Diferencias finitas como aproximaciones de derivadas parciales

tu[:, : 2]

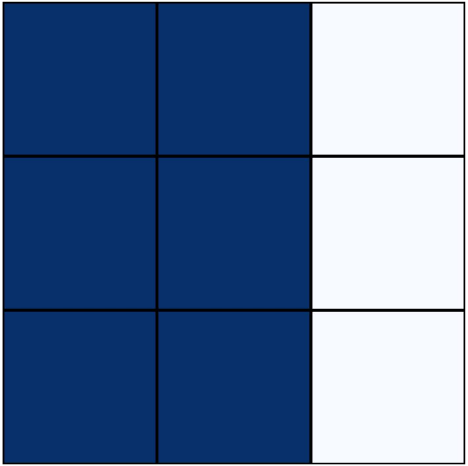


Figura: Seleccione todas las filas de u y desde la primera columna hasta la segundo.

tu[1 : 2, : 2]

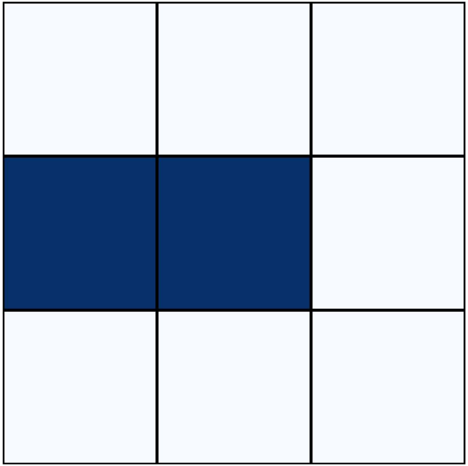


Figura: Seleccione de la segunda fila a la segunda y de la primera columna a la segunda.

Estimar el error para $\partial^{+} u(x)$

Por la expansión de Taylor, para algún $\xi \in [x, x + h]$

$$\partial_x u(x) = \partial^{+} u(x) - \frac{\Delta x}{2} \partial_x^2 u(\xi). \tag{4}$$

Para $u(x) = \operatorname{sen}(x)$ en $x = \frac{5\pi}{9}$

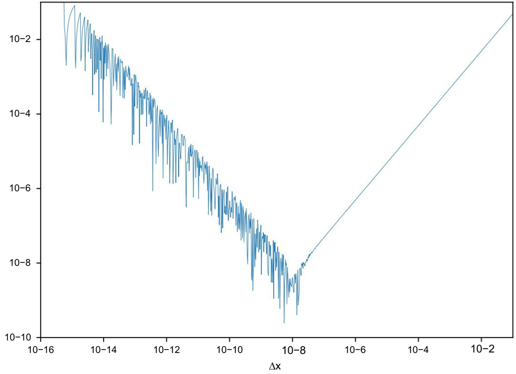


Figura: Error de $\partial^{+} u(x)$ para varios valores de Δx Python .

Parte del error se debe a las imprecisiones en (4) es

$$\text{error truncado} = \frac{\Delta x}{2} \partial_x^2 u(\xi).$$

Depende de ε_{mach} (para float64 es $2{,}22044604925 \times 10^{-16}$).

$$\text{error redondeado} \approx \frac{|u(x)|\, \varepsilon_{máquina}}{\Delta x} + \partial_x u(x)\, \varepsilon_{máquina}.$$

El mejor valor de Δx se obtiene minimizando el error total en función de Δx, es decir,

$$0 = \frac{1}{2} \partial_x^2 u(\xi) - \frac{|u(x)|\, \varepsilon_{mach}}{(\Delta x)^2}.$$

$$\Delta x = \sqrt{\frac{2\, |u(x)|\, \varepsilon_{mach}}{|\partial_x^2 u(\xi)|}}.$$

Si $u(x)$ y $\partial_x^2 u(\xi)$ no son ni grandes ni pequeños, entonces

$\Delta x \approx \sqrt{\varepsilon_{mach}}$ (para float64 es $1{,}49011611938 \times 10^{-8}$).

