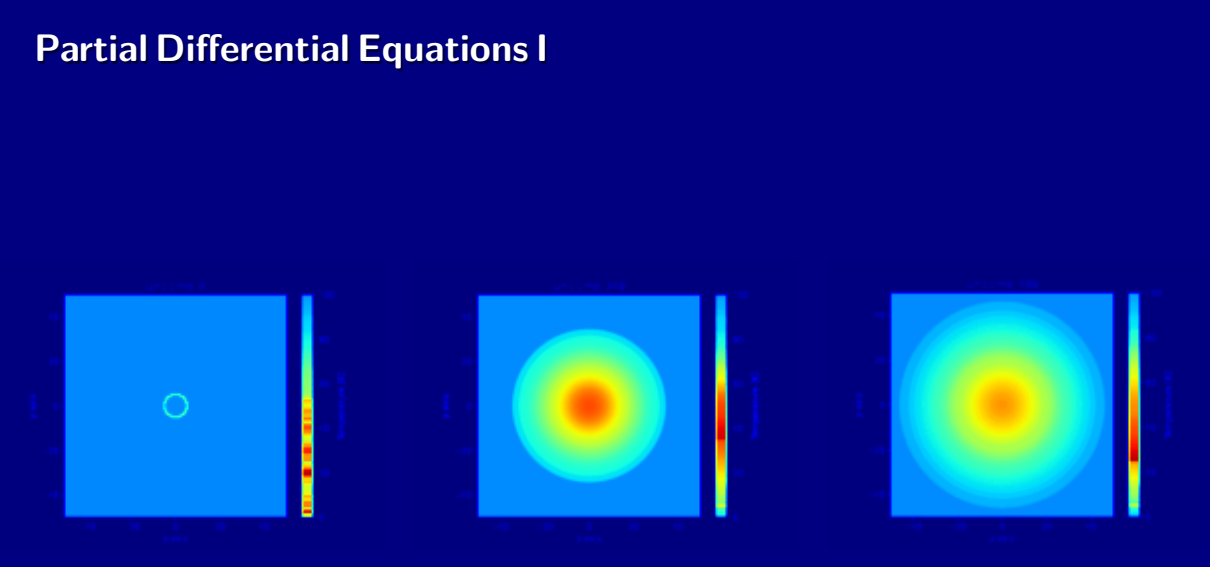
Partial Differential Equations I



Carlos Aznarán Laos

Last change: October 28, 2024 at 12:15am.

Useful links

Click on each vignette for updated resources or the book cover on next slides.

- Pad + general information
- Meeting link `Mon, Fri 09:00:00 PM -05`
- Beamer slides + Report lecture
- Analytical methods for solve the wave equation (1D, 2D and 3D) course + books
- Live recordings + Jason Bramburger's lectures
- Repository
- Animations with matplotlib
- Shared folder + exercises

Remark

We'll try to follow this outline https://math.dartmouth.edu/~m53f22.

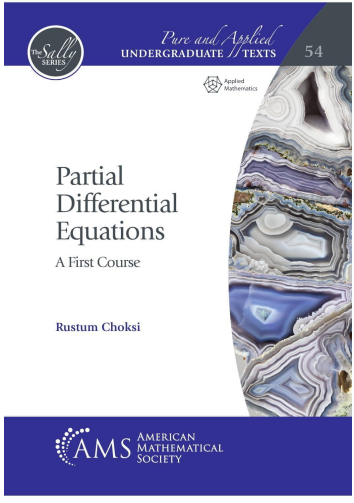
VisualPDE

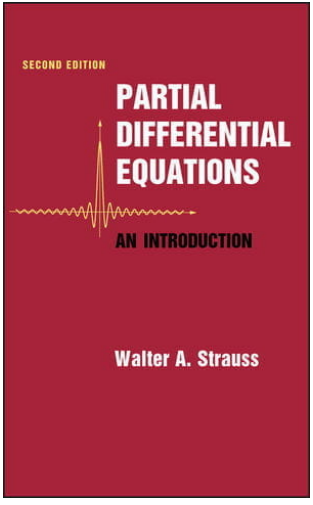
Every time we explore a new PDE we are likelihood to visualize the animation on https://visualpde.com.

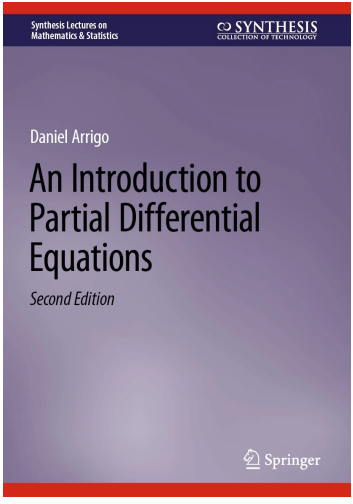
Universal document viewer

Okular is a PDF viewer that allows interaction with forms, e.g., display animations of time dependent PDE solutions.

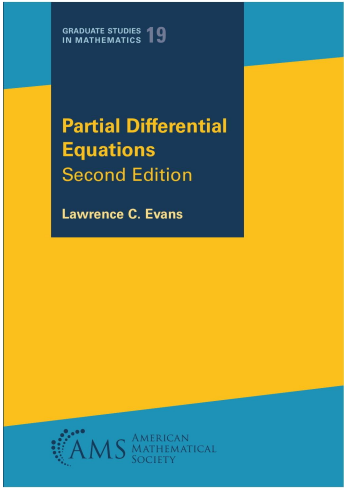
References with foundations on ODE

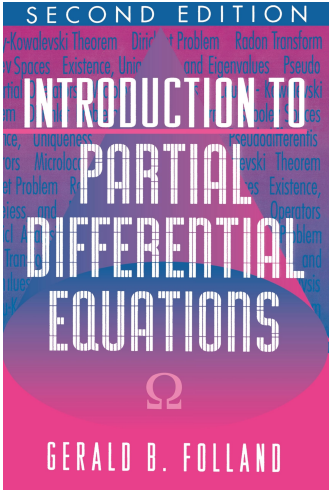


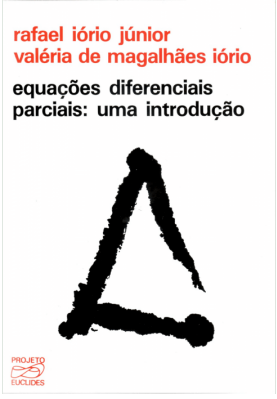




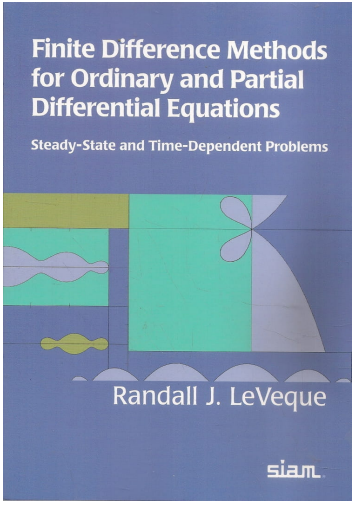
References with foundations on Functional Analysis

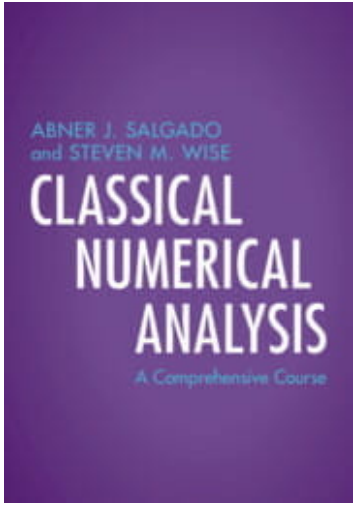


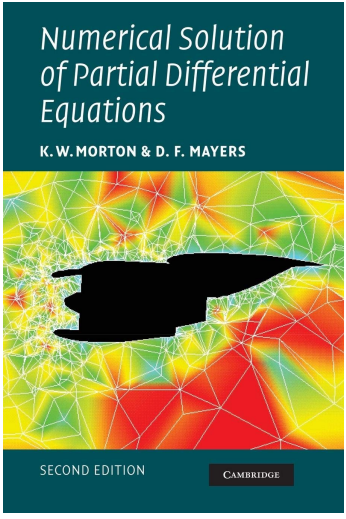




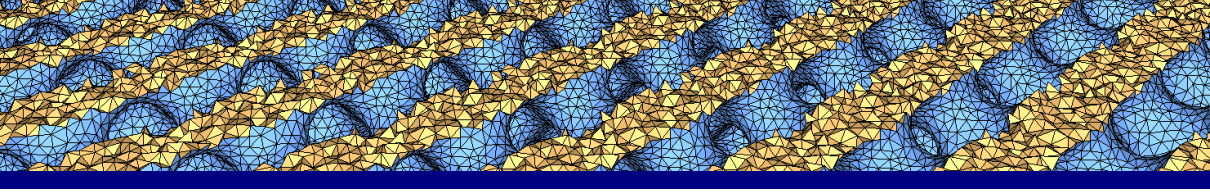
References with foundations on Numerical Analysis







Contents



Review of ODEs

Review of ODEs

An ordinary differential equation (ODE) is a *functional equation* that relates some function with its derivatives.

Example (Classification of ODEs)

- Heterogeneous first-order linear constant coefficient.

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \pi u + \cos(x).$$

- Homogeneous second-order linear.

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} - x\frac{\mathrm{d}u}{\mathrm{d}x} + u = 0.$$

- Homogeneous second-order linear constant coefficient.

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + \alpha^2 u = 0.$$

- Heterogeneous first-order nonlinear.

$$\frac{\mathrm{d}u}{\mathrm{d}x} = u^5 + 1.$$

Review of ODEs

For functions of several variables, an ODE becomes in a PDE.

Example (PDE models)

- Models the concentration of a substance flowing in a fluid at a constant rate $c \in \mathbb{R} \setminus \{0\}$.

  (Advection) $$\partial_t u + c\partial_x u = 0.$$

  Its general solution is $u(x,t) = \phi(x - ct)$ where ϕ is an arbitrary function.
- Type of propagating disturbance that moves faster than the speed of sound in a medium.

  (Shock waves) $$\partial_x u + u\partial_y u = 0.$$

  Like a common wave, a shock wave carries energy and can propagate through a medium, but is characterized by an abrupt, almost discontinuous change in the pressure, temperature, and density of the medium.
- Models the constant heat flow in a region where the temperature is fixed at the boundary.

  (Laplace) $$\triangle u = 0.$$

More classifications of differential equations

- An integro-differential equation involving both the derivatives and its anti-derivatives of a solution.

(RLC circuit)

$$L\frac{\mathrm{d}I(t)}{\mathrm{d}t} + RI(t) + \frac{1}{C}\int_0^t I(\tau)\,\mathrm{d}\tau = E(t).$$

- A functional differential equation with deviating argument and more applicable than ODEs.

