Finite differences as approximations of partial derivatives

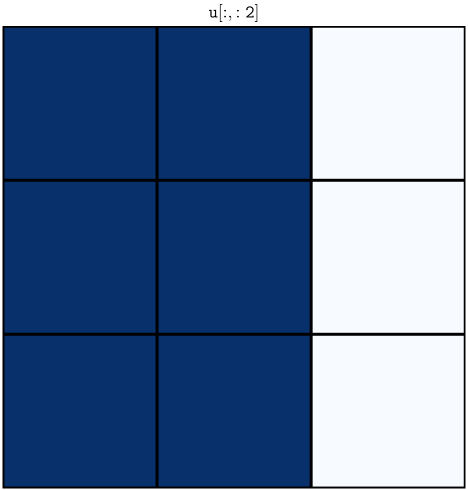


Figure: Select every all rows of u and from the first column to the second.

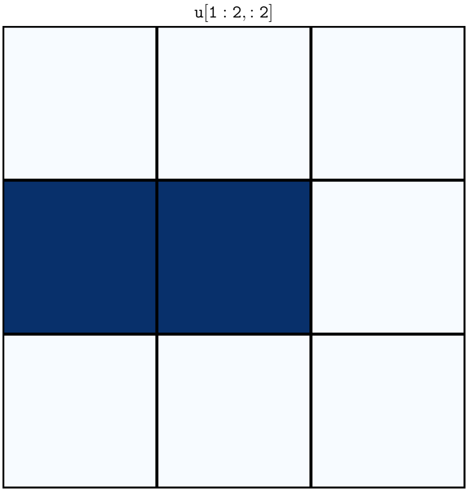


Figure: Select from the second row to the second and from the first column to the second.

Estimate the error for $\partial^+ u(x)$

By Taylor's expansion, for some $\xi \in [x, x+h]$

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

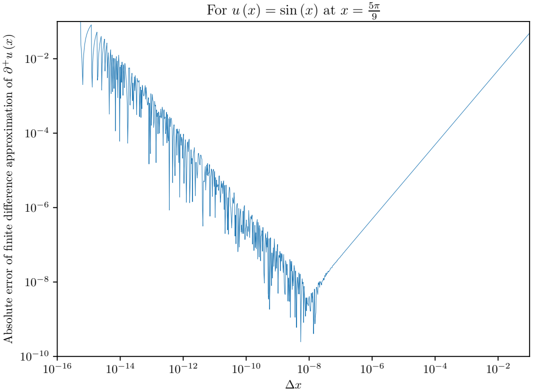


Figure: Error of $\partial^+ u(x)$ for several values of Δx.

Part of the error is due to the inaccuracies in (4) is

$$\text{truncated error} = \frac{\Delta x}{2} \left| \partial_x^2 u(\xi) \right|.$$

Depends of $\varepsilon_{\text{mach}}$ (for `float64` is $2.22044604925 \times 10^{-16}$).

$$\text{rounded error} \approx \frac{|u(x)|\, \varepsilon_{\text{mach}}}{\Delta x} + \left| \partial_x u(x) \right| \varepsilon_{\text{mach}}.$$

The best value of Δx is obtained by minimizing the total error as function of Δx, i.e.,

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

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

If $u(x)$ and $\partial_x^2 u(\xi)$ are neither large nor small, then

$$\Delta x \approx \sqrt{\varepsilon_{\text{mach}}} \quad (\text{for float64 is } 1.49011611938 \times 10^{-8}).$$

Truncation error

$$u\left(x+\Delta x\right)=\underbrace{u\left(x\right)+\Delta x\partial_{x}u\left(x\right)}_{1^{\text{st}}\text{ order approximation}}+\underbrace{\frac{\left(\Delta x\right)^{2}}{2!}\partial_{x}^{2}u\left(x\right)+\cdots}_{\text{truncation error}}.$$

$$u\left(x+\Delta x\right)=\text{approximation}+E\left(\Delta x\right)+\text{high order terms}.$$

