

Finite differences as approximations of partial derivatives

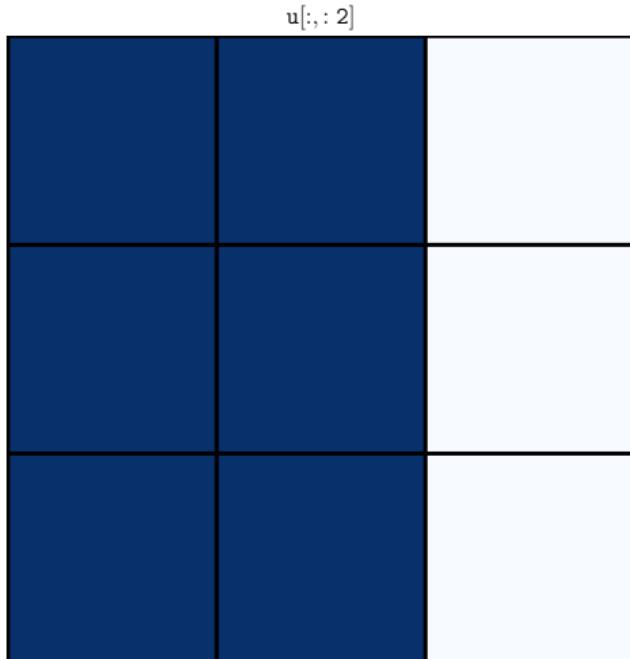


Figure: Select **every all rows** of u and from the **first column to the second**.

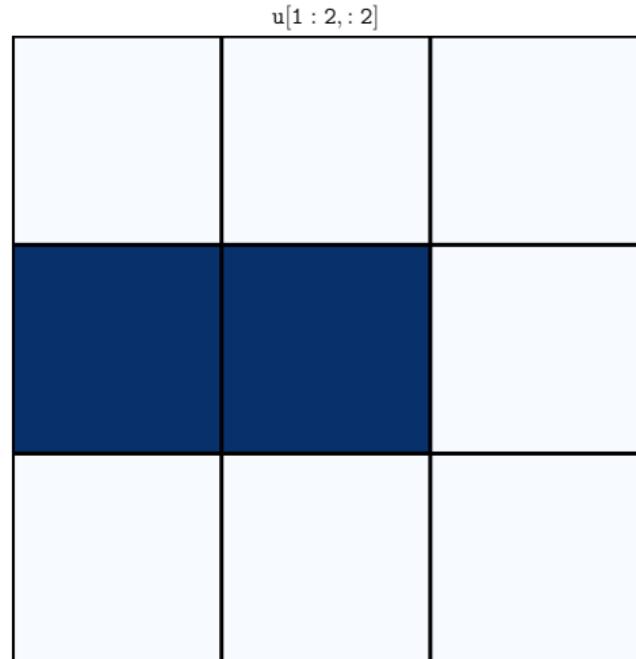


Figure: Select from the **second row to the second** and from the **first column to the second**.

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

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

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

For $u(x) = \sin(x)$ at $x = \frac{5\pi}{9}$

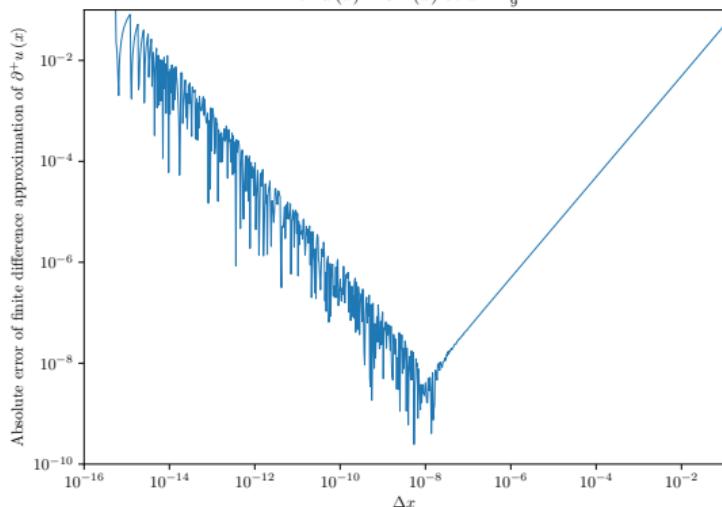


Figure: Error of $\partial^+ u(x)$ for several values of Δx .

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

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

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

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

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

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

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

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

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

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

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

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

```
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_omitted_term = np.abs(np.power(Δx, 2) / 2 * du2dx(x))
truncation_error = np.abs(exact - approximation)
error_attributed_to_other_omitted_terms = np.abs(
    truncation_error - absolute_lower_order_omitted_term
)

First order approximation: 1.0
Exact: 0.9950041652780258
Truncation error: 0.0049958347219741794
Absolute lower order omitted term: 0.0050000000000000001
Error attributed to other terms: 4.165278025821534e-06
```

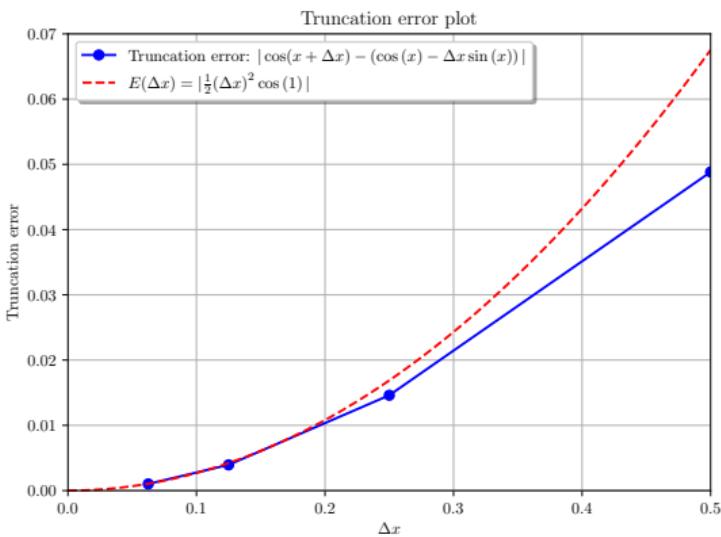


Figure: Truncation error for several values of Δx at $x = 1$

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

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

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

```
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 .

Order of convergence