(Population growth)

$$\frac{\mathrm{d}u(t)}{\mathrm{d}t} = \rho u(t)\left(1 - \frac{u(t-\tau)}{k}\right).$$

- A stochastic differential equation is composed in terms of stochastic process.

(Arithmetic Brownian motion)

$$\mathrm{d}X_t = \mu\,\mathrm{d}t + \sigma\,\mathrm{d}B_t.$$

- A differential algebraic equation involves differential and algebraic terms.
- Stiff PDE, Delay PDE, Controlled PDE, Fractional PDE, Neural PDE and so on.

Review of ODEs

Let the IVP

$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}t} & = -\frac{u}{2}, \quad t \in [0, 10]\,. \\ u\,(0) & = a_i, \end{cases}$$

where $a_1 = 2$, $a_2 = 4$, $a_3 = 6$ and $a_4 = 8$.

```python
import numpy as np
from jaxtyping import Array, Float
from scipy.integrate import solve_ivp


def exponential_decay(
    t: Float[Array, "dim"], u: Float[Array, "dim"]
) -> Float[Array, "2"]:
    return -0.5 * u
```

```python
sol = solve_ivp(
    fun=exponential_decay,
    t_span=(0, 10),
    y0=(2, 4, 6, 8),
    t_eval=np.linspace(start=0, stop=10),
    dense_output=True,
)
```

Program : Recovered from https://docs.scipy.org/doc/scipy-1.14.1/reference/generated/scipy.integrate.solve_ivp.html.

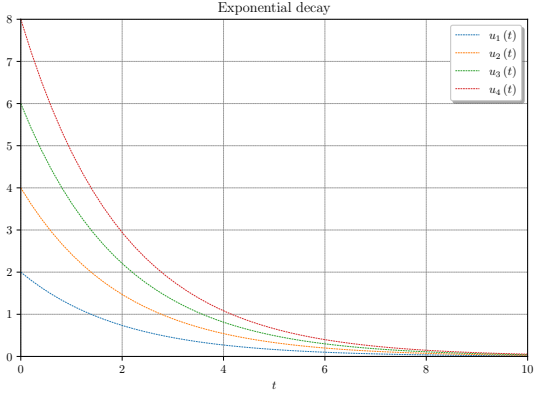


Figure: Numerical solution.

Review of ODEs

The BVP

$$\frac{\mathrm{d}u}{\mathrm{d}x} + \exp(u) = 0, \quad u(0) = u(1) = 0.$$

```python
import numpy as np
from jaxtyping import Array, Float
from scipy.integrate import solve_bvp


def fun(x: Float[Array, "dim"], u: Float[Array, "2"]) -> Float[Array, "2"]:
    return np.vstack((u[1], -np.exp(u[0])))


def bc(ua: float, ub: float) -> Float[Array, "2"]:
    return np.array([ua[0], ub[0]])


x = np.linspace(start=0, stop=1, num=5)
u_a = np.zeros(shape=(2, x.size))
u_b = np.copy(a=u_a)
u_b[0] = 3

sol_a = solve_bvp(fun=fun, bc=bc, x=x, y=u_a)
sol_b = solve_bvp(fun=fun, bc=bc, x=x, y=u_b)
```

Program : Recovered from https://docs.scipy.org/doc/scipy-1.14.1/reference/generated/scipy.integrate.solve_bvp.html.

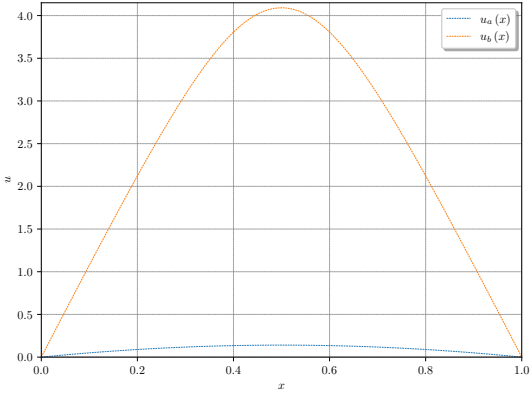


Figure: Numerical solution.

Theorem (Existence and Uniqueness of solutions - Picard-Lindelöf)

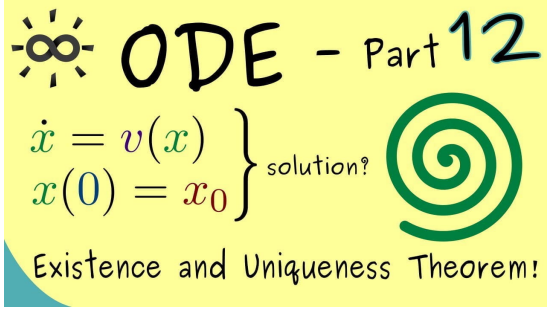
Consider the initial value problem

(1)
$$\begin{cases} \frac{\mathrm{d}u}{\mathrm{d}x} = f(x,u), \\ u(\xi) = \eta. \end{cases}$$

Here it is assumed that $f(\cdot,\cdot)$ is continuous on $[\xi,\xi+a]\times\mathbb{R}$ where $a>0$, and furthermore satisfies

(Lipschitz condition)
$$|f(x,u)-f(x,\overline{u})| \leq L\,|u-\overline{u}|$$

for some $L\in\mathbb{R}_{\geq 0}$; here all $x\in[\xi,\xi+a]$, u, $\overline{u}\in\mathbb{R}$ are allowed. Then (1) admits precisely one C^1-solution $u(x)$ on $[\xi,\xi+a]$.



Idea of proof.

1. Formulation as a fixed point problem.

$$u(x) = \eta + \int_{\xi}^{x} f(t, u(t)) \, \mathrm{d}t.$$

2. Introduction of a Banach space, verifying contraction property.

$$\begin{aligned} T \colon C^0(I_b) &\longrightarrow C^0(I_b) \\ u &\longmapsto \eta + \int_{\xi}^{x} f(t, u(t)) \, \mathrm{d}t. \end{aligned}$$

3. Application of Contraction Principle, construction of local solution.

□

Theorem (Peano)

For $I = [\xi, \xi + a]$, $J = [\eta - b, \eta + b]$, *we have* $f \in C^0(I \times J), |f|_{C^0(I \times J)} \leq M$ *for some* $M, a, b > 0$, *there exists a solution* $u(x) \in C^1\left(\left[\xi, \xi + \min\left\{a, \frac{b}{M+1}\right\}\right]\right)$.

Idea of proof.

1. The idea is to reduce to the situation in Picard's theorem.
2. The mollification of f is now given by the family of functions.

$$f_\varepsilon(x, u) := f *_u \chi_\varepsilon(x, u) = \int_{\mathbb{R}} f(x, u - z)\, \chi_\varepsilon(z)\, \mathrm{d}z.$$

3. In order to be able to invoke the version of Picard's theorem, we need to extend $f_\varepsilon(x, u)$ to all $\mathbb{R}$.

$$|f_\varepsilon(x, u) - f_\varepsilon(x, \overline{u})| \leq \frac{C}{\varepsilon} M |u - \overline{u}|.$$

4. Use the Arzelà-Ascoli theorem.

□

Separable equation

If the right hand side of the equation

$$\frac{\mathrm{d}u}{\mathrm{d}x} = g\left(x\right) p\left(u\right)$$

can be expressed as function $g\left(x\right)$ that depends only of x times a function $p\left(u\right)$ that depends only on u, the differential equation is called separable.

Example (Separable equation)

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{x-5}{u^2}.$$

Solution

$$u^2\,\mathrm{d}u = (x-5)\,\mathrm{d}x.$$

$$\int u^2\,\mathrm{d}u = \int (x-5)\,\mathrm{d}x.$$

$$\frac{u^3}{3} = \frac{x^2}{2} - 5x + C \implies u\left(x\right) = \left(\frac{3x^2}{2} - 15x + K\right)^{\frac{1}{3}}.$$

Linear equation

In order to solve the ODE in the standard form

$$\frac{\mathrm{d}u}{\mathrm{d}x} + P(x)\,u(x) = Q(x)\,. \tag{2}$$

Calculate the integrating factor $\mu(x)$ by

$$\mu(x) = \exp\left[\int P(x)\ \mathrm{d}x\right]. \tag{3}$$

And multiply (2) by (3)

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\mu(x)\,u(x)\right] = \mu(x)\,Q(x)\,.$$

And obtain the solution

$$u(x) = \frac{1}{\mu(x)}\left[\int \mu(x)\,Q(x)\ \mathrm{d}x + C\right].$$

Example (Linear equation)

$$\frac{\mathrm{d}u}{\mathrm{d}x} + 2u(x) = 50\exp(-10x)\,.$$

Homogeneous linear second order ode

Let be $a \in \mathbb{R} \setminus \{0\}$.

$$a\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + b\frac{\mathrm{d}u}{\mathrm{d}x} + cu = 0.$$

Find a solution of the form $u(x) = e^{rx}$.

$$ar^2e^{rx} + bre^{rx} + ce^{rx} = 0.$$

$$e^{rx}\left(ar^2 + br + c\right) = 0.$$

Since $e^{rx} > 0$

$$ar^2 + br + c = 0.$$

Example (Homogeneous linear second order)

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + 5\frac{\mathrm{d}u}{\mathrm{d}x} - 6u = 0.$$

Solution

$$r^2 + 5r - 6 = (r-1)(r+6) = 0.$$

e^x and e^{-6x} are solutions.