$$u\left(x+\Delta x\right)\approx u\left(x\right)+\Delta x\partial_{x}u.$$

$$E\left(\Delta x\right)=\frac{\left(\Delta x\right)^{n+1}}{\left(n+1\right)!}\partial_{x}^{n+1}u\left(x\right).$$

```python
import numpy as np

x = 0
Δx = 0.1
u = np.cos

def dudx(x):
    return -np.sin(x)

def du2dx(x):
    return -np.cos(x)

exact = u(x + Δx)
approximation = u(x) + Δx * dudx(x)
absolute_lower_order_ommited_term = np.abs(np.power(Δx, 2) / 2 * du2dx(x))
truncation_error = np.abs(exact - approximation)
error_attributed_to_other_ommited_terms = np.abs(
    truncation_error - absolute_lower_order_ommited_term
)
```

```
First order approximation: 1.0
Exact: 0.9950041652780258
Truncation error: 0.0049958347219741794
Absolute lower order ommited term: 0.005000000000000001
Error atributed to other terms: 4.165278025821534e-06
```

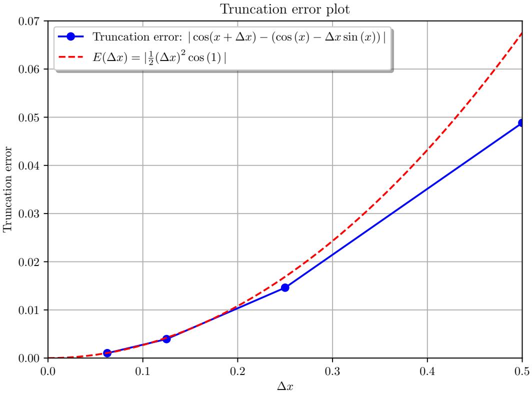


Figure: Truncation error for several values of Δx at $x=1$.

Finite differences as approximations of partial derivatives

Given a known truncation error $E(\Delta x) = O((\Delta x)^n)$ it is possible to estimate the order of an approximation.

$$E(\Delta x) \approx C(\Delta x)^n.$$

Taking logarithms of both sides gives

$$\log|E(\Delta x)| = n\log(\Delta x) + \log(C).$$

which is a linear function in Δx. Therefore, one way of approximate the slope n as a gradient is

$$\boxed{n \approx \frac{\log|E(\max(\Delta x))| - \log|E(\min(\Delta x))|}{\log(\max(\Delta x)) - \log(\min(\Delta x))}}.$$

```python
import numpy as np
from jaxtyping import Array, Float

u = np.cos

def dudx(x):
    return -np.sin(x)

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

backward: Float[Array, "dim1"] = (u(x) - u(x - Δx)) / Δx
centered: Float[Array, "dim1"] = (u(x + Δx) - u(x - Δx)) / (2 * Δx)
forward: Float[Array, "dim1"] = (u(x + Δx) - u(x)) / Δx

error_backward: Float[Array, "dim1"] = np.abs(dudx(x) - backward)
error_centered: Float[Array, "dim1"] = np.abs(dudx(x) - centered)
error_forward = np.abs(dudx(x) - forward)

def estimate_order(
    Δx: Float[Array, "dim1"], truncation_error: Float[Array, "dim1"]
) -> float:
    assert Δx.size == truncation_error.size

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

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

print(f"Order of convergence of backward is {estimate_order(Δx, error_backward)}")
print(f"Order of convergence of centered is {estimate_order(Δx, error_centered)}")
print(f"Order of convergence of forward is {estimate_order(Δx, error_forward)}")
```

Program : Determine order of convergence .

Δx	backward	centered	forward
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	backward	centered	forward
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

```
Order of convergence of backward is 1.013379633444132
Order of convergence of centered is 1.9997633049971728
Order of convergence of forward is 0.9853046896368071
```

Order of convergence

```python
from typing import Callable

import numpy as np

def u(x: float) → float:
    """Sample function
    u:ℝ ⟶ ℝ
      x ⟼ exp(x^2)
    """
    return np.exp(np.pow(x, 2))

def up(x: float) → float:
    """Derivative of sample function
    u':ℝ ⟶ ℝ
       x ⟼ 2*x*f(x)
    """
    return 2 * x * u(x)

def ff(u: Callable, Δx: np.array) → np.array:
    """Forward finite difference approximation
    u' = (u(x + Δx) - u(x)) / Δx
    """
    return (u(x + Δx) - u(x)) / Δx

def bf(u: Callable, Δx: np.array) → np.array:
    """Backward finite difference approximation
    u' = (u(x) - u(x - Δx)) / Δx
    """
    return (u(x) - u(x - Δx)) / Δx

def cf(u: Callable, Δx: np.array) → np.array:
    """Centered finite difference approximation
    u' = (u(x + Δx / 2) - u(x - Δx / 2)) / Δx
    """
    return (u(x + Δx / 2) - u(x - Δx / 2)) / Δx

x = 2
exact = up(x)
Δx = np.logspace(start=-16, stop=-1.0, num=16)
```