$$u(x + \Delta x) = u(x) + \underbrace{\Delta x \partial_x u(x)}_{\text{Aproximación de primer orden}} + \underbrace{\frac{(\Delta x)^2}{2!} \partial_x^2 u(x)}_{\text{error de truncamiento}} + \cdots .$$

$$u(x + \Delta x) = \text{aproximación} + E(\Delta x) + \text{términos de orden superior.}$$

$$u(x + \Delta x) \approx u(x) + \Delta x \partial_x u.$$

$$\text{es}(\Delta x) = \frac{(\Delta x)^{n+1}}{(n+1)!} \partial^{n+1} \text{tu}(x).$$

```
importar numpy como np

x = 0
Δx = 0,1
u = np.cos

definición dudx(x):
        devuelve -np.sin(x)

definición du2dx(x):
        devuelve -np.cos(x)

exacto = u(x + Δx)
aproximación = u(x) + Δx * dudx(x)
término_omitido_de_orden_inferior_absoluto = np.abs(np.power(Δx, 2) / 2 * du2dx(x))
truncamiento_error = np.abs(exacto - aproximación)
error_atribuido_a_otros_términos_omitidos = np.abs(
        error_de_truncamiento : término_omitido_orden_inferior_absoluto
)
```

```
Aproximación de primer orden: 1,0
Exacto: 0,9950041652780258
Error de truncamiento: 0,0049958347219741794
Término omitido de orden inferior absoluto: 0,005000000000000001
Error atribuido a otros términos: 4.165278025821534e-06
```

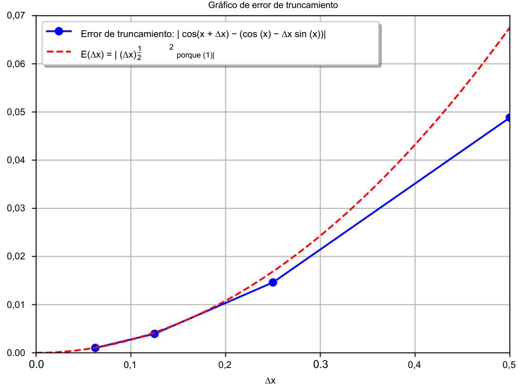


Figura: Error de truncamiento para varios valores de Δx en x = 1 Python .

Diferencias finitas como aproximaciones de derivadas parciales

Dado un error de truncamiento conocido E (Δx) = O ((Δx) norte) es
Es posible estimar el orden de una aproximación.

E (Δx) ≈ C(Δx) norte .

Tomando logaritmos de ambos lados se obtiene

iniciar sesión |E (Δx)| = norte log (Δx) + log (C) .

que es una función lineal en Δx. Por lo tanto, una forma de aproximar la pendiente n como un gradiente es

$$\boxed{n \approx \frac{\log |E\,(\text{máx}\,(\Delta x))| - \log |E\,(\text{mín}\,(\Delta x))|}{\log\,(\text{máx}\,(\Delta x)) - \log\,(\text{mín}\,(\Delta x))}\,.}$$

```
importar numpy como np
desde jaxtyping importar Array, Float

u = np.cos

definición dudx(x):
    devuelve -np.sin(x)

x = np.pi / 4
Δx = np.logspace(inicio=-3, fin=0, num=4, base=2) / 10

hacia atrás: Float[Array, "dim1"] = (u(x) - u(x - Δx)) / Δx
centrado: Float[Array, "dim1"] = (u(x + Δx) - u(x - Δx)) / (2 * Δx)
adelante: Float[Array, "dim1"] = (u(x + Δx) - u(x)) / Δx

error_backward: Float[Array, "dim1"] = np.abs(dudx(x) - hacia atrás)
error_centered: Float[Array, "dim1"] = np.abs(dudx(x) - centrado)
error_adelante = np.abs(dudx(x) - adelante)

def estimar_orden(
    Δx: Flotante[Matriz, "dim1"], error_de_truncamiento: Flotante[Matriz, "dim1"]
) ->- flotante:
    afirmar Δx.size === error_de_truncamiento.size

    E_max = error_de_truncamiento[np.argmax(a=Δx)]
    E_min = error_de_truncamiento[np.argmin(a=Δx)]

    retorno (np.log(np.abs(E_max)) - np.log(np.abs(E_min))) / (
        np.log(Δx.max()) - np.log(Δx.min())
    )

print(f"El orden de convergencia hacia atrás es {estimate_order(Δx, error_backward)}")
print(f"El orden de convergencia del centrado es {estimate_order(Δx, error_centered)}")
print(f"El orden de convergencia de avance es {estimate_order(Δx, error_forward)}")
```