Techniques to solve Second order ODEs

Example ( )

$$\frac{\mathrm{d}^3u}{\mathrm{d}x^3} + 3\frac{\mathrm{d}^2u}{\mathrm{d}x^2} - \frac{\mathrm{d}u}{\mathrm{d}x} - 3u = 0.$$

Nonhomogeneous

$$a\frac{\mathrm{d}^2u}{\mathrm{d}x^2}+b\frac{\mathrm{d}u}{\mathrm{d}x}+cu=f\left(x\right).$$

Example (Nonhomogeneous)

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}+3\frac{\mathrm{d}u}{\mathrm{d}x}+2u=3x.$$

Method of Variation of Parameters

$$a\frac{\mathrm{d}^2u}{\mathrm{d}x^2}+b\frac{\mathrm{d}u}{\mathrm{d}x}+cu=f\left(x\right).$$

$$u_h\left(x\right)=c_1u_1\left(x\right)+c_2u_2\left(x\right).$$
$$u_p\left(x\right)=v_1\left(x\right)y_1\left(x\right)+v_2\left(x\right)y_2\left(x\right).$$

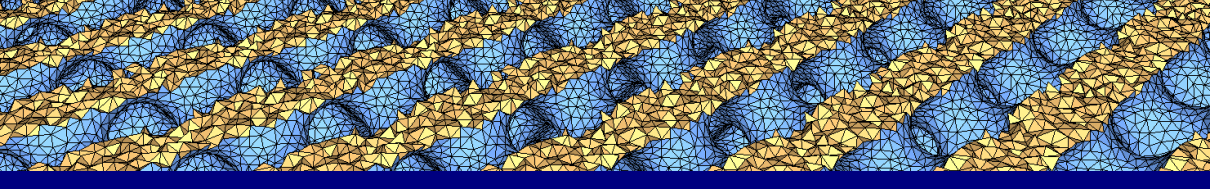
Example ( )

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}+u=\tan x.$$

Solution

The homogeneous equation $\frac{\mathrm{d}^2u}{\mathrm{d}x^2}+u=0$ *are* $\cos x$ *and* $\sin x$.

$$\begin{aligned}u_p\left(x\right)&=v_1\left(x\right)\cos\left(x\right)+v_2\left(x\right)\sin\left(x\right).\\v_1\left(x\right)&=\sin\left(x\right)-\ln\left|\sec x+\tan x\right|+C_1.\\v_2\left(x\right)&=-\cos x+C_2.\\u\left(x\right)&=c_1\cos x+c_2\sin x-\left(\cos x\right)\ln\left(\sec x+\tan x\right).\end{aligned}$$



Using Python to Solve PDEs

Symbolic computing in Python

- Library for symbolic mathematics.
- It aims to become a full-featured computer algebra system while keeping the code as simple as possible in order to be comprehensible and easily extensible.

Recovered from `https://www.sympy.org`.

- 2 hours tutorial by Aaron Meurer: `https://youtu.be/FZWevQ6Xz6U?t=3059`.
- Featured modules: `sympy.calculus`, `sympy.integrals`, `sympy.series.fourier`, `sympy.solvers.ode`, and `sympy.solvers.pde`.
- Latest version `1.13.3` dated September 18, 2024.

Symbolic computing in Python

```python
from sympy import Derivative as D
from sympy.abc import U, c, k, r, theta, x, y
from sympy.core import Eq, Function, I, Symbol, pi, var
from sympy.functions import exp
from sympy.integrals import fourier_transform, inverse_fourier_transform

u = Function("u")

laplace = Eq(lhs=D(U, x, 2) + D(U, y, 2), rhs=0)

Δt = Symbol("Δt")
Δx = Symbol("Δx")
u = Function("u")
taylor = Eq(u(x + Δx), u(x + Δx).series(x=Δx, x0=0, n=3).simplify())

expresion1 = (exp(I * theta) + exp(-I * theta)) / 2
expresion2 = (exp(I * theta) - exp(-I * theta)) / (2 * I)
expresion3 = (exp(I * theta) - exp(-I * theta)) / 2

Unp1j = var("U^{n+1}_j")
Unj = var("U^n_j")
Unjm1 = var("U^n_{j-1}")
FOU = Eq(
    Unp1j,
    solve(((Unp1j - Unj) / Δt + c * (Unj - Unjm1) / Δx).subs({Δt: r * Δx / c}), Unp1j)[
        0
    ],
)

a = fourier_transform(f=exp(-(x**2)), x=x, k=k)

b = inverse_fourier_transform(F=a, k=k, x=x)

s = fourier_series(f=x**2, limits=(x, -pi, pi))
iterator = s.truncate(n=None)
for n in range(10):
    term = next(iterator)
    print(f"a_{n}:")
    pprint(term.subs(x, 0))
```

```
Laplace equation:
 2         2
d         d
──(U) + ──(U) = 0
  2         2
dx        dy
Taylor expansion:
                                        2
                                   2 d
                                  Δx ⋅──(u(x))
                                        2
                                  d    dx            ⎛ 3⎞
u(x + Δx) = u(x) + Δx⋅──(u(x)) + ──────────── + O⎝Δx ⎠
                                  dx     2

Simplify:
cos(θ)
sin(θ)
ⅈ⋅sin(θ)

Solve equation:
U^{n+1}_j = -U^n_j*r + U^n_j + U^n_{j-1}*r

Laplace transform:
      2  2
    -π ⋅k
√π⋅ℯ

Inverse Laplace transform:
   2
 -x
ℯ
```

py-pde: A Python package for solving partial differential equation

- Contains classes for grids on which scalar and tensorial fields can be defined. The associated differential operators are computed using a implementation of finite differences.

  Recovered from `https://www.zwickergroup.org/software`.

- 60 minutes tutorial by me: `https://youtu.be/2xnK2ubFAto?t=273`.

- Highlights three useful modules: `pde.pdes.laplace`, `pde.pdes.diffusion` and `pde.pdes.wave`.

- Latest version `0.41.0` dated August 5, 2024.

py-pde: A Python package for solving partial differential equation

```python
from math import pi

from pde import CartesianGrid, solve_laplace_equation

res = solve_laplace_equation(
    grid=CartesianGrid(bounds=[[0, 2 * pi]] * 2, shape=2**8),
    bc={"value": "sin(y)"}, {"value": "sin(x)"}),
)
```

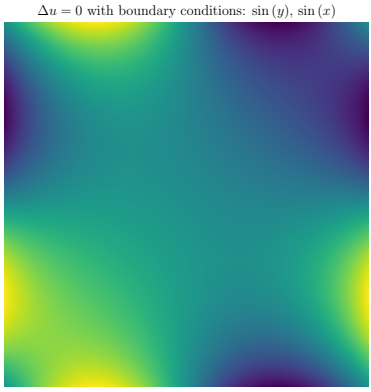


Figure: Solution of Laplace equation using the methods of lines in a rectangular grid.

DOLFINx: Next generation FEniCS problem solving environment

- The FEniCS project is a research and software project aimed at creating mathematical methods and software for solving PDEs.
- The latest version of the FEniCS project, FEniCSx, consists of several building blocks, namely DOLFINx, UFL, FFCx, and Basix.
- DOLFINx, being a state of the art finite element solver.

Recovered from `https://jsdokken.com/dolfinx-tutorial`.

- 45 minutes tutorial by Antonio Baiano Svizzero: `https://youtu.be/uQW2wSDtW5k`.
- Highlights five useful modules: `dolfinx.mesh.create_rectangle`, `dolfinx.fem.functionspace`, `dolfinx.mesh.locate_entities_boundary`, `ufl.TestFunction` and `ufl.TrialFunction`.
- Latest version `0.9.0.post0` dated October 9, 2024.

DOLFINx: Next generation FEniCS problem solving environment

```python
import numpy as np
from dolfinx.fem import dirichletbc, functionspace, locate_dofs_topological
from dolfinx.fem.petsc import LinearProblem
from dolfinx.io import VTKFile
from dolfinx.mesh import CellType, create_rectangle, locate_entities_boundary
from mpi4py import MPI
from petsc4py.PETSc import ScalarType
from ufl import (
    SpatialCoordinate,
    TestFunction,
    TrialFunction,
    ds,
    dx,
    exp,
    grad,
    inner,
    sin,
)

msh = create_rectangle(
    comm=MPI.COMM_WORLD,
    points=((0.0, 0.0), (2.0, 1.0)),
    n=(32, 16),
    cell_type=CellType.triangle,
)
```

```python
V = functionspace(mesh=msh, element=("Lagrange", 1))
facets = locate_entities_boundary(
    msh=msh,
    dim=1,
    marker=lambda x: np.logical_or(np.isclose(x[0], 0.0), np.isclose(x[0], 2.0)),
)

dofs = locate_dofs_topological(V=V, entity_dim=1, entities=facets)
bc = dirichletbc(value=ScalarType(0), dofs=dofs, V=V)

u = TrialFunction(function_space=V)
v = TestFunction(function_space=V)
x = SpatialCoordinate(domain=msh)
f = 10 * exp(-((x[0] - 0.5) ** 2 + (x[1] - 0.5) ** 2) / 0.02)
g = sin(f=5 * x[0])
a = inner(a=grad(u), b=grad(v)) * dx
L = inner(a=f, b=v) * dx + inner(a=g, b=v) * ds

problem = LinearProblem(
    a=a, L=L, bcs=[bc], petsc_options={"ksp_type": "preonly", "pc_type": "lu"}
)
uh = problem.solve()
```

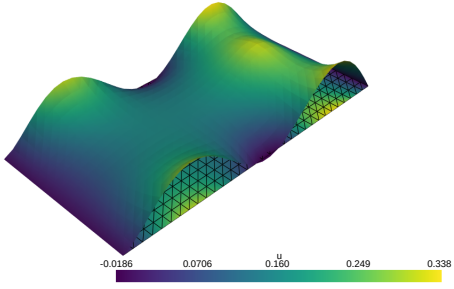


Figure: Warp By Scalar filter over solution.

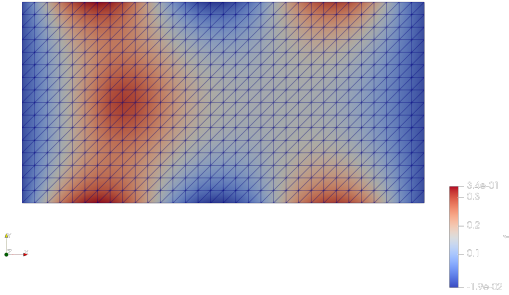


Figure: Solution.

Conservation Laws Package (Clawpack)

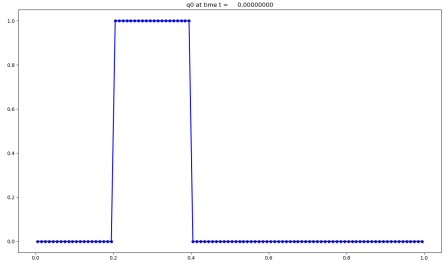
- Is a collection of finite volume methods for linear and nonlinear hyperbolic systems of conservation laws.
- Employs high-resolution Godunov-type methods with limiters in a general framework applicable to many kinds of waves.

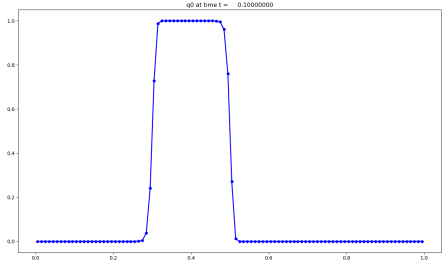
Recovered from `https://www.clawpack.org`.

- 15 minutes tutorial by Felix Köhler: `https://youtu.be/tr348El2A4Q`.
- Latest version `5.11.0` dated August 26, 2024.
- PyClaw: Python version of the hyperbolic PDE solvers that allows solving the problem in Python without explicitly using any Fortran code.
- Friendly tutorial: `https://en.ancey.ch/cours/doctorat/tutorial_clawpack.pdf`
- Randall J. LeVeque and et.al wrote two books around this package:
  - *Finite Volume Methods for Hyperbolic Problems* (2002).
  - Riemann Problems and Jupyter Solutions (2020).