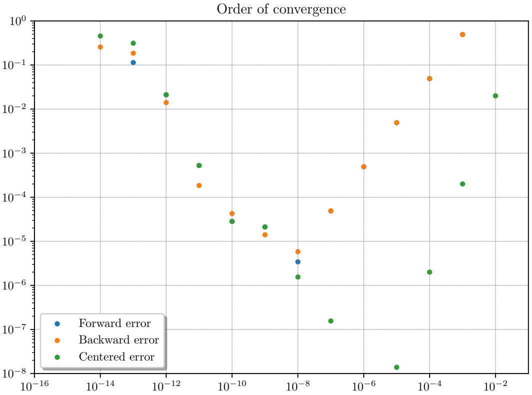


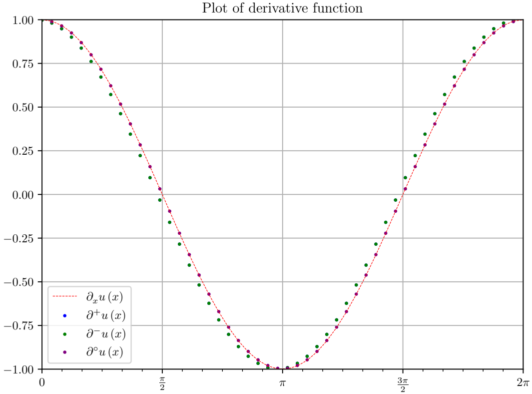
Figure: $\partial^{+}u\left(x\right)$, $\partial^{-}u\left(x\right)$, $\partial^{\circ}u\left(x\right)$ of $u\left(x\right)=\exp\left(x^2\right)$ for several values of Δx at $x=2$.

Finite differences as approximations of partial derivatives

In the array computation, we do not consider the last element of the difference $u_{i+1} - ui$. and the first element of the difference $u_{i+1} - ui$.

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

$$\partial^{-} u\left(x\right) = \frac{u_{i} - u_{i-1}}{\Delta x}, \quad i = 1, \ldots, n.$$




```python
import numpy as np

x, Δx = np.linspace(start=0, stop=2 * np.pi, retstep=True)
y = np.sin(x)

forward = (np.roll(y, -1) - y)[:-1] / Δx
backward = (y - np.roll(y, 1))[1:] / Δx
centered = (np.roll(y, -1) - np.roll(y, 1))[1:-1] / (2 * Δx)
first_derivative = np.cos(x)
```

$\mathrm{np.roll}(a = u, \mathrm{shift} = -1), u, \mathrm{np.roll}(a = u, \mathrm{shift} = -1) - u$

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

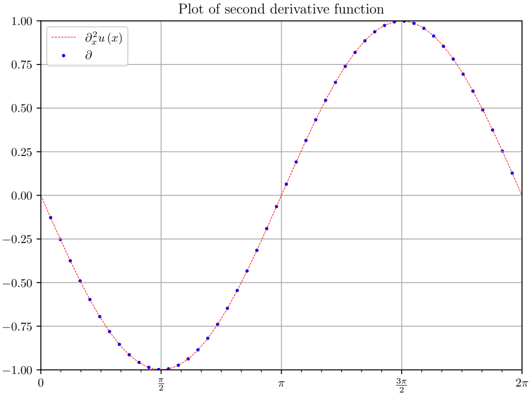
$\mathrm{np.roll}(a = u, \mathrm{shift} = 1), u, u - \mathrm{np.roll}(a = u, \mathrm{shift} = 1)$

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

Finite differences as approximations of partial derivatives

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

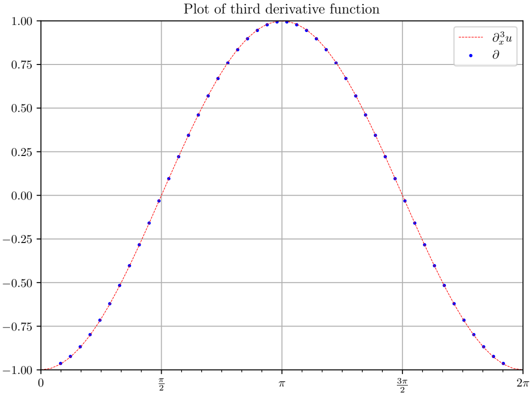
```python
import numpy as np

x, Δx = np.linspace(start=0, stop=2 * np.pi, retstep=True)
y = np.sin(x)

plt.ylim(-1, 1)
plt.grid()
```