```
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)
```

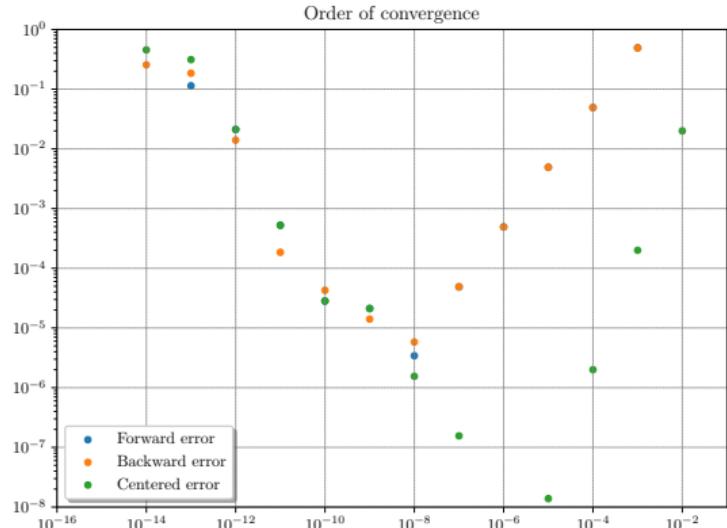


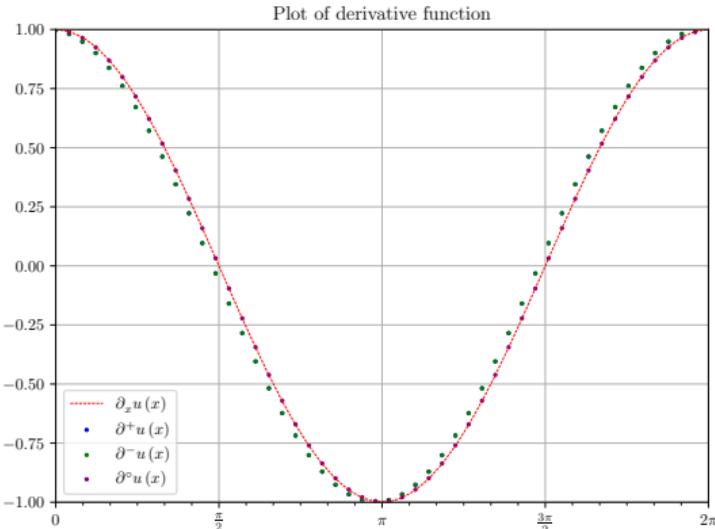
Figure: $\partial^+ u(x), \partial^- u(x), \partial^0 u(x)$ of $u(x) = \exp(x^2)$ for several values of Δx at $x = 2$

Finite differences as approximations of partial derivatives

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

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

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



```
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)  
np.roll(a=u, shift=-1), u, np.roll(a=u, 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]

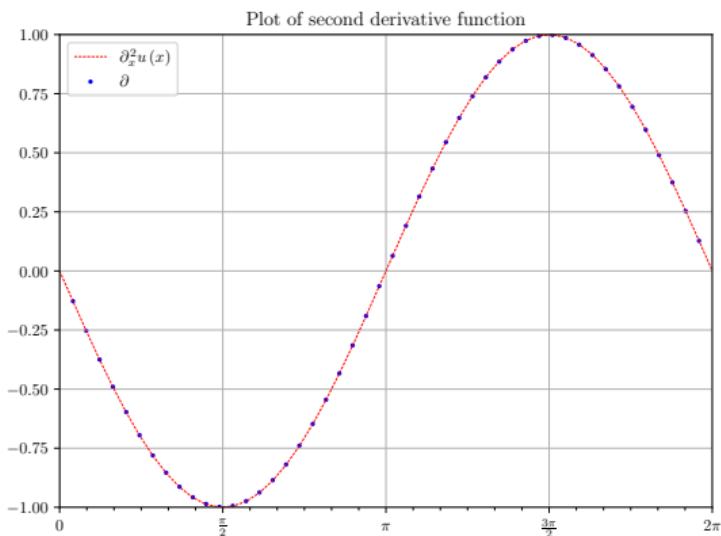
```
np.roll(a=u, shift=1), u, u - np.roll(a=u, shift=1)
```

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

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

```
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()
```

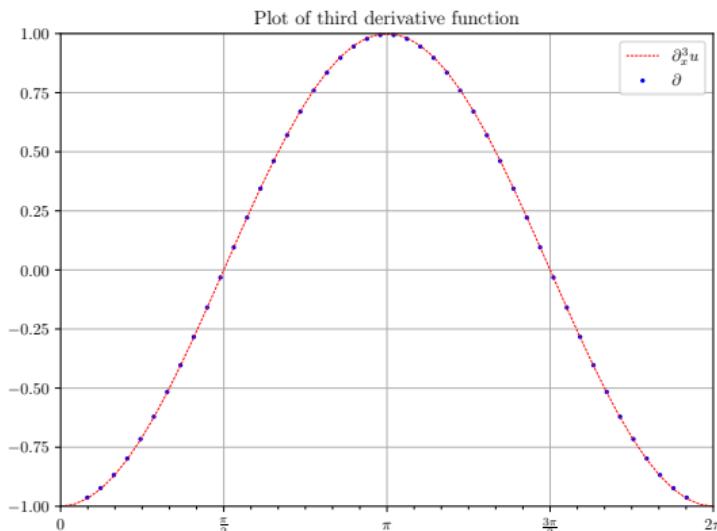


Finite differences as approximations of partial derivatives

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

```
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[1:-1], y=partial2x, c="blue", s=3, label=r"\partial")
plt.title(label="Plot of second derivative function")
```