Programa Python : Determinar el orden de convergencia Python .

```
      Δx     hacia atrás    centrado      adelante
 --------
0,1          -0,670603 -0,705929 -0,741255
0,05         -0,689138 -0,706812 -0,724486
0,025        -0,698195 -0,707033 -0,715872
0,0125 -0,702669 -0,707088 -0,711508

      Δx     hacia atrás    centrado      adelante
 --------
0,1         0,0365038 0,00117792 0,034148
0,05        0,0179686 0,000294591 0,0173794
0,025 0,00891203 7,36547e-05 0,00876472
0,0125 0,00443777 1,84141e-05 0,00440095

El orden de convergencia hacia atrás es 1.013379633444132
El orden de convergencia del centrado es 1,9997633049971728
El orden de convergencia hacia adelante es 0,9853046896368071
```

```
desde escribir importar Llamable

importar numpy como np

def u(x: flotante) ->- flotante:
    """Función de muestra
    u:     --->-
    """x |--->- exp(x^2)

    devuelve np.exp(np.pow(x, 2))

def up(x: float) ->- float:
    """Derivada de la función de muestra u':    --->-     x
    |--->- 2*x*f(x)

    """

    devuelve 2 * x * u(x)

def ff(u: Callable, Δx: np.array) ->- np.array: """Aproximación de
    diferencia finita hacia adelante u' = (u(x + Δx) - u(x)) / Δx

    """

    devuelve (u(x + Δx) - u(x)) / Δx

def bf(u: Invocable, Δx: np.array) ->- np.array:
    """Aproximación de diferencias finitas hacia atrás u' = (u(x) -
    u(x - Δx)) / Δx
    """

    devuelve (u(x) - u(x - Δx)) / Δx

def cf(u: Invocable, Δx: np.array) ->- np.array:
    """Aproximación de diferencias finitas centradas u' = (u(x +
    Δx / 2) - u(x - Δx / 2)) / Δx
    """

    devuelve (u(x + Δx / 2) - u(x - Δx / 2)) / Δx

x = 2
exacto = arriba(x)
Δx = np.logspace(inicio=-16, fin=-1.0, num=16)
```

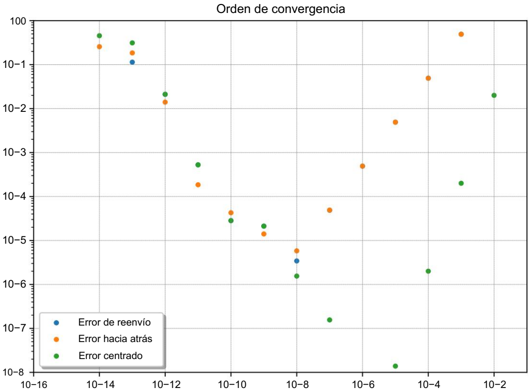


Figura: $\partial_+ u(x)$, $\partial_- u(x)$, $\partial_\circ u(x)$ de $u(x) = \exp x^2$ para varios valores de Δx en $x = 2$ Python .

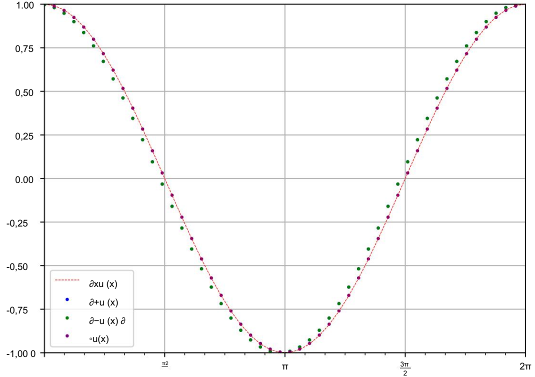
Diferencias finitas como aproximaciones de derivadas parciales

En el cálculo de la matriz, no consideramos el último elemento de la diferencia ui+1 − ui. y el primer elemento de la diferencia ui+1 − ui.

$$+u\ (x) = \frac{ui+1 - ui\ \partial}{\Delta x}, \quad i = 0, \ldots, n-1.$$

$$\partial -u\ (x) = \frac{\text{interfaz de usuario − interfaz de usuario −1}}{\Delta x}, \quad \text{yo} = 1, \ldots, \text{norte.}$$

Gráfica de la derivada de una función




```
importar numpy como np

x, Δx = np.linspace(inicio=0, fin=2 * np.pi, retstep=True) y = np.sin(x) hacia
adelante =

(np.roll(y, -1) - y)[:-1] / Δx hacia atrás = (y - np.roll(y, 1))
[1:] / Δx centrado = (np.roll(y, -1) - np.roll(y, 1))[1:-1] /
(2 * Δx) primera_derivada = np.cos(x) np.roll(a = u, desplazamiento = −1), u, np.roll(a
= u, desplazamiento = −1) − u
```

tu[1]	tu[2]	tu[3]	tu[4]	tu[0]
tu[0]	tu[1]	tu[2]	tu[3]	tu[4]
u[1] − u[0]	u[2] − u[1]	u[3] − u[2]	u[4] − u[3]	u[0] − u[4]