Finite differences as approximations of partial derivatives

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

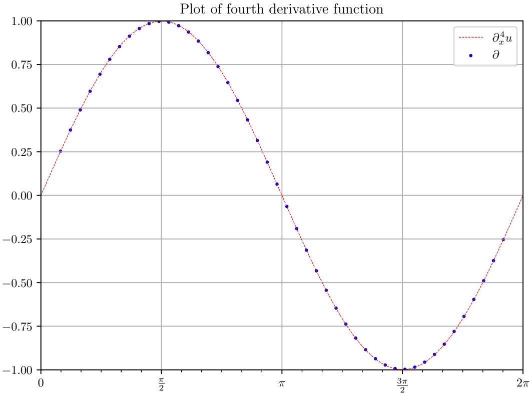
```python
import numpy as np

x, Δx = np.linspace(start=0, stop=2 * np.pi, retstep=True)
y = np.sin(x)

plt.gca().xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
plt.gca().xaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))
plt.scatter(x=x[1:-1], y=partial2x, c="blue", s=3, label=r"$\partial$")
plt.title(label="Plot of second derivative function")
```



Finite differences as approximations of partial derivatives

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

```
import numpy as np

x, Δx = np.linspace(start=0, stop=2 * np.pi, retstep=True)
y = np.sin(x)

plt.gca().xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 12))
plt.gca().xaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter()))
plt.scatter(x=x[2:-2], y=partial3x, c="blue", s=3, label=r"$\partial$")
plt.title(label="Plot of third derivative function")
```



Complex step method

Definition (Holomorphic function)

Let $D \subset \mathbb{C}$ be a simply connected, open region and $u \colon D \to \mathbb{C}$. We say that u is complex differentiable at $a \in D$ iff

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

exists. If u is complex differentiable at every point of D, then we say that u is holomorphic in D.

The complex step derivative approximation is a technique to compute the derivative of a real-valued function $u(x)$. For u analytic,

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

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

Comparing imaginary parts of the two sides gives

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

Remark

Behind the scenes, the complex step method is a particular case of automatic differentiation.

Complex step method

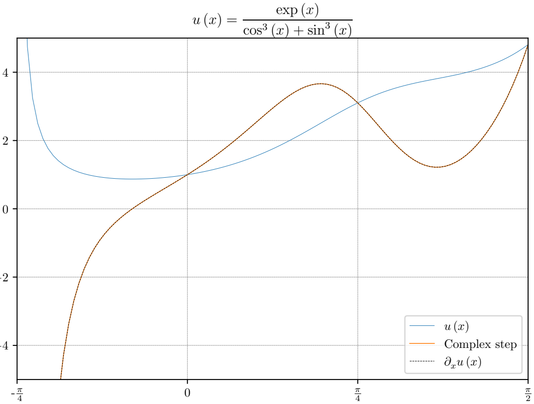
$$u(x+i\Delta x) = u(x) + i\Delta x \partial_x u(x) + \frac{(i\Delta x)^2}{2!}\partial_x^2 u(x) + \cdots .$$

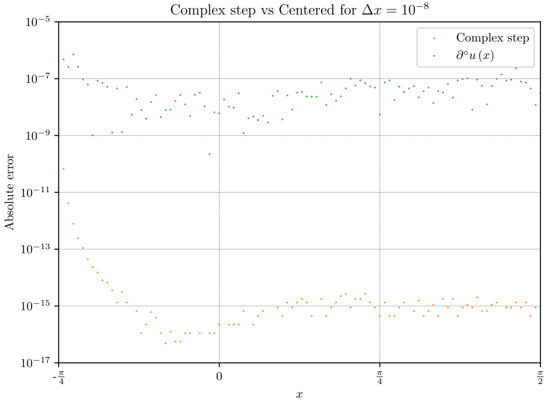
$$\operatorname{Re}[u(x+i\Delta x)] + i\operatorname{Im}[u(x+i\Delta x)] \approx u(x) + i\Delta x \partial_x u(x).$$