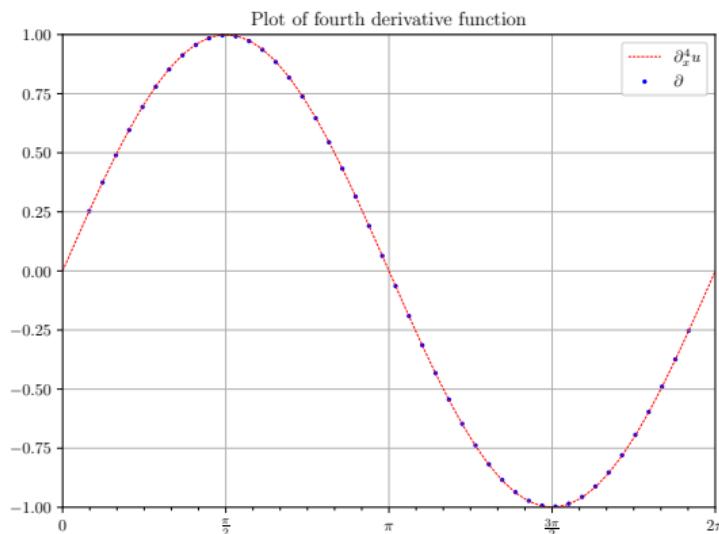


Finite differences as approximations of partial derivatives

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

```
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 $u: D \rightarrow \mathbb{C}$. We say that u is **complex differentiable** at $a \in D$ iff

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

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,

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

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

Comparing 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

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

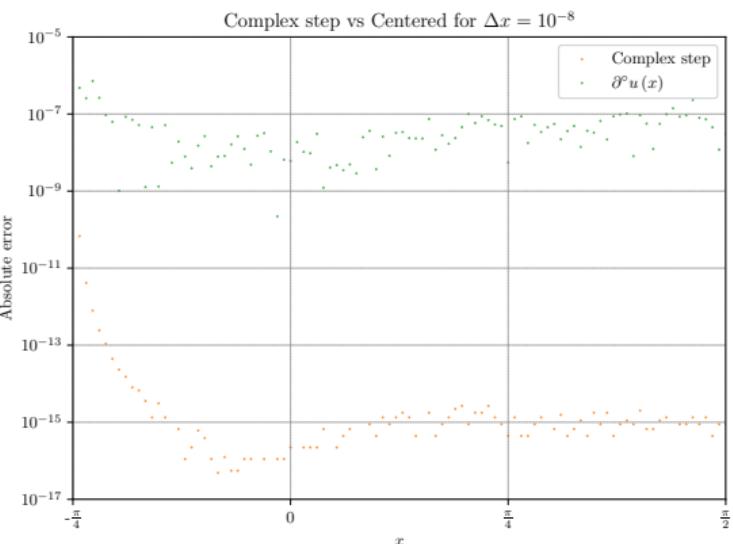
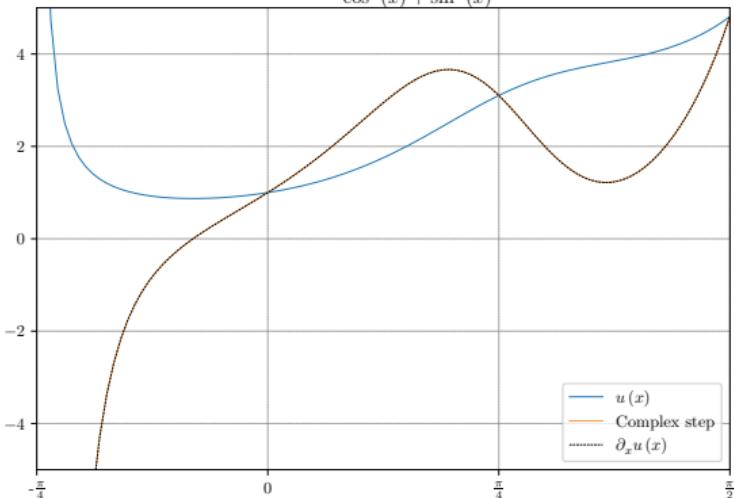
$$\operatorname{Re}[u(x + i\Delta x)] + i \operatorname{Im}[u(x + i\Delta x)] \approx u(x) + i\Delta x \partial_x u(x).$$

Comparing real parts of the two sides gives

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

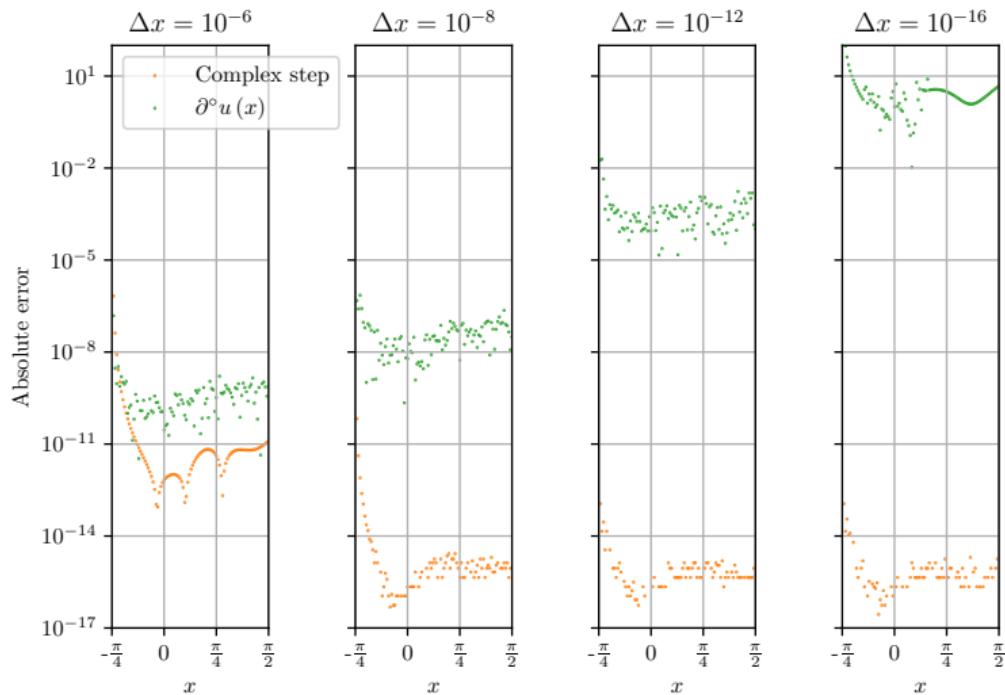
Complex step method

$$u(x) = \frac{\exp(x)}{\cos^3(x) + \sin^3(x)}$$



Complex step method

Complex step vs Centered with varying step size



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

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

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

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

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 defining $n+1$ equally spaced points $\{x_i\}_{i=0}^n$ with a grid size $h = \frac{b-a}{n}$ so that

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

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:

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

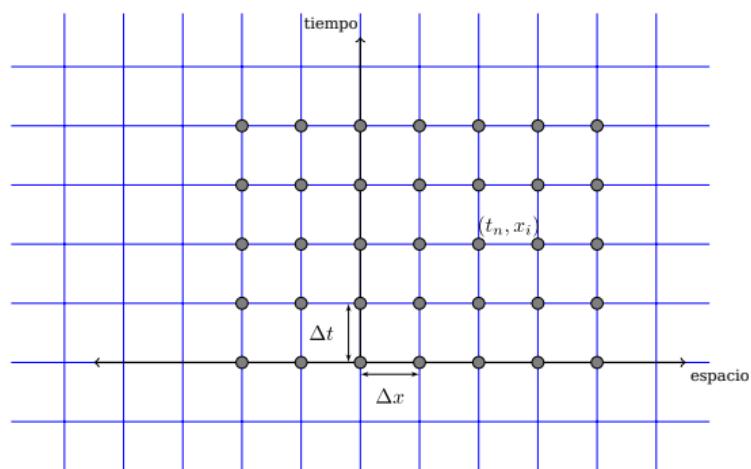
or

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

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

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

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



```

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`.