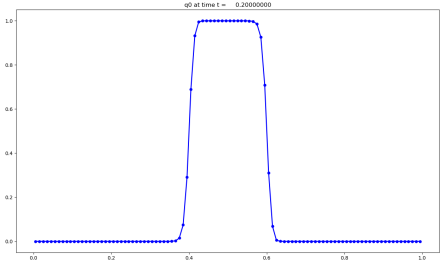
Conservation Laws Package (Clawpack)

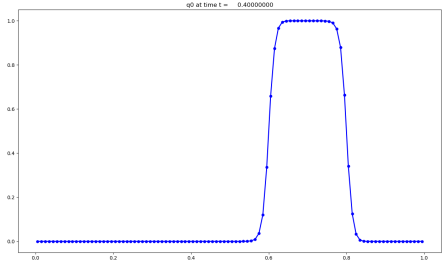
```python
import numpy as np
from clawpack.pyclaw import BC, ClawSolver1D, Controller, Dimension, Domain, Solution
from clawpack.pyclaw.plot import interactive_plot
from clawpack.riemann import advection_1D

# the library
solver = ClawSolver1D(riemann_solver=advection_1D)
solver.bc_lower[0] = BC.periodic
solver.bc_upper[0] = BC.periodic


x_dimension = Dimension(lower=0.0, upper=1.0, num_cells=100)
domain = Domain(x_dimension)


solution = Solution(solver.num_eqn, domain)


state = solution.state
cell_center_coordinates = state.grid.p_centers[0]
state.q[0, :] = np.where(
    (cell_center_coordinates > 0.2) & (cell_center_coordinates < 0.4),
    1.0,
    0.0,
)


state.problem_data["u"] = 1.0


controller = Controller()
controller.solution = solution
controller.solver = solver
controller.tfinal = 1.0
```

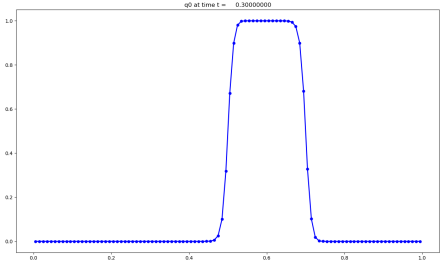


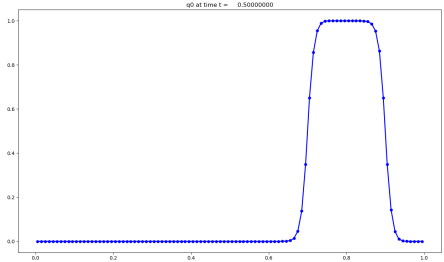


Conservation Laws Package (Clawpack)

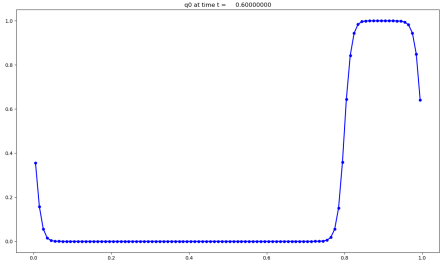


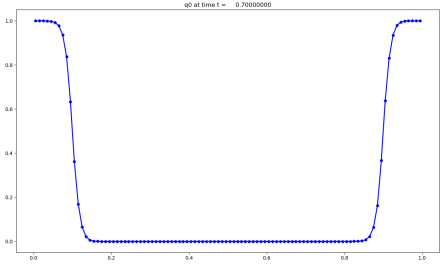


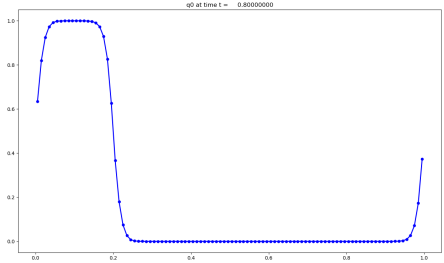




Conservation Laws Package (Clawpack)







Portable, Extensible Toolkit for Scientific Computation (PETSc)

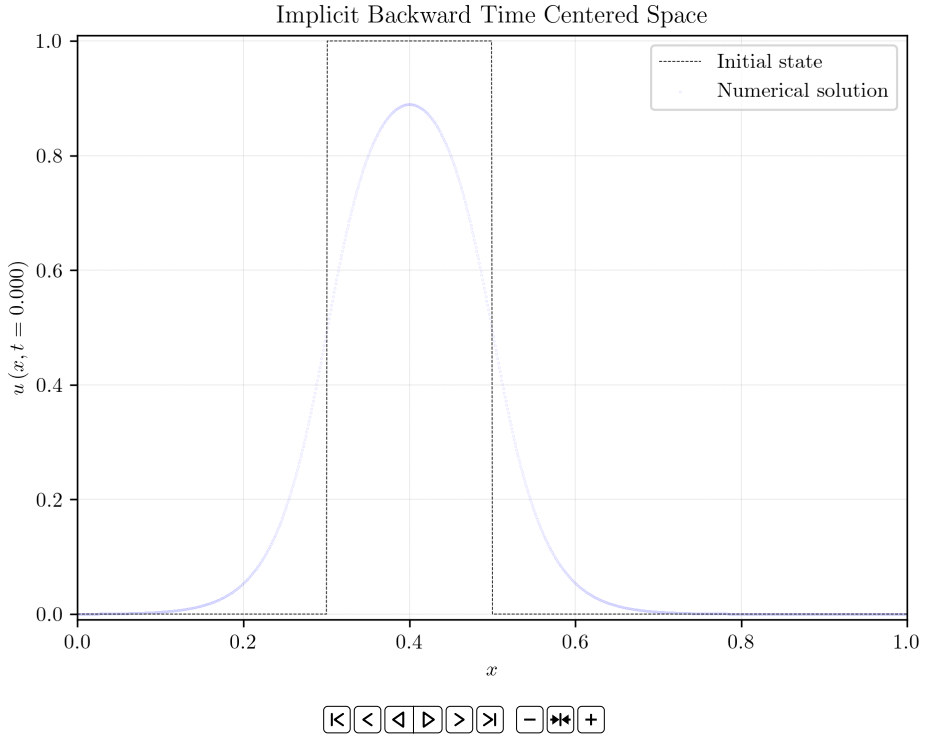
- is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations.
- It employs the MPI standard for all message-passing communication.

Recovered from `https://petsc.org/release/petsc4py`.

- 25 minutes tutorial by Felix Köhler: `https://youtu.be/oqxPyRZKOu4`.
- Latest version `3.22.0` dated September 28, 2024.
- Demo: Solve a constant coefficient Poisson problem on a regular grid.
- Book: PETSc for Partial Differential Equations: Numerical Solutions in C and Python (2020).

Portable, Extensible Toolkit for Scientific Computation (PETSc)

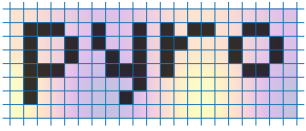
```python
import numpy as np
import petsc4py

from petsc4py.PETSc import KSP, Mat, Vec

N_POINTS = 1001
TIME_STEP_LENGTH = 0.001
N_TIME_STEPS = 100
mesh, Δx = np.linspace(start=0.0, stop=1.0, num=N_POINTS, retstep=True)

# Create a new sparse PETSc matrix, fill it and then assemble it
A = Mat().createAIJ([N_POINTS, N_POINTS])
A.setUp()

diagonal_entry = 1.0 + 2.0 * TIME_STEP_LENGTH / Δx**2
off_diagonal_entry = -1.0 * TIME_STEP_LENGTH / Δx**2

A.setValue(0, 0, 1.0)
A.setValue(N_POINTS - 1, N_POINTS - 1, 1.0)

for i in range(1, N_POINTS - 1):
    A.setValue(i, i, diagonal_entry)
    A.setValue(i, i - 1, off_diagonal_entry)
    A.setValue(i, i + 1, off_diagonal_entry)

A.assemble()

# Define the initial condition
initial_condition = np.where(
    (mesh > 0.3) & (mesh < 0.5),
    1.0,
    0.0,
)

# Assemble the initial rhs to the linear system
b = Vec().createSeq(N_POINTS)
b.setArray(initial_condition)
b.setValue(0, 0.0)
b.setValue(N_POINTS - 1, 0.0)

# Allocate a PETSc vector storing the solution to the linear system
x = Vec().createSeq(N_POINTS)

# Instantiate a linear solver: Krylov subspace linear iterative solver
ksp = KSP().create()
ksp.setOperators(A)
ksp.setFromOptions()

chosen_solver = ksp.getType()
print(f"Solving with {chosen_solver:}")

def animate(ti):
    print(f"Frame: {ti + 1}")
    plt.clf()
    plt.plot(
        mesh,
        initial_condition,
        color="black",
        label="Initial state",
        linewidth=0.4,
        linestyle="dashed",
    )
    ksp.solve(b, x)

    # Re-assemble the rhs to move forward in time
    current_solution = x.getArray()
    b.setArray(current_solution)
    b.setValue(0, 0.0)
    b.setValue(N_POINTS - 1, 0.0)
```


Implicit Backward Time Centered Space
Initial state
Numerical solution
1.0
0.8
0.6
0.4
0.2
0.0
$u\,(x, t = 0.000)$
0.0
0.2
0.4
0.6
0.8
1.0
x

A python hydrodynamics code for teaching and prototyping (pyro2)

- It is designed to provide a tutorial for students in computational astrophysics (and hydrodynamics in general) and for easily prototyping new methods.
- Builds off of a finite-volume framework for solving PDEs.

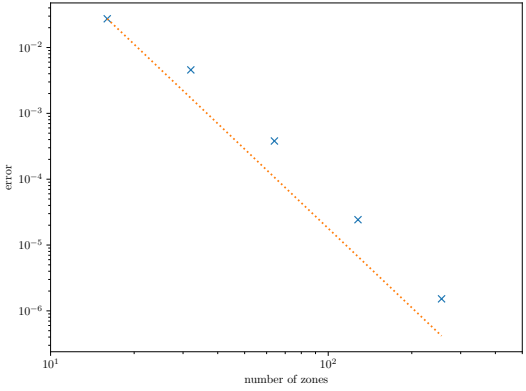
  Recovered from `https://python-hydro.github.io/pyro2`.
- Latest version `4.4.0` dated September 21, 2024.
- Michael Zingale wrote an open text Introduction to Computational Astrophysical Hydrodynamics that introduces the core finite-volume methods used in astrophysics simulation codes.