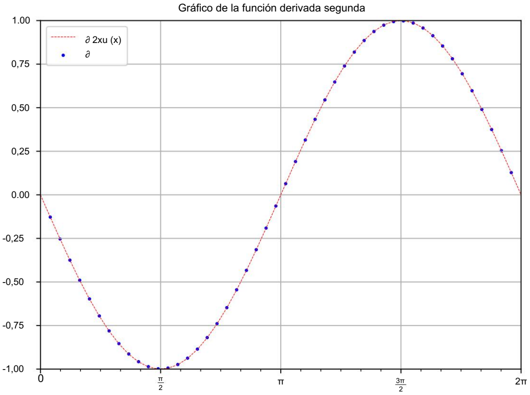
np.roll(a = u, desplazamiento = 1), u, u − np.roll(a = u, desplazamiento = 1)

tu[4]	tu[0]	tu[1]	tu[2]	tu[3]
tu[0]	tu[1]	tu[2]	tu[3]	tu[4]
u[4] − u[0]	u[0] − u[1]	u[1] − u[2]	u[2] − u[3]	u[3] − u[4]

Diferencias finitas como aproximaciones de derivadas parciales

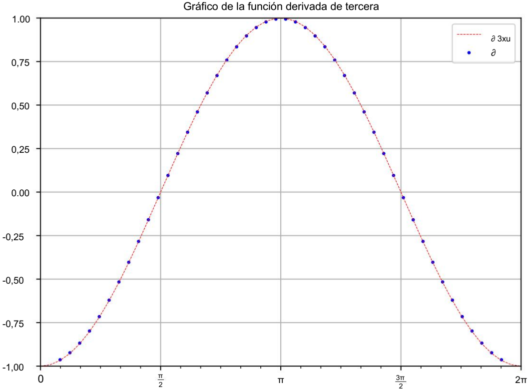
$$\partial + \partial - u\ (x) = \frac{ui+1 - 2ui + ui-1}{(\Delta x)^2}, \quad i = 1, \ldots, n - 1.$$

```
importar numpy como np

x, Δx = np.linspace(inicio=0, fin=2 * np.pi, retstep=Verdadero)
y = np.sin(x)

plt.ylim(-1, 1)
plt.cuadrícula()
```



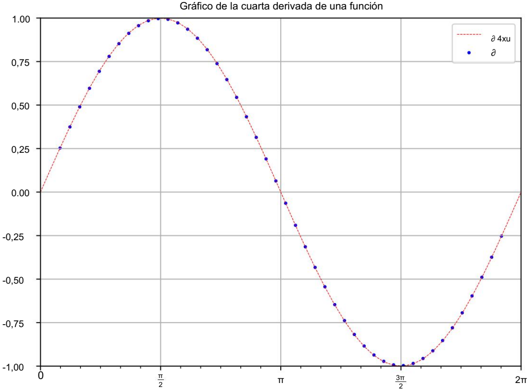
$$= \frac{u_{i+2} - 2u_{i+1} + 2u_{i-1} - u_{i-2}}{2(\Delta x)^3}, \quad i = 2, \ldots, n-2.$$

```
importar numpy como np

x, Δx = np.linspace(inicio=0, fin=2 * np.pi, retstep=Verdadero)
y = np.sin(x)

plt.gca().xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
plt.gca().xaxis.set_major_formatter(plt.FuncFormatter(formateador_múltiple()))
plt.scatter(x=x[1:-1], y=partial2x, c="azul", s=3, etiqueta=r"$\partial$")
plt.title(label="Gráfico de la segunda derivada de la función")
```



$$= \frac{u_{i+2} - 4u_{i+1} + 6u_i - 4u_{i-1} + u_{i-2}}{(\Delta x)^4}, \quad i = 2, \ldots, n-2.$$

```
importar numpy como np

x, Δx = np.linspace(inicio=0, fin=2 * np.pi, retstep=Verdadero)
y = np.sin(x)

plt.gca().xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
plt.gca().xaxis.set_major_formatter(plt.FuncFormatter(formateador_múltiple()))
plt.scatter(x=x[2:-2], y=partial3x, c="azul", s=3, etiqueta=r"$\partial$")
plt.title(label="Gráfico de la función derivada de tercera")
```



Método de pasos complejos

Definición (Función holomorfa)

Sea D C una región abierta simplemente conexa y u: D → C. Decimos que u es compleja diferenciable en a D si y solo si

$$\lim_{z \to a} \frac{u(z) - u(a)}{z - a}$$

existe. Si u es compleja diferenciable en cada punto de D, entonces decimos que u es holomorfo en D.