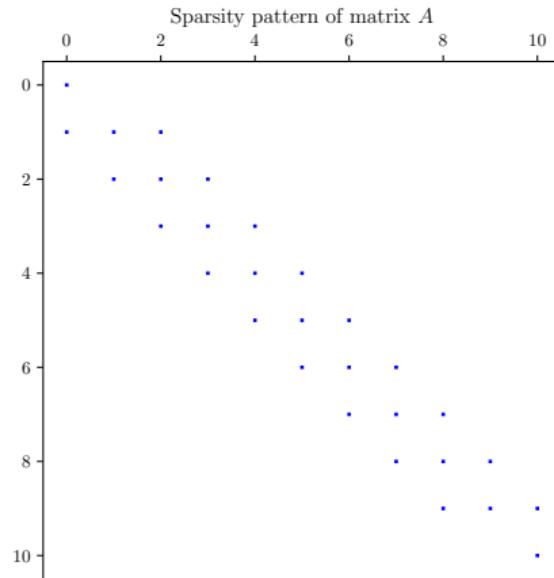
```

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 -9.99999999999999E1
2 2 1.999999999999997E2
2 3 -9.99999999999999E1
3 2 -9.99999999999999E1
3 3 1.999999999999997E2
3 4 -9.99999999999999E1
4 3 -9.99999999999999E1
4 4 1.999999999999997E2
4 5 -9.99999999999999E1
5 4 -9.99999999999999E1
5 5 1.999999999999997E2
5 6 -9.99999999999999E1
6 5 -9.99999999999999E1
6 6 1.999999999999997E2
6 7 -9.99999999999999E1
7 6 -9.99999999999999E1
7 7 1.999999999999997E2
7 8 -9.99999999999999E1
8 7 -9.99999999999999E1
8 8 1.999999999999997E2
8 9 -9.99999999999999E1
9 8 -9.99999999999999E1
9 9 1.999999999999997E2
9 10 -9.99999999999999E1
10 9 -9.99999999999999E1
10 10 1.999999999999997E2
10 11 -9.99999999999999E1
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.

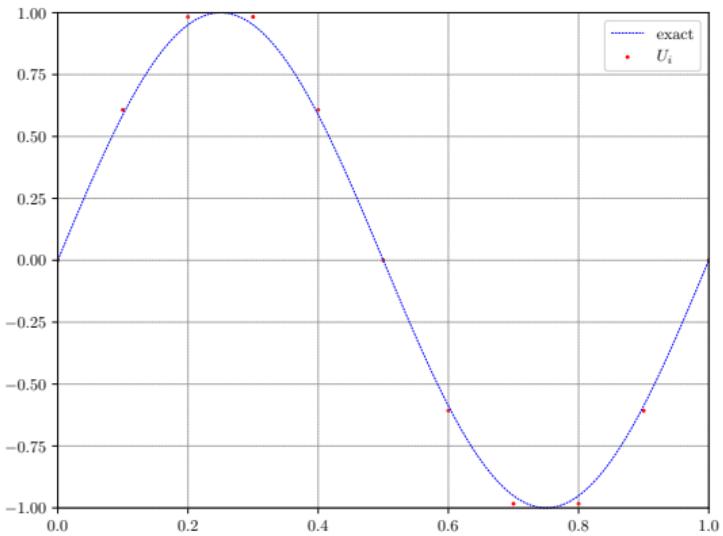


Figure: Solution.

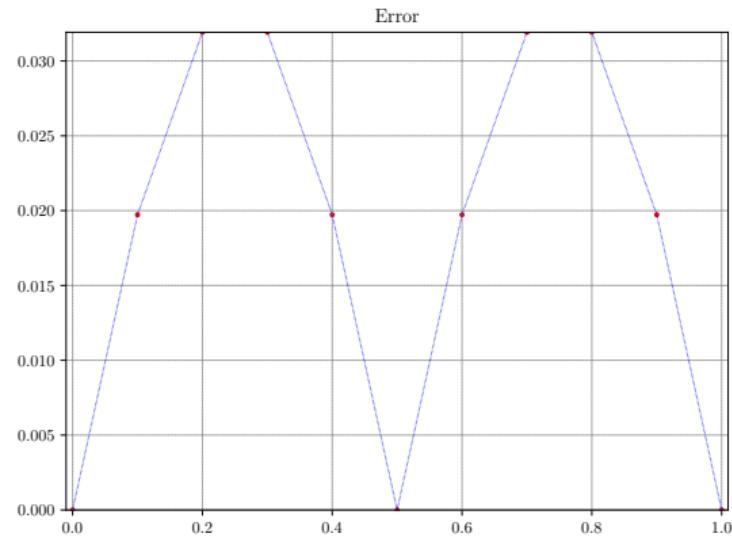


Figure: Error.

```

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)

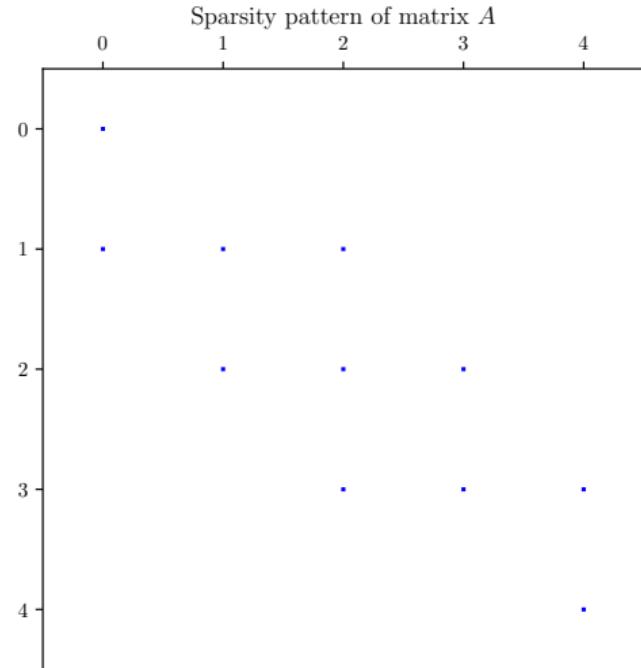
p = 2
q = -3
r = 9 * x

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

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
```

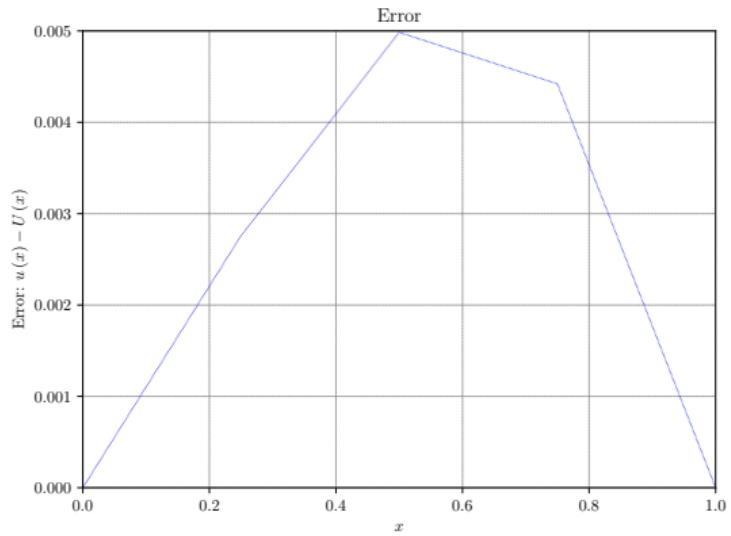
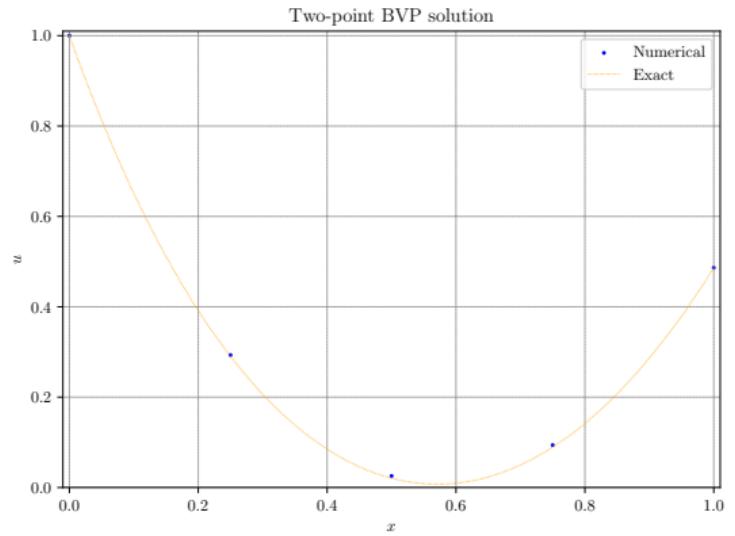
Program  : `twopointboundaryA.mm`.

```
%%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`.