A python hydrodynamics code for teaching and prototyping (pyro2)

```python
from pyro import Pyro

nzones = [16, 32, 64, 128, 256]
err = []
params_all = {"driver.cfl": 0.5, "driver.max_steps": 5000}

for N in nzones:
    params = {"mesh.nx": N, "mesh.ny": N}
    p = Pyro("advection_fv4")
    p.initialize_problem(problem_name="smooth", inputs_dict=params | params_all)
    a_init = p.get_var("density").copy()
    p.run_sim()
    print(f"N = {N}, number of steps = {p.sim.n}")
    a = p.get_var("density")
    err.append((a - a_init).norm())
```

```
N = 16, number of steps = 64
N = 32, number of steps = 128
N = 64, number of steps = 256
N = 128, number of steps = 512
N = 256, number of steps = 1024
```



Finite differences as approximations of partial derivatives

Definition (Partial derivatives)

Let $u\colon \mathbb{R}^d \to \mathbb{R}$ be a sufficiently smooth function and $x \in \mathbb{R}^d$.

$$\forall i = 1, \ldots, d : \frac{\partial u\,(x)}{\partial x_i} \coloneqq \lim_{\Delta x \to 0} \frac{u\,(x + \Delta x e_i) - u\,(x)}{\Delta x}$$

where e_i is the i-th vector of the canonical basis of $\mathbb{R}^d$.

Remark

The finite difference method (FDM) arises by approximating the derivative of a function u by an expression of differences in its values at certain nearby discrete points, and thus, we convert a differential equation into a finite system of algebraic equations that can be solved on the computer.

The choice of this "finite difference" should be

Consistent The approximation should be as accurate as possible and find a finite difference approximation of the derivatives which is consistent with the highest possible order.

Stable Not only with respect to data perturbations, but in discrete versions of the same rules where the solution to the continuous problem has its own stability properties.

Second-order finite difference approximation for $\partial_x^2 u$

To derive a finite difference approximation of a higher-order derivative, we need to truncate the Taylor series to a higher order than that of the derivative. For example, to derive a finite difference approximation of $\partial_x^2 u$, we require the third-order forward and backward Taylor series:

$$u(x+\Delta x) = u(x) + \Delta x \partial_x u(x) + \frac{(\Delta x)^2}{2}\partial_x^2 u(x) + \frac{(\Delta x)^3}{6}\partial_x^3 u(x) + O\left((\Delta x)^4\right).$$
$$u(x-\Delta x) = u(x) - \Delta x \partial_x u(x) + \frac{(\Delta x)^2}{2}\partial_x^2 u(x) - \frac{(\Delta x)^3}{6}\partial_x^3 u(x) + O\left((\Delta x)^4\right).$$

We want to approximate $\partial_x^2 u(x)$, so we need to eliminate the terms $\partial_x u(x)$ and $\partial_x^3 u(x)$. Since these terms have opposite signs in the forward and backward Taylor expansions, we can do this by adding the two, i.e.,

$$u(x-\Delta x) + u(x+\Delta x) = 2u(x) + (\Delta x)^2 \partial_x^2 u(x) + O\left((\Delta x)^4\right).$$

Rearranging to clear $\partial_x^2 u(x)$, we get:

$$\partial_x^2 u(x) = \frac{u(x-\Delta x) - 2u(x) + u(x+\Delta x)}{(\Delta x)^2} + O\left((\Delta x)^2\right).$$

This is the second order symmetric difference approximation of $\partial_x^2 u(x)$.

Remark

When adding the forward and backward Taylor expansions, the odd-order terms cancel out. This means that using an odd-order Taylor expansion will result in a first order approximation of accuracy.

Finite differences as approximations of partial derivatives

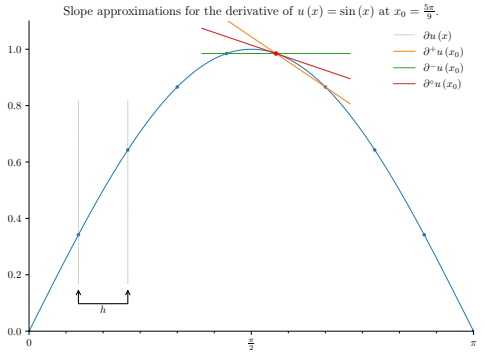


Figure: Forward difference $\partial^+ u(x)$, backward difference $\partial^- u(x)$, centered difference $\partial^\circ u(x)$.

Finite differences are analogous to partial derivatives in several variables.

$$\partial_x u(x) \approx \begin{cases} \partial^+ u(x) \coloneqq \dfrac{u_{i+1} - u_i}{\Delta x}, & i = 0, \dots, n-1. \\ \partial^- u(x) \coloneqq \dfrac{u_i - u_{i-1}}{\Delta x}, & i = 1, \dots, n. \\ \partial^\circ u(x) \coloneqq \dfrac{u_{i+1} - u_{i-1}}{2\Delta x}, & i = 1, \dots, n-1. \end{cases}$$

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

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

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

Mixed derivatives

Suppose that

$$\partial_x \partial_y u\,(x,y) = \partial_y \partial_x u\,(x,y)\,.$$

Let Δx and Δy be the step sizes for variables x and y, respectively, and using a forward approximation ∂^+ for $\partial_x u\,(x,y)$ and $\partial_y u\,(x,y)$:

$$\partial_x u\,(x,y) \approx \partial_x^+ u\,(x,y) = \frac{u\,(x+\Delta x,y) - u\,(x,y)}{\Delta x}.$$
$$\partial_y u\,(x,y) \approx \partial_y^+ u\,(x,y) = \frac{u\,(x,y+\Delta y) - u\,(x,y)}{\Delta y}.$$

Then,

$$\begin{aligned}
\partial_x \partial_y u\,(x,y) \approx \partial_x^+ \left[\partial_y^+ u\,(x,y)\right] &= \frac{\partial_y^+ u\,(x+\Delta x,y) - \partial_y^+ u\,(x,y)}{\Delta x}. \\
&= \frac{\frac{u(x+\Delta x,y+\Delta y)-u(x+\Delta x,y)}{\Delta y} - \frac{u(x,y+\Delta y)-u(x,y)}{\Delta y}}{\Delta x}. \\
&= \frac{u\,(x+\Delta x,y+\Delta y) - u\,(x+\Delta x,y) - u\,(x,y+\Delta y) + u\,(x,y)}{\Delta x \Delta y}.
\end{aligned}$$

Theorem (Properties of difference operators)

Let be $u_1, u_2 : \mathbb{R} \to \mathbb{R}$ two functions.

$$\partial^+ (u_1 u_2)(x) = \partial^+ u_1(x)\, u_2(x) + u_1(x+h)\, \partial^- u_2(x+h).$$
$$\partial^- (u_1 u_2)(x) = \partial^- u_1(x)\, u_2(x) + u_1(x-h)\, \partial^+ u_2(x-h).$$
$$h \sum_{k=0}^{N-1} \partial^+ u_1(kh)\, u_2(kh) = u_1(Nh)\, u_2(Nh) - u_1(0)\, u_2(0) - h \sum_{k=1}^{N} u_1(kh)\, \partial^- u_2(kh).$$
$$\partial^- \partial^+ u(x) = \partial^+ \partial^- u(x).$$

Gradient and Hessian

Let be $u : \mathbb{R}^d \to \mathbb{R}$ a function.

$$\forall j = 1, \ldots, d : [\nabla u(x)]_j \approx \partial_j^+ u(x).$$

$$\forall j, k = 1, \ldots, d : \left[\nabla^2 u(x)\right]_{jk} = \partial_{x_j} \partial_{x_k} u(x) \approx \partial_k^+ [\nabla u(x)]_j.$$

$$\left[\nabla^2 u(x)\right]_{ij} \approx \frac{u(x + \Delta x \hat{e}_i + \Delta x \hat{e}_j) - u(x - \Delta x \hat{e}_i + \Delta x \hat{e}_j) - u(x + \Delta x \hat{e}_i - \Delta x \hat{e}_j) + u(x - \Delta x \hat{e}_i - \Delta x \hat{e}_j)}{4(\Delta x)^2}.$$

Finite differences as approximations of partial derivatives

Remark

Use `np.roll` in FDM, performs periodic shift of an array.

```python
import numpy as np

u = np.linspace(start=0, stop=20, num=6)
print(f"u_{{i+1}} = {np.roll(a=u, shift=-1)}")
print(f"u_{{i-1}} = {np.roll(a=u, shift=1)}")
```

index	u[:]	u[1:]	u[1:-1]	u[:-1]
0	0	4	nan	nan
1	4	8	4	0
2	8	12	8	4
3	12	16	12	8
4	16	20	nan	12
5	20	nan	nan	16

```
u_{i+1} = [ 4.  8. 12. 16. 20.  0.]

u_{i-1} = [20.  0.  4.  8. 12. 16.]
```

Program : `subarrays.py`.

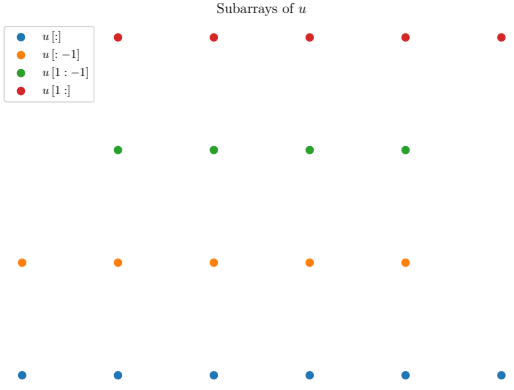


Figure: Point diagram of subarrays of u.

Finite differences as approximations of partial derivatives

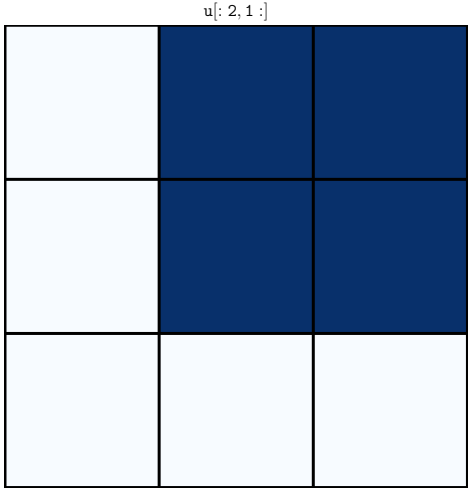


Figure: Select from the first row to the second and from the second column to the end.

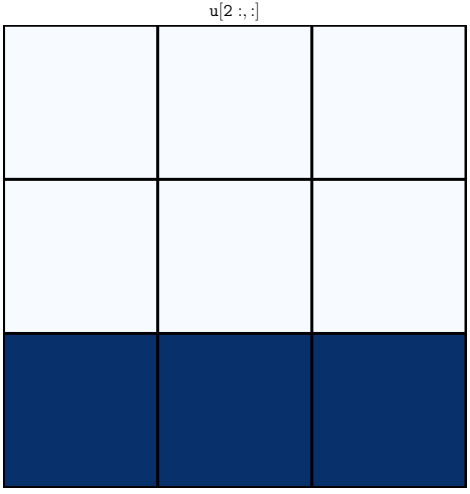


Figure: Select from the third row to the end and every all columns of u.