Comparing real parts of the two sides gives

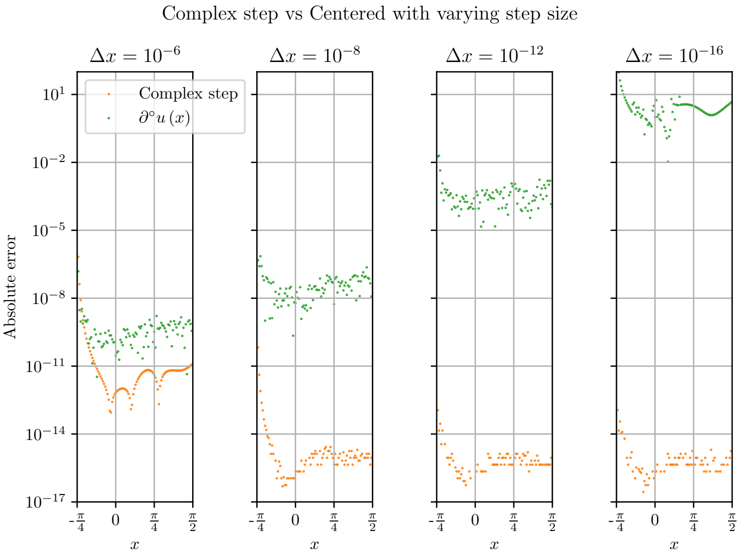
$$\partial_x^2 u(x) \approx \frac{2}{\Delta x^2}\left(u(x) - \operatorname{Re}[u(x+i\Delta x)]\right).$$

Complex step method





Complex step method



Solve BVP for ODEs

Let's follow the steps:

1. Discretize the domain on which the equation is defined.
2. On each grid point, replace the derivatives with an approximation, using the values in neighbouring grid points.
3. Replace the exact solutions by their approximations.
4. Solve the resulting system of equations.

Finite difference for Two-point BVP

We will first see how to find approximations to the derivative of a function, and then how these can be used to solve boundary value problems like

$$\begin{cases} \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + p(x)\frac{\mathrm{d}u}{\mathrm{d}x} + q(x)\,u = r(x) & \text{for } a \leq x \leq b. \\ u(a) = u_a, \quad u(b) = u_b \end{cases}.$$

This technique described here is applicable to several other time dependent PDEs, and it is therefore important to try to understand the underlying idea.

Example (Two-point BVP FDM for the 1D Poisson Problem)

Let $f\colon [0,1] \to \mathbb{R}$ be a function. Find a $u\colon [0,1] \to \mathbb{R}$ such that

$$\begin{cases} -\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = f(x), & x \in (0,1). \\ u(0) = u_a, \quad u(1) = u_b. \end{cases} \tag{5}$$

Instead of trying to compute $u(x)$ exactly, we will now try to compute a numerical approximation u_{Δ} of $u(x)$. As many times before we start by defininig $n+1$ equally spaced points $\{x_i\}_{i=0}^{n}$ with a grid size $h = \frac{b-a}{n}$ so that

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

Consider a collection of equally spaced points, labeled with an index i, with the physical spacing between them denoted Δx. We can express the first derivative of a quantity a at i as:

$$\frac{\partial a}{\partial x}_i \approx \frac{a_i - a_{i-1}}{\Delta x}$$

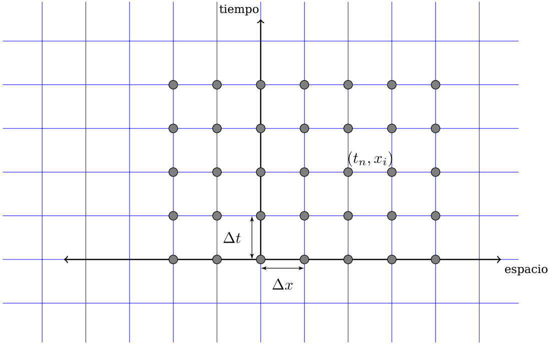
or

$$\frac{\partial a}{\partial x}_i \approx \frac{a_{i+1} - a_i}{\Delta x}$$

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

Solving for $\left.\partial a / \partial x\right|_i$, we see

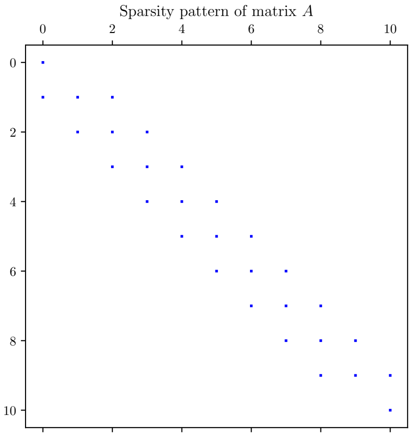
$$\begin{aligned}\left.\frac{\partial a}{\partial x}\right|_i &= \frac{a_i - a_{i-1}}{\Delta x} - \frac{1}{2}\Delta x \left.\frac{\partial^2 a}{\partial x^2}\right|_i + \ldots \\ &= \frac{a_i - a_{i-1}}{\Delta x} + \mathcal{O}(\Delta x)\end{aligned}$$



```python
import numpy as np

from scipy.sparse import csr_array, diags_array
from scipy.sparse.linalg import spsolve

def fdm_poisson1d_matrix(N: int):
    """Computes the finite difference matrix for the Poisson problem in 1D

    Parameters:
    N (int): Number of grid points :math:`\\{x_i\\}_{i=0}^N` counting from 0.

    Returns:
    A (scipy.sparse._csr.csr_array): Finite difference sparse matrix

    """
    Δx = 1 / N
    diag = np.concatenate(
        (
            np.ones(shape=1),
            np.full(shape=N - 1, fill_value=2 / Δx**2),
            np.ones(shape=1),
        )
    )
    diag_sup = np.concatenate(
        (np.zeros(shape=1), np.full(shape=N - 1, fill_value=-1 / Δx**2))
    )
    diag_inf = np.flipud(m=diag_sup)

    return diags_array(
        [diag, diag_sup, diag_inf],
        offsets=[0, 1, -1],
        shape=(N + 1, N + 1),
        format="csr",
    )

N = 10
x = np.linspace(start=0, stop=1, num=N + 1)
A = fdm_poisson1d_matrix(N)
F = (2 * np.pi) ** 2 * np.sin(2 * np.pi * x)
```

Program : fdmpoisson1d.py.

```python
xfine = np.linspace(start=0, stop=1, num=10 * N)
# Analytical reference solution
u = np.sin(2 * np.pi * xfine)

# Incorporate boundary condition into rhs vector
F[0], F[-1] = u[0], u[-1]

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



```
%%MatrixMarket matrix coordinate 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 1.9999999999999997E2
6 7 -9.999999999999999E1
7 6 -9.999999999999999E1
7 7 1.9999999999999997E2
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
```

Program : `poissonA.mm`.

```
%%MatrixMarket matrix coordinate 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
```

Program : `poissonF.mm`.

```
%%MatrixMarket matrix coordinate real general
%
1 11 10
1 2 6.07510379673303E-1
1 3 9.829724428297576E-1
1 4 9.829724428297576E-1
1 5 6.07510379673303E-1
1 6 -5.1532467934608046E-17
1 7 -6.075103796733031E-1
1 8 -9.829724428297577E-1
1 9 -9.829724428297578E-1
1 10 -6.075103796733033E-1
1 11 -2.4492935982947064E-16
```

Program : `poissonU.mm`.

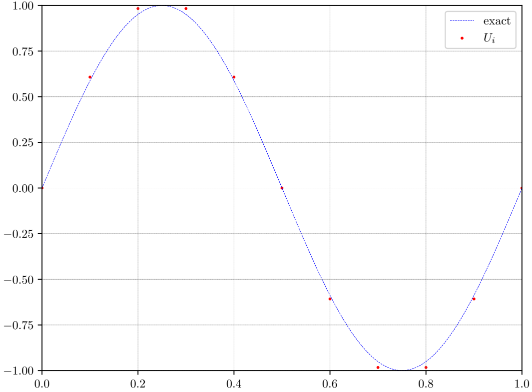


Figure: Solution.

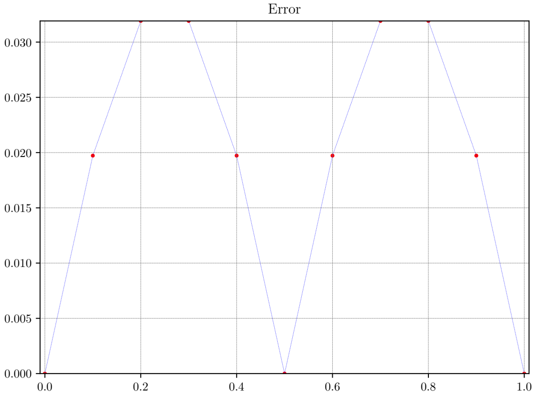
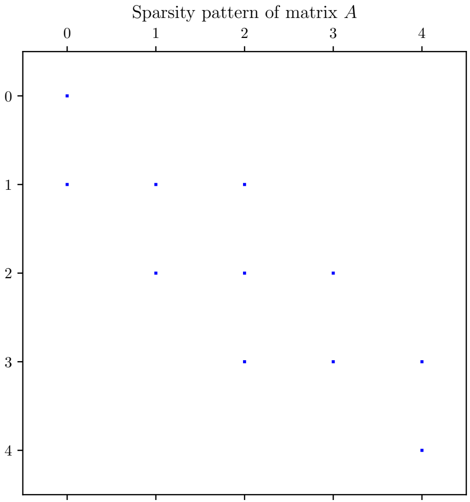


Figure: Error.