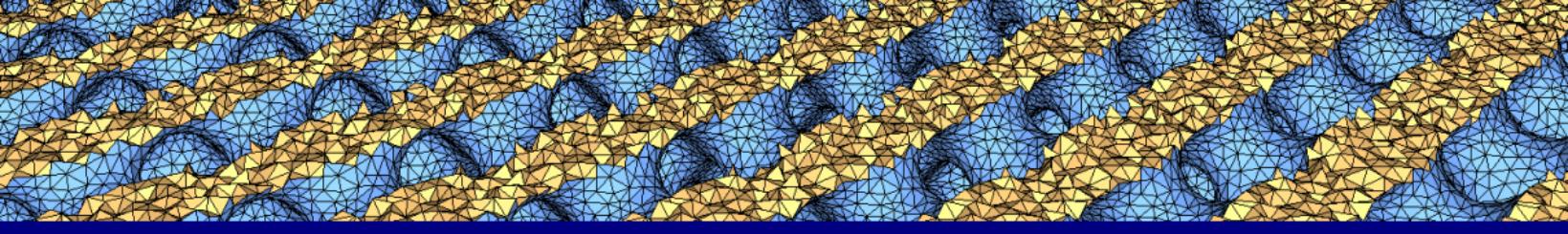
```
%%MatrixMarket matrix coordinate real general
%
1 5 5
1 1 1
1 2 2.931756779400817E-1
1 3 2.5557436395142973E-2
1 4 9.382010692745119E-2
1 5 4.863507252859538E-1
```

Program  : `twopointboundaryU.mm`.



$$u(x, t) = \sum_{\nu=0}^{\infty} a_{\nu}^0 e^{i\nu x} e^{\nu^2 t}$$

$$U_n^m = \sum_{\nu=0}^N a_{\nu}^m e^{i\nu nh} := \sum_{\nu=0}^N a_{\nu}^0 \omega_{\nu}^m e^{i\beta_{\nu} n}.$$



Fourier stability analysis

Fourier stability analysis

We will introduce some simple spatial discretizations for the periodic constant-coefficient advection-diffusion problem on a uniform grid $\Omega_h = \{x_1, \dots, x_m\}$ with grid points $x_j = jh$ and mesh width $h = \frac{1}{m}$.

Discrete Fourier decomposition is an important tool to analyze **linear difference schemes** with spatial periodic conditions. It can be used to study fundamental properties like **stability**, **dissipation** and **dispersion**.

Definition

Fourier modes

$$\forall k \in \mathbb{Z} : \varphi_k(x) = \exp(2\pi i k x).$$

These modes form an orthonormal basis for $L^2[0, 1]$ space, i.e.,

$$\forall v \in L^2[0, 1] : \sum_{k \in \mathbb{Z}} \alpha_k \varphi_k(x),$$

where the right-hand side is a convergent series, which we now call the Fourier series. The Fourier coefficients are given by $\alpha_k = \langle \varphi_k, v \rangle$.

Definition

Discrete Fourier modes

$$\forall k \in \mathbb{Z} : \phi_k(x) = (\varphi_k(x_1), \dots, \varphi_k(x_m)) \in \mathbb{C}^m.$$

The Courant-Friedrichs-Lowy (CFL) condition

Explicit schemes for the advection equation $\partial_t u + c \partial_x u = 0$ give rise to step size restrictions (stability conditions) of the form

$$|c| \frac{\Delta t}{\Delta x} \leq C,$$

with C an appropriate positive constant independent of Δx and Δt .

Remark

In the beginning of numerical analysis, finite difference approximations were used to prove the existence of PDE solutions. For **convergence** of finite difference approximations are necessary in order to guarantee that the mathematical domain of dependence of a PDE problem lies within the numerical counterpart of the finite difference method.

$$U_j^{n+1} = \sum_{k=-r}^r \gamma_k U_{j+k}^n.$$

A necessary condition for stability is that the mathematical domain of dependence of PDE is contained in the numerical domain of dependence.

Fourier stability analysis

In the 1940's, John von Neumann introduced Fourier analysis in the theory of finite difference schemes for time-dependent PDEs. We are interested in the propagation of small errors at different grid points. If these errors are not controlled at each stage of the time iteration, they can grow and create a solution completely different from the desired solution at a later time. An initial pulse which has bounded size but oscillates with frequency k . Take a complex exponential for some $k \in \mathbb{R}$

$$e^{ikx} = \cos(kx) + i \sin(kx).$$

The size of the complex-valued pulse is given by its modulus, and

$$|e^{ikx}| = |\cos(kx) + i \sin(kx)| = 1.$$

The **growth in size** when applying the scheme once to e^{ikx} , captured by an amplification constant called the **growth factor**. At time step n , for some $j \in \mathbb{Z}$ we take

$$U_j^n(k) \equiv \lambda(k)^n e^{ik(j\Delta x)} \implies \begin{cases} U_j^{n+1}(k) &= \lambda(k)^{n+1} e^{ik(j\Delta x)}. \\ U_{j+1}^n(k) &= \lambda(k)^n e^{ik(j+1)\Delta x} e^{ik\Delta x}. \\ U_{j-1}^n(k) &= \lambda(k)^n e^{ik(j-1)\Delta x} e^{-ik\Delta x}. \\ U_j^{n-1}(k) &= \lambda(k)^{n-1} e^{ik(j-1)\Delta x}. \end{cases}$$

Remark

$$\operatorname{Re}(e^{ik\Delta x}) = \cos(k\Delta x) = \frac{e^{ik\Delta x} + e^{-ik\Delta x}}{2}, \quad \operatorname{Im}(e^{ik\Delta x}) = \sin(k\Delta x) = \frac{e^{ik\Delta x} - e^{-ik\Delta x}}{2i}.$$

Example (Transport equation on a periodic domain)

For some constant $c > 0$ and some continuous and bounded function $g: [0, 1] \rightarrow \mathbb{R}$.

$$\begin{cases} \partial_t u + c\partial_x u = 0, & x \in (0, 1), t > 0. \\ u(0, t) = u(1, t), & t > 0. \\ u(x, 0) = g(x), & x \in [0, 1]. \end{cases}$$

We consider the **implicit** FDM

$$(6) \quad \begin{cases} \frac{v_j^{m+1} - v_j^m}{\Delta t} + c \frac{v_j^{m+1} - v_{j-1}^{m+1}}{\Delta x} = 0 & j = 1, \dots, n+1, m = 0, 1, \dots \\ v_0^{m+1} = v_{n+1}^{m+1} & m = 0, 1, \dots \\ v_j^0 = g(x_j) & j \in \mathbb{Z}. \end{cases}$$

Show that for any choice of $\Delta t, \Delta x > 0$, any solution of (6) satisfies

$$\inf_{x \in [0, 1]} g(x) \leq v_j^m \leq \sup_{x \in [0, 1]} g(x).$$

Fourier stability analysis

Solution

Let J be a point at which $\{v_J^{m+1}\}_{j=1}^{n+1}$ attains its maximum. Then

$$v_J^{m+1} = v_J^m - c \frac{\Delta t}{\Delta x} (v_J^{m+1} - v_{J-1}^{m+1}).$$

By assumption, $c > 0$, and by the choice of J we have $v_J^{m+1} - v_{J-1}^{m+1} \geq 0$. Iterating the inequality over all m yields

$$v_j^m \leq \max_{j=0, \dots, n+1} v_j^0 \leq \sup_{x \in [0,1]} g(x).$$

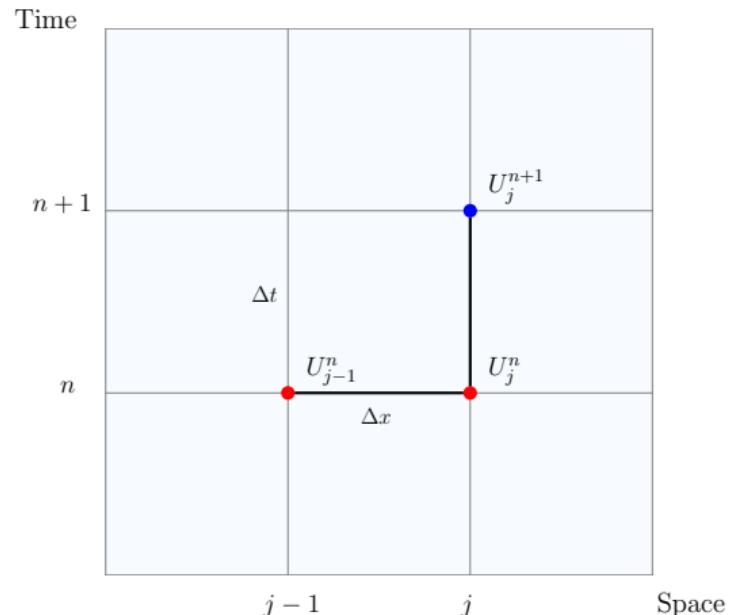
Fourier stability analysis

Example (First-Order Upwind (FOU))