Finite differences as approximations of partial derivatives

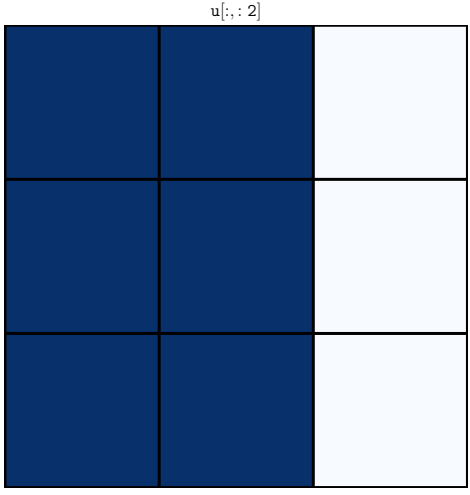


Figure: Select every all rows of u and from the first column to the second.

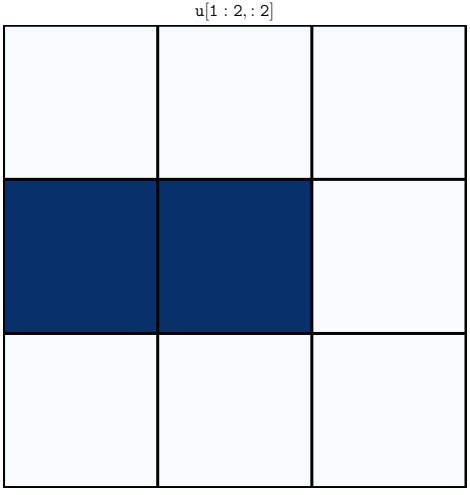


Figure: Select from the second row to the second and from the first column to the second.

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

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

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

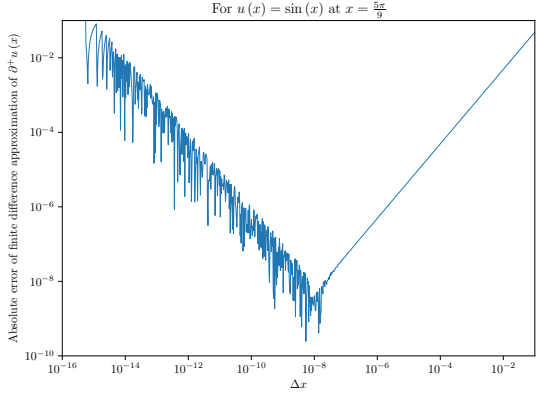


Figure: Error of $\partial^+ u(x)$ for several values of Δx .

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

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

Truncation error

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

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

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

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

```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_atributed_to_other_ommited_terms = np.abs(
    truncation_error - absolute_lower_order_ommited_term
)
```

```
First order approximation: 1.0
Exact: 0.9950041652780258
Truncation error: 0.004995834721974794
Absolute lower order ommited term: 0.005000000000000001
Error atributed to other terms: 4.165278025821534e-06
```

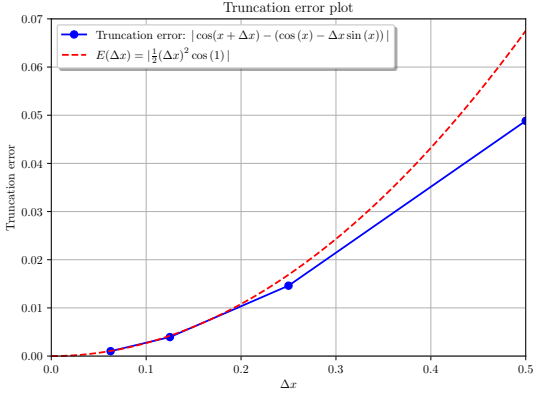


Figure: Truncation error for several values of Δx at $x = 1$.

Finite differences as approximations of partial derivatives

Given a known truncation error $E(\Delta x) = O\left((\Delta x)^n\right)$ 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
```

```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 .

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)
```

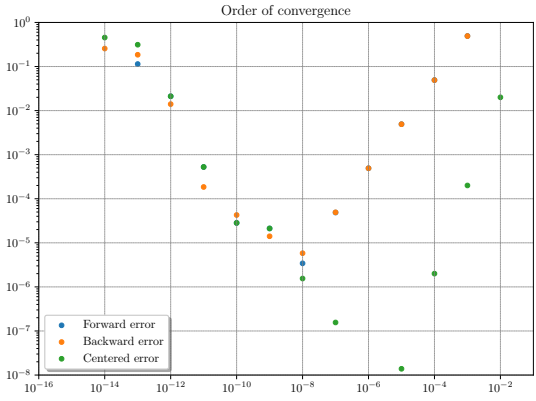


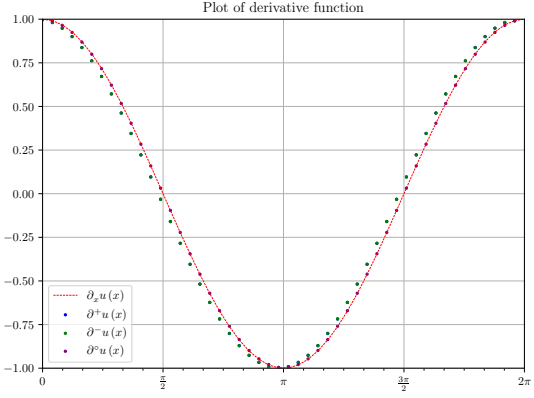
Figure: $\partial^{+}u\left(x\right)$, $\partial^{-}u\left(x\right)$, $\partial^{\circ}u\left(x\right)$ of $u\left(x\right)=\exp\left(x^{2}\right)$ for several values of Δx at $x=2$.

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 $u_{i+1} - ui.$

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

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




```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)
```

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]

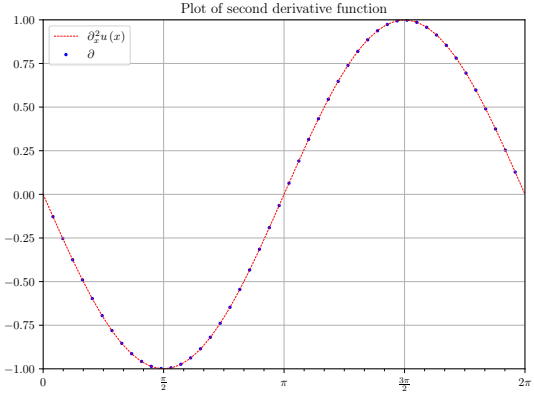
np.roll(a = u, shift = 1), u, u − np.roll(a = u, shift = 1)

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

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

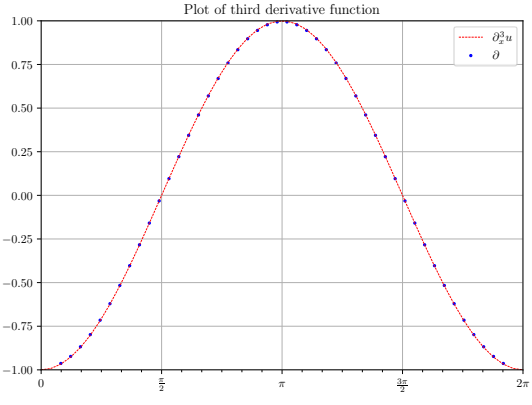
```
import numpy as np

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

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

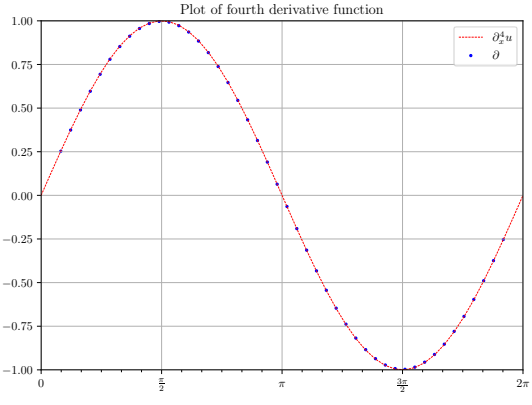
```
import numpy as np

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[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,

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

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

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

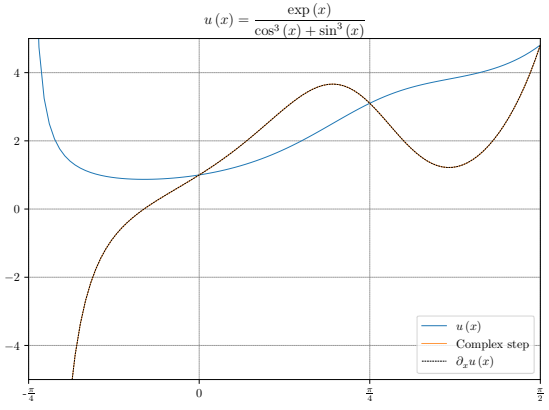
$$u\left(x+i\Delta x\right)=u\left(x\right)+i\Delta x\partial_{x}u\left(x\right)+\frac{\left(i\Delta x\right)^{2}}{2!}\partial_{x}^{2}u\left(x\right)+\cdots.$$

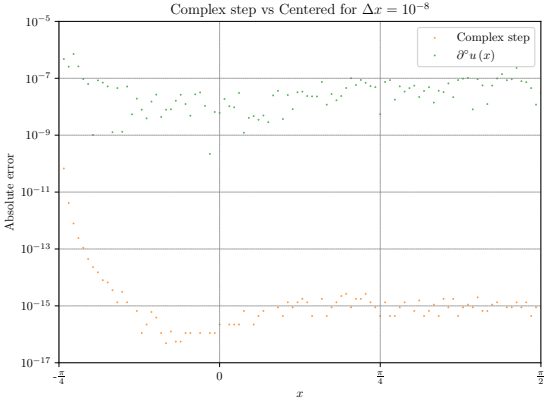
$$\operatorname{Re}\left[u\left(x+i\Delta x\right)\right]+i\operatorname{Im}\left[u\left(x+i\Delta x\right)\right]\approx u\left(x\right)+i\Delta x\partial_{x}u\left(x\right).$$

Comparing real parts of the two sides gives

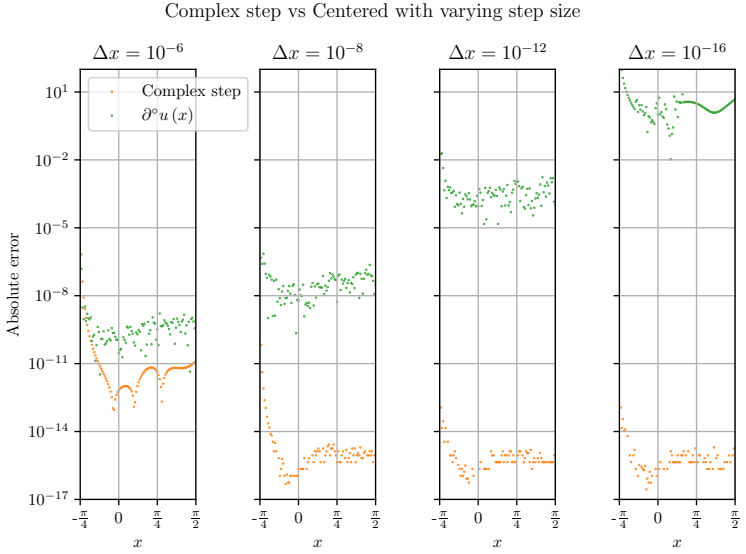
$$\partial_{x}^{2}u\left(x\right)\approx\frac{2}{\Delta x^{2}}\left(u\left(x\right)-\operatorname{Re}\left[u\left(x+i\Delta x\right)\right]\right).$$

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:

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

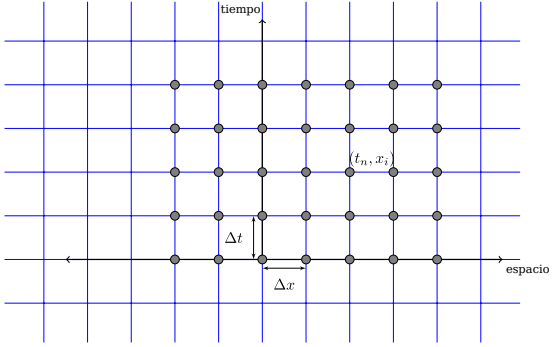
or

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

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

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

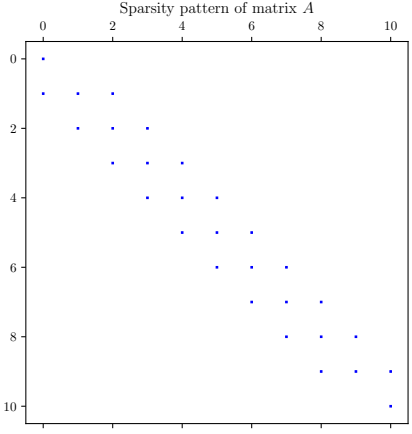
$$\begin{aligned}\frac{\partial a}{\partial x}\bigg|_i &= \frac{a_i - a_{i-1}}{\Delta x} - \frac{1}{2}\Delta x \frac{\partial^2 a}{\partial x^2}\bigg|_i + \ldots \\ &= \frac{a_i - a_{i-1}}{\Delta x} + \mathcal{O}(\Delta x)\end{aligned}$$