La aproximación de derivada escalonada compleja es una técnica para calcular la derivada de una función de valor real u (x). Para u analítico,

$$u(x + i\Delta x) = u(x) + i\Delta x \partial_x u(x) + \frac{(i\Delta x)^2}{2!} \partial_x^2 u(x) + \cdots .$$

$$\mathrm{Re}[u(x + i\Delta x)] + i\,\mathrm{Im}[u(x + i\Delta x)] \approx u(x) + i\Delta x \partial_x u(x).$$

Comparando partes imaginarias de los dos lados se obtiene

$$\partial_x u(x) \approx \mathrm{Soy} \frac{u(x + i\Delta x)}{\Delta x} .$$

Observación

Entre bastidores, el método de pasos complejos es un caso particular de diferenciación automática.

Método de pasos complejos

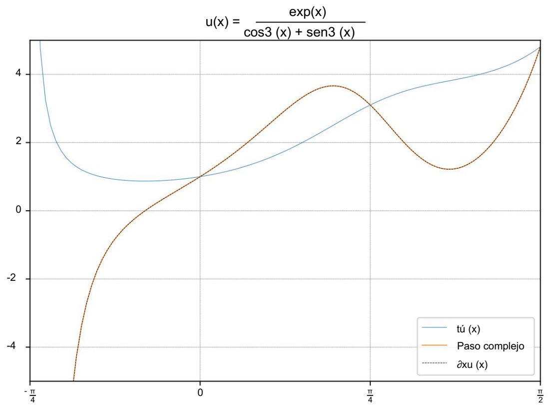
$$u(x + i\Delta x) = u(x) + i\Delta x \partial_x u(x) + \frac{(i\Delta x)^2}{2!} \partial_x^2 u(x) + \cdots .$$

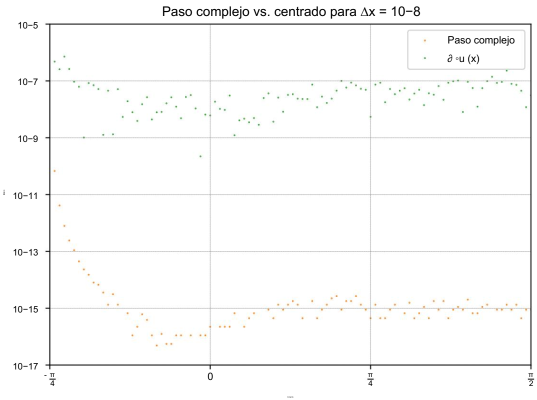
$$\operatorname{Re}[u(x + i\Delta x)] + i\operatorname{Im}[u(x + i\Delta x)] \approx u(x) + i\Delta x \partial_x u(x).$$

Comparando partes reales de los dos lados se obtiene

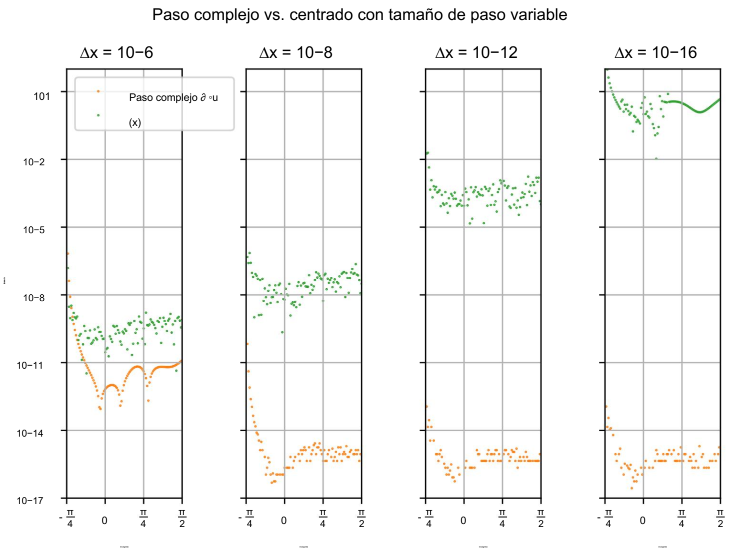
$$\partial_x^2 u(x) \approx \frac{2}{\Delta x^2}\left(u(x) - \operatorname{Re}[u(x + i\Delta x)]\right).$$

Método de pasos complejos





Método de pasos complejos



Resolver BVP para EDO

Sigamos los pasos:

1. Discretice el dominio en el que se define la ecuación.
2. En cada punto de la cuadrícula, reemplace las derivadas con una aproximación, utilizando los valores en los puntos de la cuadrícula vecinos.
3. Reemplace las soluciones exactas por sus aproximaciones.
4. Resuelva el sistema de ecuaciones resultante.