Proposed by Richard Courant, Eugene Isaacson and Mina Rees. Let $c(x, t) = c > 0$.

$$0 = \frac{U_j^{n+1} - U_j^n}{\Delta t} + c \frac{U_j^n - U_{j-1}^n}{\Delta x}.$$

$$U_j^{n+1} = U_j^n - r (U_j^n - U_{j-1}^n), \quad r = c \frac{\Delta t}{\Delta x}.$$



Fourier stability analysis

Theorem (Stability analysis for FOU scheme)

$$r \in (0, 1] \iff |\lambda(k)| \leq 1.$$

Proof.

$$U_j^{n+1} = U_j^n - r(U_j^n - U_{j-1}^n).$$

$$\lambda(k)^{n+1} e^{ik(j\Delta x)} = \lambda(k)^n e^{ik(j\Delta x)} - r\lambda(k)^n (e^{ik(j\Delta x)} - e^{ik(j-1)\Delta x}).$$

$$\lambda(k) = 1 - r(1 - e^{-ik\Delta x}).$$

$$\begin{aligned} |\lambda(k)| &= |1 - r + re^{-ik\Delta x}| \\ &\leq |1 - r| + |re^{-ik\Delta x}| \\ &= |1 - r| + r \\ &\leq 1 \iff r \in (0, 1]. \end{aligned}$$

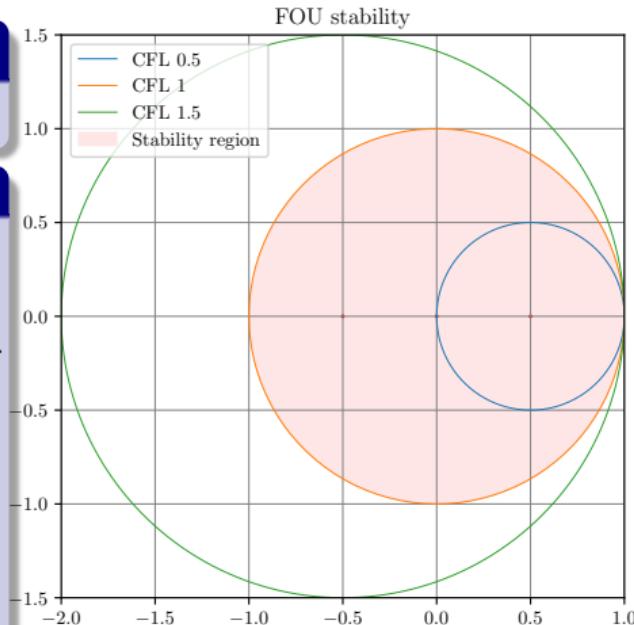


Figure: The FOU scheme is **dissipative**.

```

label = "FOU scheme for 1D linear advection PDE"
space = [0, 10]
spacepoints = [6]
speed = 5
time = [0, 1]
timesteps = 5
piecewise = [2, 4]

```

Program  : Parameters file `fou.toml`.

```

from tomllib import load
import numpy as np
with open(file="fou.toml", mode="rb") as f:
    data = load(f)
c = data["speed"]
x, Δx = np.linspace(
    start=data["space"][0],
    stop=data["space"][1],
    num=data["spacepoints"][0],
    retstep=True,
)
t, Δt = np.linspace(
    start=data["time"][0], stop=data["time"][1], num=data["timesteps"], retstep=True
)
cfl = c * Δt / Δx
u = np.where((data["piecewise"][0] ≤ x) & (data["piecewise"][1] ≤ 4), 1.0, 0)
u = np.insert(u, 0, u[0]) # left hand side ghost node
u = np.append(u, u[-1]) # right hand side ghost node
print(f"CFL number: {cfl}")
print(*u0      u1      u2      u3      u4      u5      u6*)
for timestep in t:
    print(u)
    u[1:] -= cfl * (u - np.roll(u, 1))[1:]
    u[0] = u[1]
    u[-1] = u[-2]

```

Program  : `fou.py`.

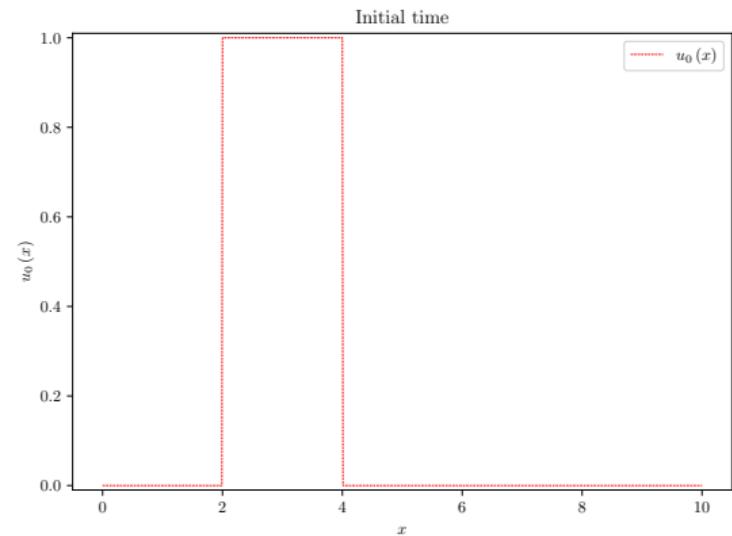
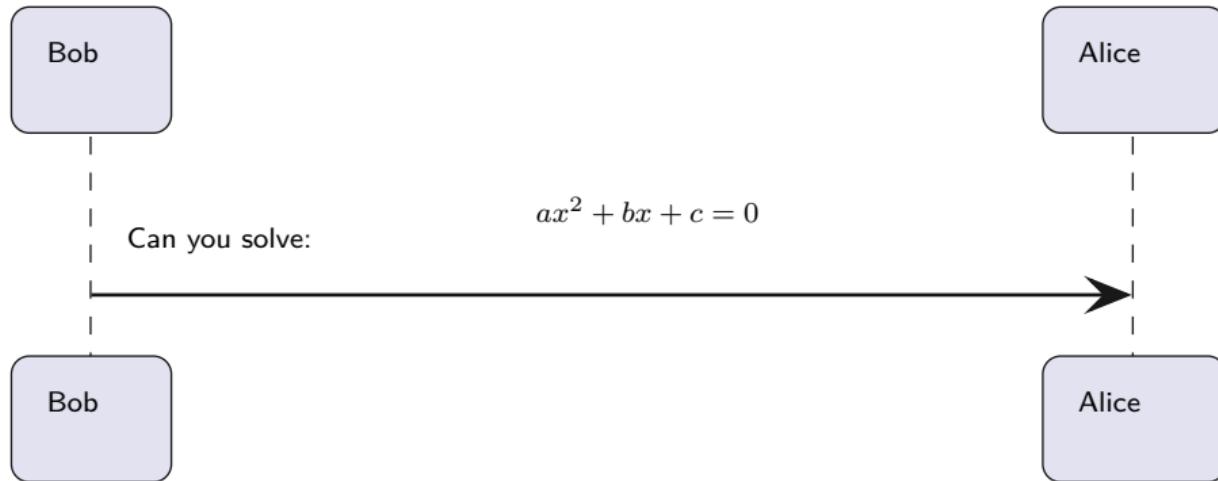


Figure: Initial profile.