```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 -9.9999999999999999E1
2 2 1.9999999999999997E2
2 3 -9.9999999999999999E1
3 2 -9.9999999999999999E1
3 3 1.9999999999999997E2
3 4 -9.9999999999999999E1
4 3 -9.9999999999999999E1
4 4 1.9999999999999997E2
4 5 -9.9999999999999999E1
5 4 -9.9999999999999999E1
5 5 1.9999999999999997E2
5 6 -9.9999999999999999E1
6 5 -9.9999999999999999E1
6 6 1.9999999999999997E2
6 7 -9.9999999999999999E1
7 6 -9.9999999999999999E1
7 7 1.9999999999999997E2
7 8 -9.9999999999999999E1
8 7 -9.9999999999999999E1
8 8 1.9999999999999997E2
8 9 -9.9999999999999999E1
9 8 -9.9999999999999999E1
9 9 1.9999999999999997E2
9 10 -9.9999999999999999E1
10 9 -9.9999999999999999E1
10 10 1.9999999999999997E2
10 11 -9.9999999999999999E1
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 2.3204831651684845E1
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 -5.153246793460846E-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.

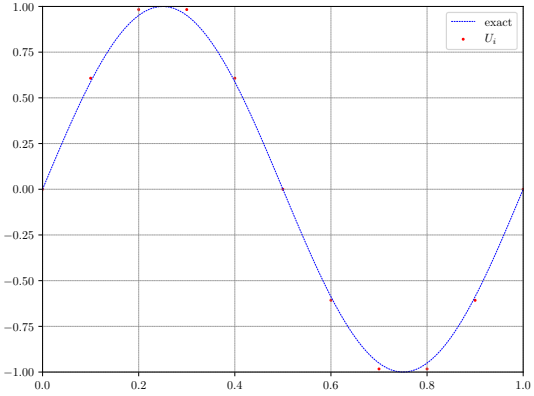


Figure: Solution.

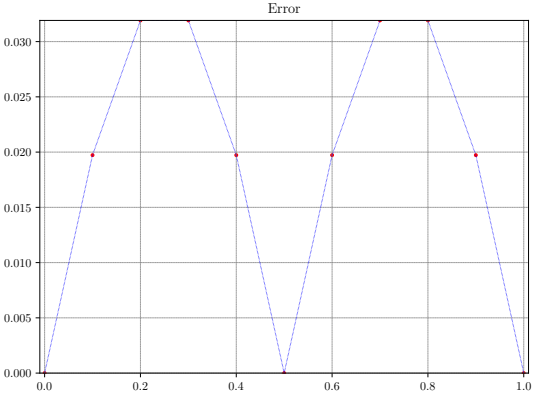
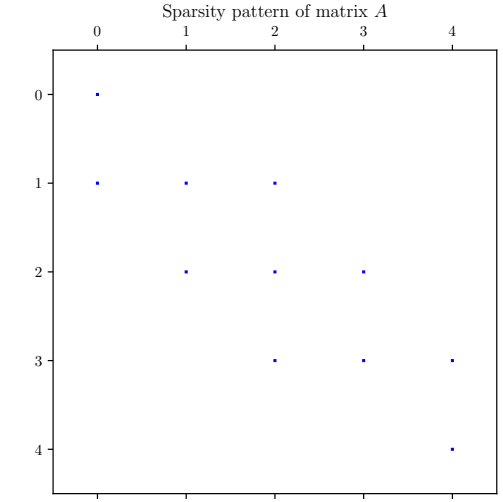


Figure: Error.

```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

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
```

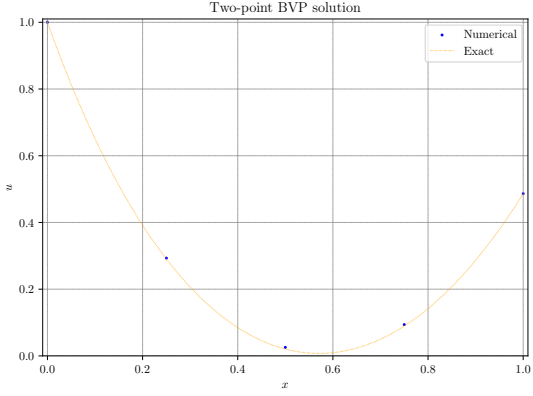
Program : twopointboundaryA.mm.

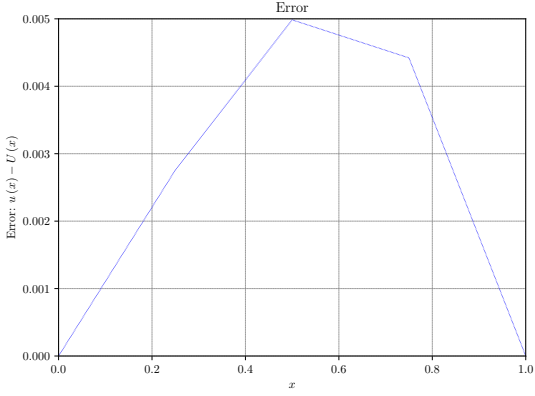
```
%%MatrixMarket matrix coordinate real general
%
1 5 5
1 1 1
1 2 1.40625E-1
1 3 2.8125E-1
1 4 4.21875E-1
1 5 4.863507252859538E-1
```

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
```

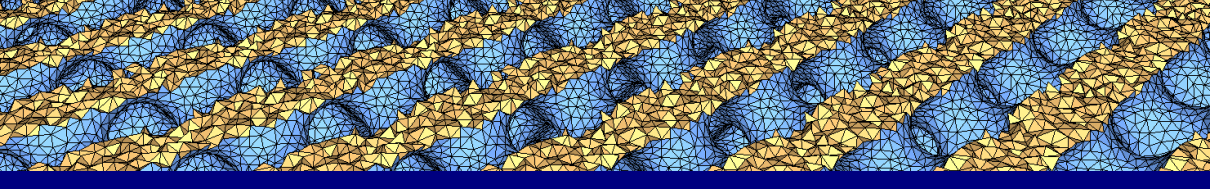
Program : twopointboundaryU.mm.


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

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

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



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 $\Omega_h = \{x_1, \ldots, x_m\}$ with grid points $x_j = jh$ and mesh width $h = \frac{1}{m}$.
Discrete Fourier decomposition is an important tool to analize 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

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

The Courant-Friedrichs-Lewy (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 appropiate 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

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

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

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

Remark

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

Example (Transport equation on a periodic domain)

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

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

We consider the implicit FDM

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

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

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

Solution

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

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

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

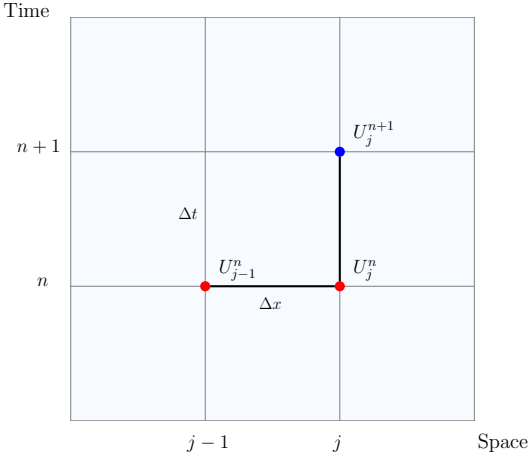
$$v_j^m \leq \max_{j=0,\ldots,n+1} v_j^0 \leq \sup_{x\in[0,1]} g(x).$$

Example (First-Order Upwind (FOU))

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

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

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



Theorem (Stability analysis for FOU scheme)

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

Proof.

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

□

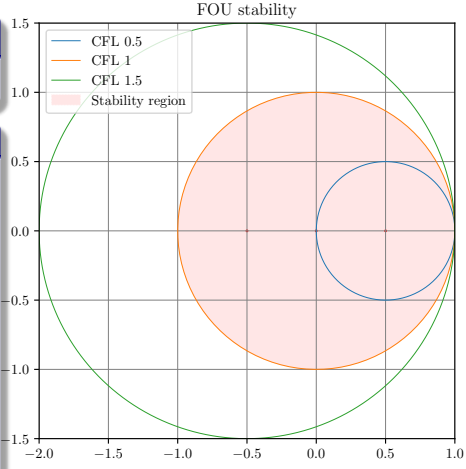


Figure: The FOU scheme is dissipative.