Diferencia finita para BVP de dos puntos

Primero veremos cómo encontrar aproximaciones a la derivada de una función, y luego cómo se pueden usar para resolver
Problemas de valores límite como

$$\frac{\text{para ti}}{dx2} + p(x) \frac{\text{tú}}{Dx} + q(x)\, u = r(x) \text{ para } a \leq x \leq b.$$

$$u(a) = ua,\ u(b) = ub$$

Esta técnica descrita aquí es aplicable a varias otras PDE dependientes del tiempo y, por lo tanto, es importante intentar
Entender la idea subyacente.

Ejemplo (FDM BVP de dos puntos para el problema de Poisson 1D)

Sea $f : [0, 1] \to R$ una función. Halla una $u: [0, 1] \to R$ tal que

(5)

$$- \frac{\text{para ti}}{u(0)} = f(x)\text{ , dx2} \qquad x \qquad (0, 1)\ .$$

$$= ua,\ u(1) = ub.$$

En lugar de intentar calcular u (x) con exactitud, ahora intentaremos calcular una aproximación numérica u_Δ de u (x). Como muchas veces antes, comenzamos definiendo n + 1 puntos igualmente espaciados $\{x_i\}_{yo=0}^{norte}$ con un tamaño de cuadrícula $h = \frac{b-a}{norte}$ de modo que

$$i = 0, 1, \ldots, n : x_i := a + ih.$$

Consideremos una colección de puntos espaciados de manera uniforme, etiquetados con un índice i, con el espaciamiento físico entre ellos denotado como Δx. Podemos expresar la primera derivada de una cantidad a en i como:

$$\frac{\partial a}{\partial x}\Big|_{yo} \approx \frac{a_i - a_{i-1}}{\Delta x}$$

o

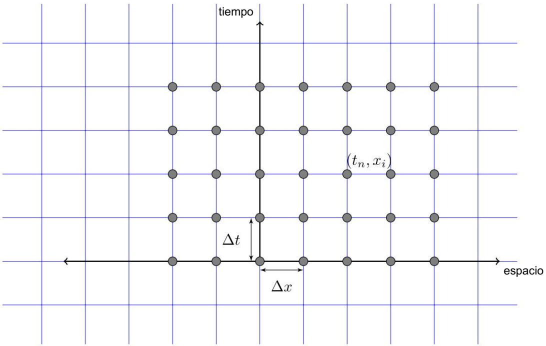
$$\frac{\partial a}{\partial x}\Big|_{yo} \approx \frac{a_{i+1} - a_i}{\Delta x}$$

$$a_{i+1} = a_i + \Delta x \frac{\partial a}{\partial x}\Big|_i + \frac{1}{2}\Delta x^2 \frac{\partial^2 a}{\partial x^2}\Big|_i + \ldots$$

Resolviendo para $\partial a/ \partial x|_i$, Nosotros vemos

$$\frac{\partial a}{\partial x}\Big|_i = \frac{a_i - a_{i-1}}{\Delta x} - \frac{1}{2}\Delta x \frac{\partial^2 a}{\partial x^2}\Big|_i + \ldots$$

$$= \frac{a_i - a_{i-1}}{\Delta x} + O(\Delta x)$$

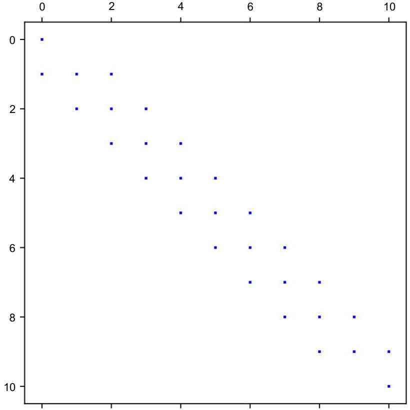


```
importar numpy como np

desde scipy.sparse importar csr_array, diags_array desde
scipy.sparse.linalg importar spsolve def

fdm_poisson1d_matrix(N: int):
    """Calcula la matriz de diferencias finitas para el problema de Poisson en 1D

    Parámetros:
    N (int): Número de puntos de la cuadrícula :math:`\\{x_i\\}_{i=0}^N` contando desde 0.

    Devoluciones:
    A (scipy.sparse._csr.csr_array): Matriz dispersa de diferencias finitas
    """

    = diag    1 / N Δx
    = np.concatenar( (

            np.ones(forma=1),
            np.full(forma=N - 1, valor_de_relleno=2 / Δx***2),
            np.ones(forma=1),
        )

    ) diag_sup =
        np.concatenar( (np.zeros(forma=1), np.completo(forma=N - 1, valor_de_relleno=-1 / Δx***2))

    ) diag_inf = np.flipud(m=diag_sup)

    devuelve diags_array( [diag,
        diag_sup, diag_inf], desplazamientos=[0,
        1, -1], forma=(N + 1, N +
        1), formato="csr",

    )
N = 10
x = np.linspace(inicio=0, fin=1, num=N + 1)
A = matriz fdm_poisson1d(N)
F = (2 * np.pi) *** 2 * np.sin(2 * np.pi * x)
```