CFL number: 0.625

u0	u1	u2	u3	u4	u5	u6
[0. 0. 1. 1. 1. 1. 1.]	[0. 0. 0.375 1. 1. 1. 1.]	[0. 0. 0.140625 0.609375 1. 1. 1.]	[0. 0. 0.052734 0.316406 0.755859 1. 1.]	[0. 0. 0.019775 0.151611 0.481201 0.847412 1.]		

Flow chart

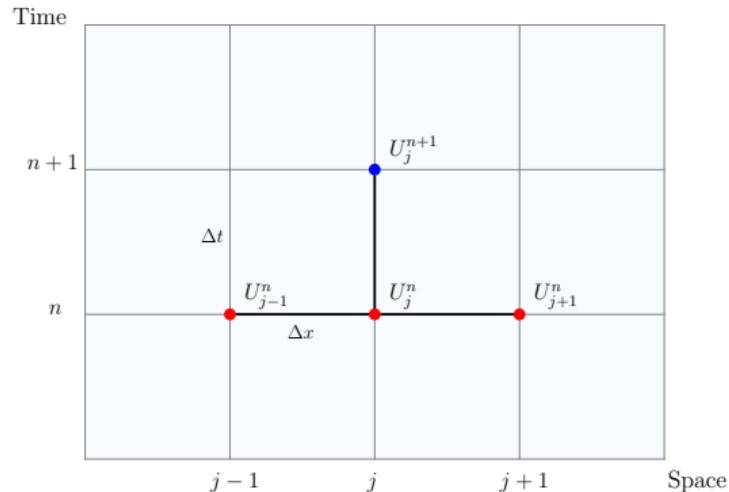


Fourier stability analysis

Example (Forward-Time Central-Space (FTCS))

$$0 = \frac{U_j^{n+1} - U_j^n}{\Delta t} + c \frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x}.$$

$$U_j^{n+1} = U_j^n - \frac{r}{2} (U_{j+1}^n - U_{j-1}^n), \quad r = c \frac{\Delta t}{\Delta x}.$$



Fourier stability analysis

Theorem (Stability analysis for FTCS scheme)

$$\forall r > 0 \iff |\lambda(k)| > 1.$$

Proof.

$$\begin{aligned} U_{j+1}^n &= U_j^n - \frac{r}{2} (U_{j+1}^n - U_{j-1}^n). \\ \lambda(k)^{n+1} e^{ik(j\Delta x)} &= \lambda(k)^n e^{ikj\Delta x} - \frac{r}{2} \lambda(k)^n (e^{ik(j+1)\Delta x} - e^{ik(j-1)\Delta x}). \\ \lambda(k) &= 1 - \frac{r}{2} (e^{ik\Delta x} - e^{-ik\Delta x}). \\ \lambda(k) &= 1 - ir \sin(k\Delta x). \\ |\lambda(k)|^2 &= 1 + r^2 \sin^2(k\Delta x) \\ &> 1 \iff \forall r > 0. \end{aligned}$$

□

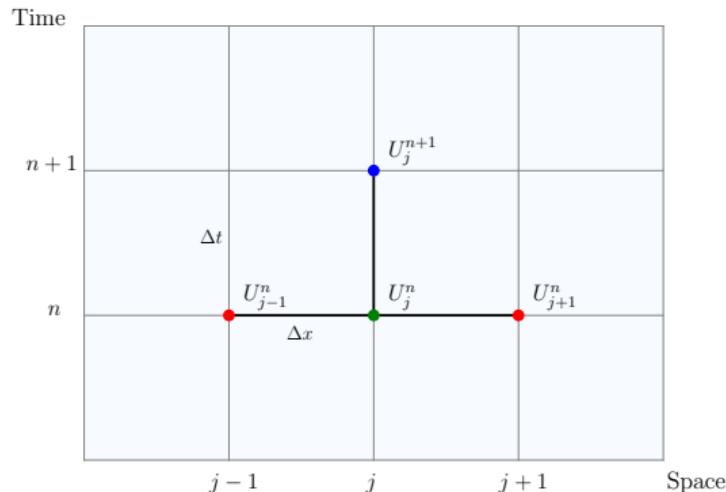
Remark

We say that the FTCS scheme is unconditionally unstable.

Fourier stability analysis

Example (Lax-Friedrichs scheme)

$$0 = \frac{U_j^{n+1} - \frac{U_{j+1}^n + U_{j-1}^n}{2}}{\Delta t} + c \frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x}.$$
$$U_j^{n+1} = \frac{U_{j+1}^n + U_{j-1}^n}{2} - \frac{r}{2} (U_{j+1}^n - U_{j-1}^n), \quad r = c \frac{\Delta t}{\Delta x}.$$



Fourier stability analysis

Theorem (Stability analysis for Lax-Friedrichs)

$$r \in (0, 1] \iff |\lambda(k)| \leq 1.$$

Proof.

$$U_j^{n+1} = \frac{U_{j+1}^n + U_{j-1}^n}{2} - \frac{r}{2} (U_{j+1}^n - U_{j-1}^n).$$

$$\lambda(k)^{n+1} e^{ik(j\Delta x)} = \frac{\lambda(k)^n}{2} (e^{ik(j+1)\Delta x} + e^{ik(j-1)\Delta x}) - \frac{r\lambda(k)^n}{2} (e^{ik(j+1)\Delta x} - e^{ik(j-1)\Delta x}).$$

$$\lambda(k) = \frac{e^{ik\Delta x} + e^{-ik\Delta x}}{2} - r \frac{e^{ik\Delta x} - e^{-ik\Delta x}}{2}.$$

$$\lambda(k) = \cos(k\Delta x) - ir \sin(k\Delta x).$$

$$|\lambda(k)|^2 = \cos^2(k\Delta x) + r^2 \sin^2(k\Delta x).$$

$$|\lambda(k)|^2 = 1 - \sin^2(k\Delta x) + r^2 \sin^2(k\Delta x).$$

$$|\lambda(k)|^2 = 1 - (1 - r^2) \sin^2(k\Delta x).$$

$$\leq 1 \iff r \in (0, 1].$$

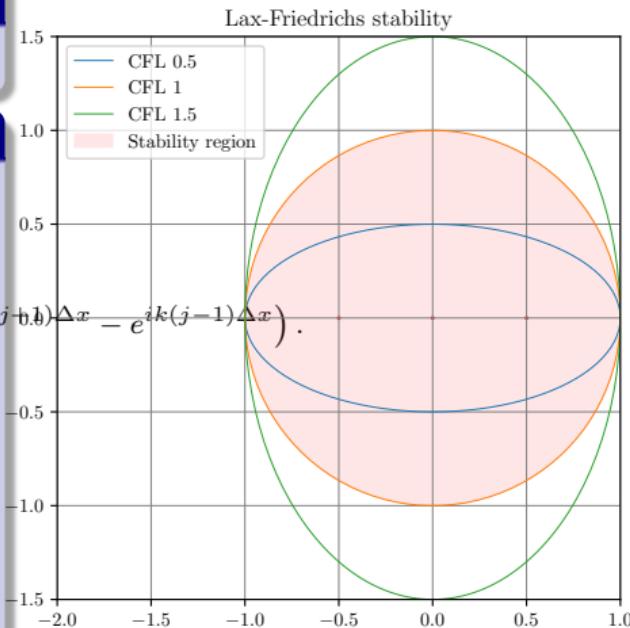


Figure: The Lax-Friedrichs scheme is oscillates.