```python
import numpy as np

from scipy.sparse import csr_array, diags_array
from scipy.sparse.linalg import spsolve

def tridiag(p: np.ufunc, q: np.ufunc, N: int):
    """
    Help function
    Returns a tridiagonal matrix A of dimension N+1 x N+1.
    """
    Δx = 1 / N
    diag = np.concatenate(
        (
            np.ones(shape=1),
            np.full(shape=N - 1, fill_value=-2 + Δx**2 * q),
            np.ones(shape=1),
        )
    )
    diag_sup = np.concatenate(
        (np.zeros(shape=1), np.full(shape=N - 1, fill_value=1 + Δx / 2 * p))
    )
    diag_inf = np.concatenate(
        (np.full(shape=N - 1, fill_value=1 - Δx / 2 * p), np.zeros(shape=1))
    )

    return diags_array(
        [diag, diag_sup, diag_inf],
        offsets=[0, 1, -1],
        shape=(N + 1, N + 1),
        format="csr",
    )

N = 4  # Number of intervals
x, Δx = np.linspace(start=0, stop=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
```

```python
U = spsolve(A=A, b=b)  #  Solve the equation
```



Program : twopointboundary.py.

```
%%MatrixMarket matrix coordinate 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
```

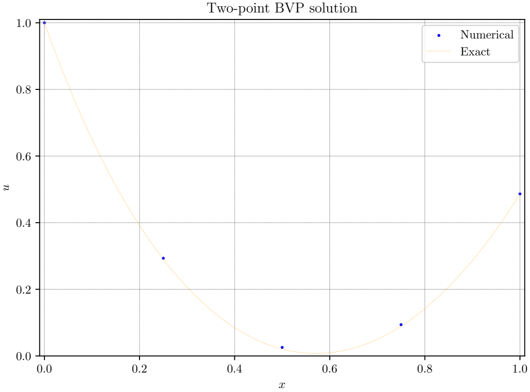
Program : `twopointboundaryA.mm`.

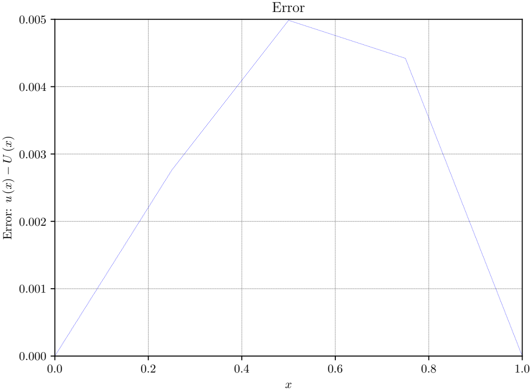
```
%%MatrixMarket matrix coordinate real general
%
1 5 5
1 1 1
1 2 1.40625E-1
1 3 2.8125E-1
1 4 4.21875E-1
1 5 4.863507252859538E-1
```

Program : `twopointboundaryb.mm`.

```
%%MatrixMarket matrix coordinate 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
```

Program : `twopointboundaryU.mm`.


Two-point BVP solution
Numerical
Exact
1.0
0.8
0.6
0.4
0.2
0.0
u
0.0
0.2
0.4
0.6
0.8
1.0
x


Error
0.005
0.004
0.003
0.002
0.001
0.000
Error: $u(x) - U(x)$
0.0
0.2
0.4
0.6
0.8
1.0
x