Programa Python : fdmpoisson1d.py.

```
xfine = np.linspace(inicio=0, fin=1, num=10 * N)
# Solución de referencia analítica u =
np.sin(2 * np.pi * xfine)

# Incorporar condición de contorno en el vector derecho
F[0], F[-1] = u[0], u[-1]

#Resolver AU = F
U = spsolve(A=A, b=F)
```

Patrón de escasez de la matriz A



```
%%%MatrixMarket matriz coordenada real general %

11 11 29
1 1 1
2 1 -9.999999999999999E1
2 2 1.9999999999999997E2
2 3 -9.999999999999999E1
3 2 -9.999999999999999E1
3 3 1.9999999999999997E2
3 4 -9.999999999999999E1
4 3 -9.999999999999999E1
4 4 1.9999999999999997E2
4 5 -9.999999999999999E1
5 4 -9.999999999999999E1
5 5 1.9999999999999997E2
5 6 -9.999999999999999E1
6 5 -9.999999999999999E1
6 6 1.9999999999999999E2 6 7
-9.999999999999999E1 7 6
-9.999999999999999E1 7 7
1.99999999999999997E2 7 8
-9.999999999999999E1
8 7 -9.999999999999999E1
8 8 1.9999999999999997E2
8 9 -9.999999999999999E1
9 8 -9.999999999999999E1
9 9 1.9999999999999997E2
9 10 -9.999999999999999E1
10 9 -9.999999999999999E1
10 10 1.9999999999999997E2
10 11 -9.999999999999999E1
11 11 1
```

Programa Python : poissonA.mm.

```
%%%MatrixMarket matriz de coordenadas real general
%
1 11 10
1 2 2.3204831651684845E1
1 3 3.754620631564544E1
1 4 3,754620631564544E1 1 5
2,320483165168485E1 1 6
4,8347117754578846E-15
1 7 -2.3204831651684856E1
1 8 -3.754620631564544E1
1 9 -3.754620631564544E1
1 10 -2.3204831651684856E1
1 11 -2.4492935982947064E-16
```

Programa Python : poissonF.mm.

```
%%%MatrixMarket matriz de coordenadas real general
%
1 11 10
1 2 6.075103796733O3E-1
139.8297244282975 76E-1
1 4 9.8297244282975 76E-1
1 5 6.075103796733O3E-1 1 6
-5.1532467934608046E-17 1 7
-6.075103796733031E-1 1 8
-9.829724428297577E-1 1 9
-9.829724428297578 E-1
1 10 -6.075103796733033E-1
1 11 -2.4492935982947064E-16
```

Programa Python : poissonU.mm.

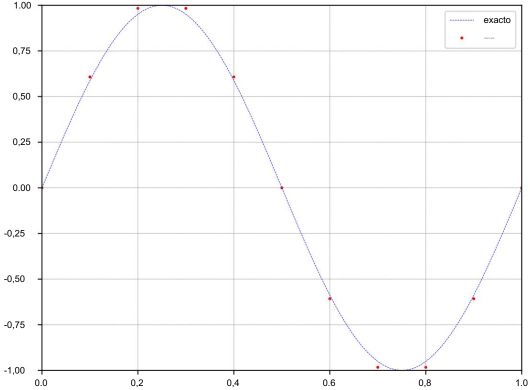


Figura: Solución.

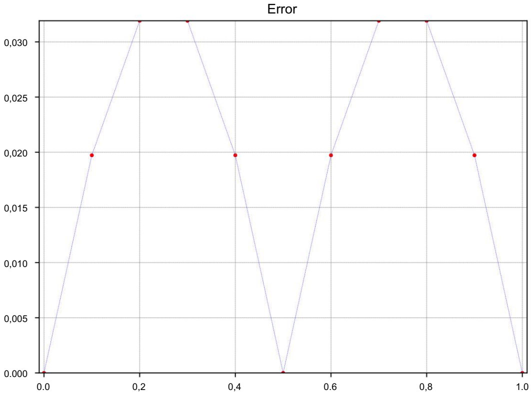
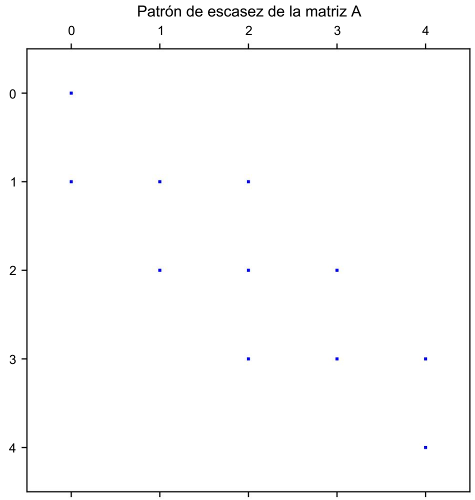