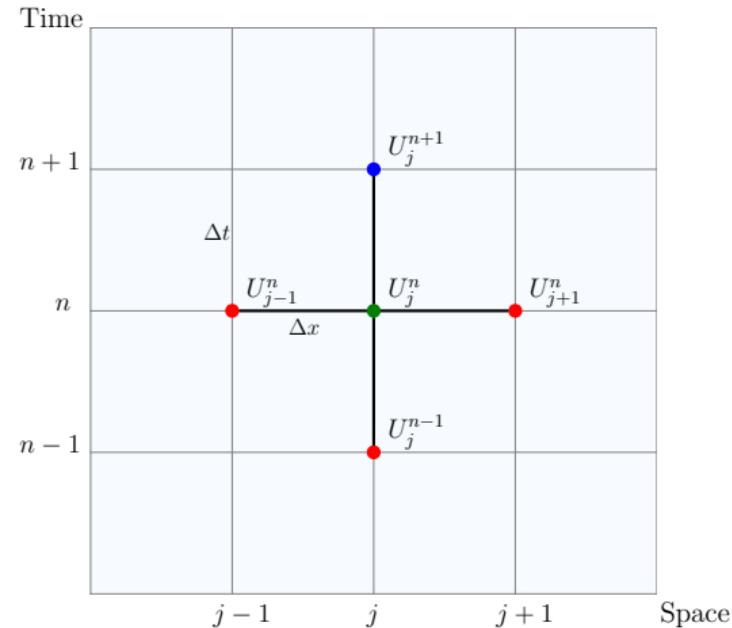


Fourier stability analysis

Example (Leapfrog scheme)

$$0 = \frac{U_j^{n+1} - U_j^{n-1}}{2\Delta t} + c \frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x}.$$

$$U_j^{n+1} = U_j^{n-1} - \frac{c\Delta t}{\Delta x} \frac{U_{j+1}^n - U_{j-1}^n}{2}.$$



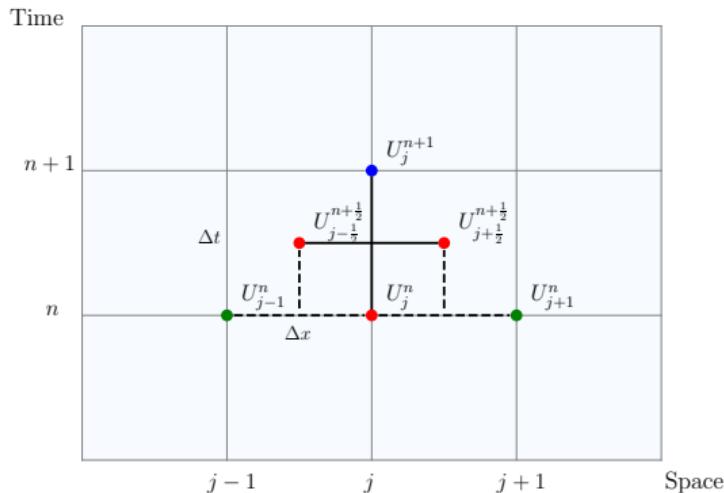
$$|U_j^{n+1}| = \left| U_j^{n-1} - r \frac{U_{j+1}^n - U_{j-1}^n}{2} \right| = \left| 0 - r \frac{e^{ikj\Delta x} e^{ik\Delta x} - e^{ikj\Delta x} e^{-ik\Delta x}}{2} \right|.$$

Fourier stability analysis

Example (Lax-Wendroff scheme)

$$0 = \frac{U_i^{n+1} - U_i^{n-1}}{2\Delta t} + c \frac{U_{i+1}^n - U_{i-1}^n}{2\Delta x}.$$

$$U_j^{n+1} = U_j^n - \frac{c\Delta t}{\Delta x} \left(U_{j+\frac{1}{2}}^{n+\frac{1}{2}} - U_{j-\frac{1}{2}}^{n+\frac{1}{2}} \right).$$

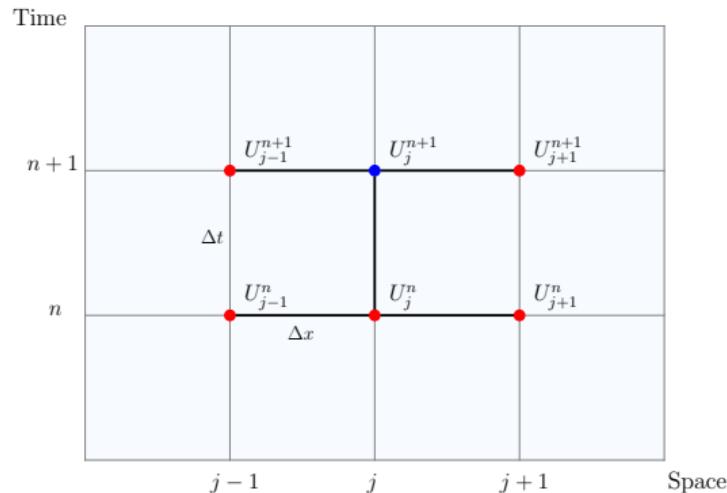


Fourier stability analysis

Example (Crank-Nicolson scheme)

$$0 = \frac{U_i^{n+1} - U_i^{n-1}}{2\Delta t} + c \frac{U_{i+1}^n - U_{i-1}^n}{2\Delta x}.$$

$$U_j^{n+1} = U_j^n - \frac{c\Delta t}{\Delta x} \frac{U_{j+1}^n - U_{j-1}^n + U_{j+1}^{n+1} - U_{j-1}^{n+1}}{4}.$$



Theorem (Lax-Richtmyer equivalence)

A consistent finite-difference scheme for a PDE for which the initial-value problem is well posed is convergent iff it is stable.

Fourier stability analysis

Since it underlies the mass conservation law, the advection equation

$$\partial_t u + \nabla \cdot (cu) = 0$$

is said to be in **conservative form** (also called **conservation** or **flux form**).

From the identity

$$\nabla \cdot (cu) = u \nabla \cdot c + c \cdot \nabla u,$$

$$\partial_t u + a \cdot \nabla u = 0$$

if the vector field c is divergence-free, that is if

$$\nabla \cdot c = 0.$$

If we define the characteristics $(\xi(t), t)$ in the (x, t) space of solutions of the ODE

$$\frac{d\xi(t)}{dt} = c(\xi(t), t),$$

then it follows that

$$\frac{du(\xi(t), t)}{dt} = 0,$$

and hence the solution $u(x, t)$ is constant along the characteristics.