0.83333  0.66667  0.50000
0.00000 -0.16667 -0.33333 -0.50000 -0.66667  0.66667  0.50000
0.00000 -0.16667 -0.33333 -0.50000 -0.66667 -0.83333  0.50000
0.00000 -0.16667 -0.33333 -0.50000 -0.66667 -0.83333 -1.00000
0.00000 -0.16667 -0.33333 -0.50000 -0.66667 -0.83333 -1.00000
0.00000 -0.16667 -0.33333 -0.50000 -0.66667 -0.83333 -1.00000
Columns 8 through 10:
0.33333  0.16667  0.00000
0.33333  0.16667  0.00000
0.33333  0.16667  0.00000
0.33333  0.16667  0.00000
0.33333  0.16667  0.00000
0.33333  0.16667  0.00000
0.33333  0.16667  0.00000
-1.16667  0.16667  0.00000
-1.16667 -1.33333  0.00000

```