Figura: Error.

```
importar numpy como np

desde scipy.sparse importar csr_array, diags_array
desde scipy.sparse.linalg importar spsolve

def tridiag(p: np.ufunc, q: np.ufunc, N: int):

    Función de ayuda
    Devuelve una matriz tridiagonal A de dimensión N+1 x N+1.

    Δx =  1 / N
    diag = np.concatenar(
        (
            np.ones(forma=1),
            np.full(forma=N - 1, valor_de_relleno=-2 + Δx***2 * q),
            np.ones(forma=1),
        )
    )
    diag_sup = np.concatenar(
        (np.zeros(forma=1), np.full(forma=N - 1, valor_de_relleno=1 + Δx / 2 * p))
    )
    diag_inf = np.concatenar(
        (np.full(forma=N - 1, valor_de_relleno=1 - Δx / 2 * p), np.zeros(forma=1))
    )

    devolver diags_array(
        [diag, soporte_diag, inf_diag],
        desplazamientos=[0, 1, -1],
        forma=(N + 1, N + 1),
        formato="csr",
    )
N = 4 # Número de intervalos
x, Δx = np.linspace(inicio=0, fin=1, num=N + 1, retstep=True)

p = 2
q = -3
r = 9 * x

A = tridiag(p, q, N)
b = Δx***2 * r
b[0] = 1
b[N] = np.exp(-3) + 2 * np.exp(1) -             5

U = spsolve(A=A, b=b) # Resuelve la ecuación
```



Programa Python : twopointboundary.py.

```
%%%MatrixMarket matriz de coordenadas real general
%
5 5 11
1 1 1
2 1 7.5E-1
2 2 -2.1875
2 3 1,25
3 2 7.5E-1
3 3 -2,1875 3 4
1,25 4 3
7,5E-1
4 4 -2.1875
4 5 1,25
5 5 1
```

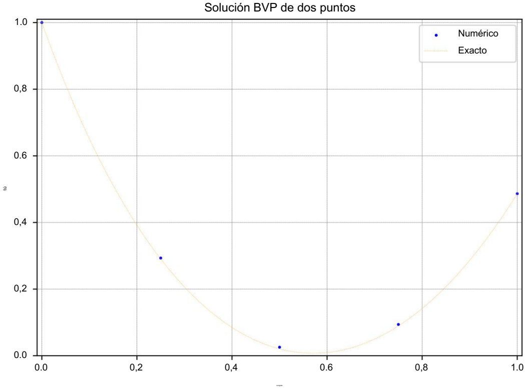
Programa Python : twopointboundaryA.mm.

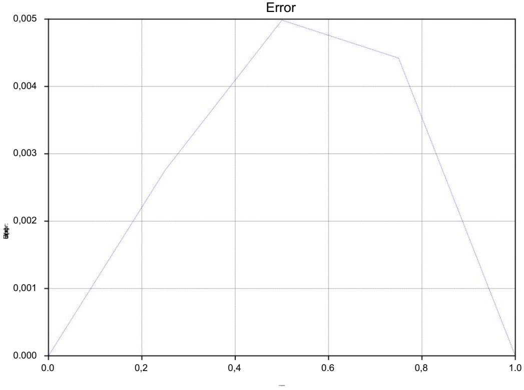
```
%%%MatrixMarket matriz coordenada real general %

1 5 5
1 1 1
1 2 1.40625E-1
132.8125E-1
1 4 4.21875E-1
1 5 4.863507252859538E-1
```

Programa Python : twopointboundaryb.mm.

```
%%%MatrixMarket matriz de coordenadas real general
%
1 5 5
1 1 1
1 2 2.931756779400817E-1
1 3 2.5557436395142973E-2
1 4 9.382010692745119E-2
1 5 4.863507252859538E-1
```

Programa Python : twopointboundaryU.mm.


Solución BVP de dos puntos
Numérico
Exacto
1.0
0,8
0.6
0,4
0,2
0.0
0.0
0,2
0,4
0.6
0,8
1.0


Error
0,005
0,004
0,003
0,002
0,001
0.000
0.0
0,2
0,4
0.6
0,8
1.0