```toml
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.

```python
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.

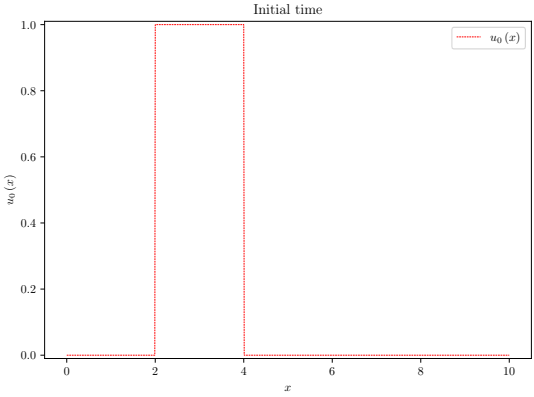
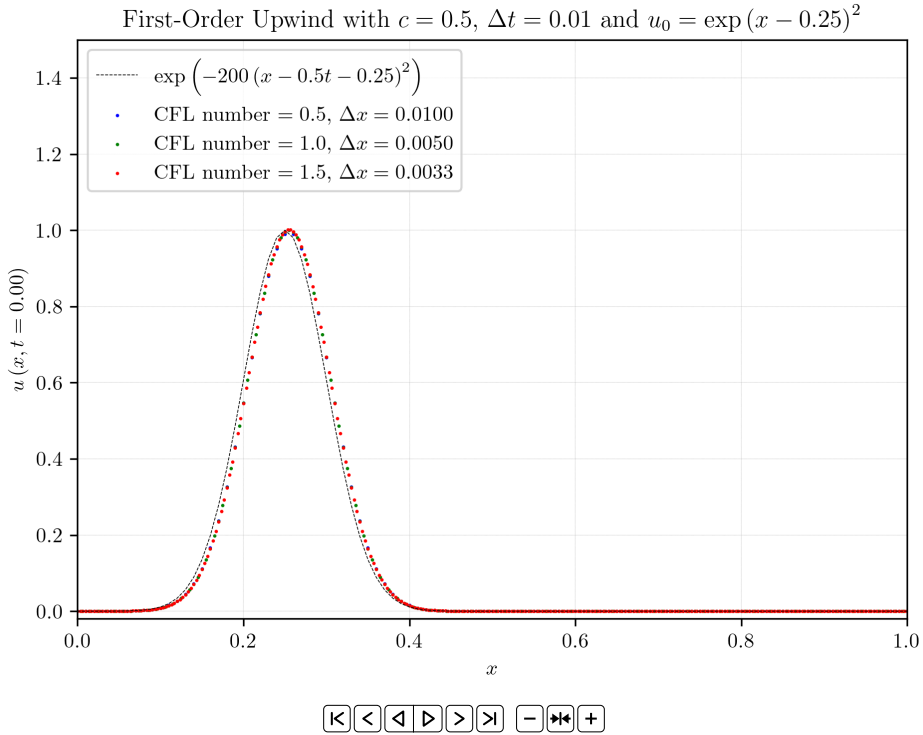
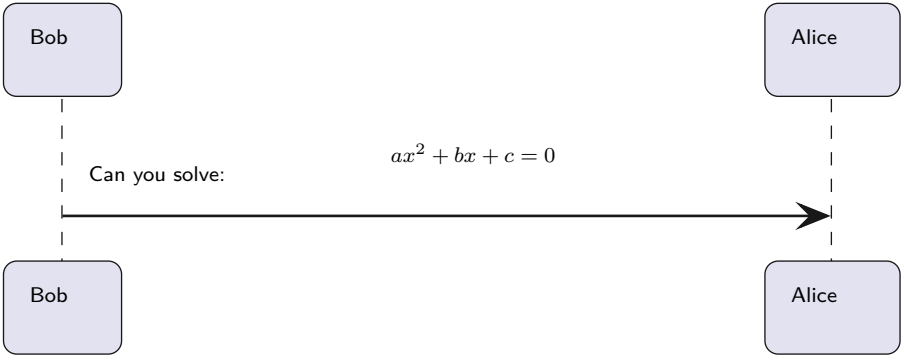


Figure: Initial profile.

```
CFL number: 0.625

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


First-Order Upwind with $c = 0.5$, $\Delta t = 0.01$ and $u_0 = \exp(x - 0.25)^2$
$\exp\left(-200\,(x - 0.5t - 0.25)^2\right)$
CFL number = 0.5, $\Delta x = 0.0100$
CFL number = 1.0, $\Delta x = 0.0050$
CFL number = 1.5, $\Delta x = 0.0033$
$u\,(x, t = 0.00)$
x
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

Flow chart

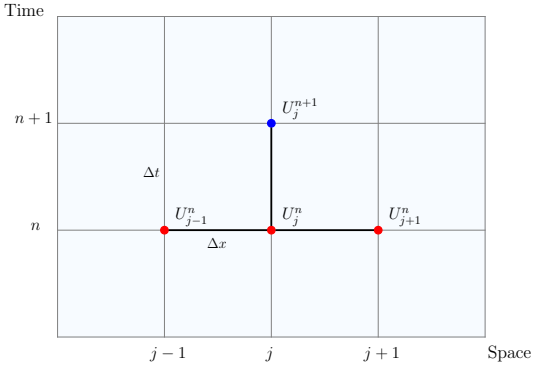


Fourier stability analysis

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

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

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



Theorem (Stability analysis for FTCS scheme)

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

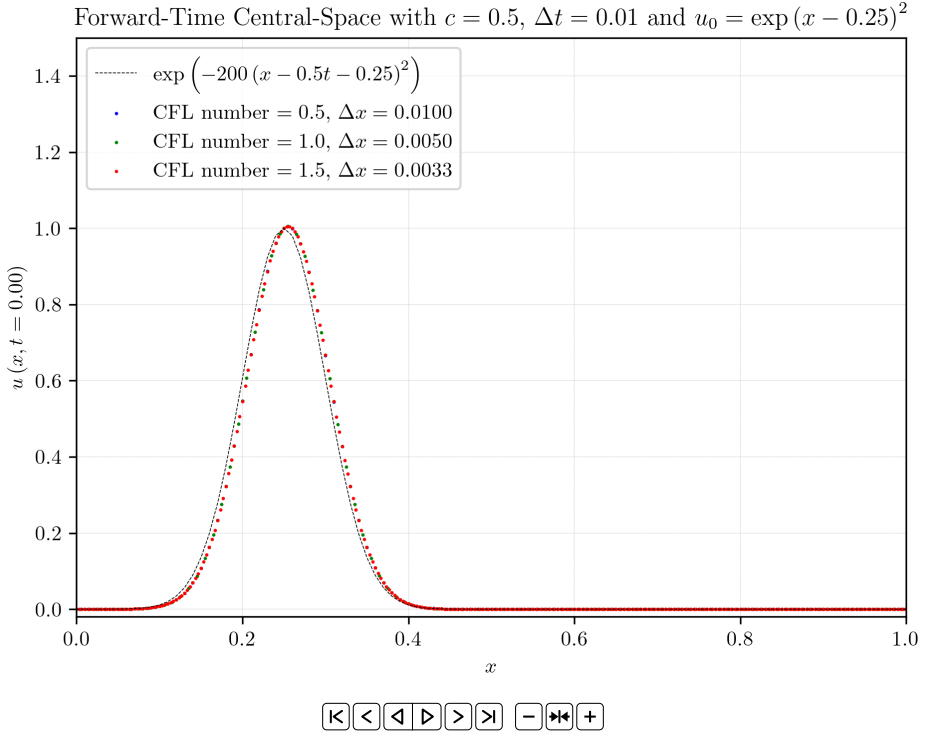
Proof.

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

□

Remark

We say that the FTCS scheme is unconditionally unstable.

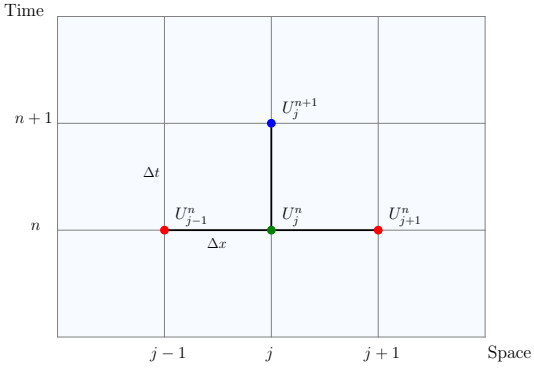

Forward-Time Central-Space with $c = 0.5$, $\Delta t = 0.01$ and $u_0 = \exp(x - 0.25)^2$
$\exp\left(-200\,(x - 0.5t - 0.25)^2\right)$
CFL number $= 0.5$, $\Delta x = 0.0100$
CFL number $= 1.0$, $\Delta x = 0.0050$
CFL number $= 1.5$, $\Delta x = 0.0033$
$u\,(x, t = 0.00)$
x
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

Fourier stability analysis

Example (Lax-Friedrichs scheme)

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

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



Theorem (Stability analysis for Lax-Friedrichs)

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

Proof.

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

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

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

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

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

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

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

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

□

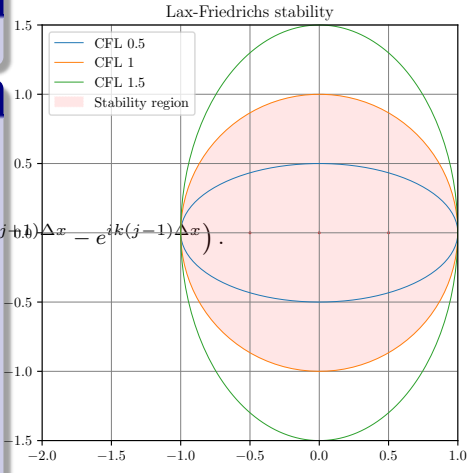


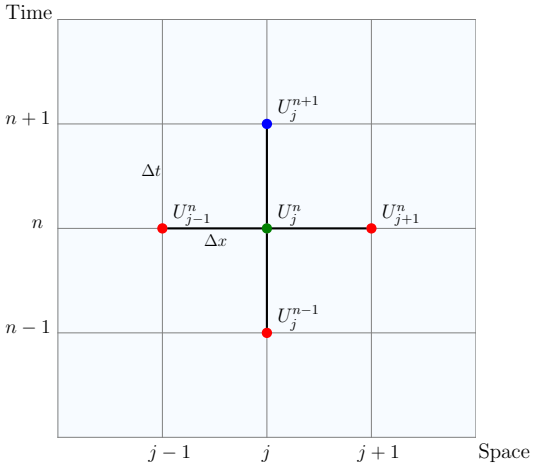
Figure: The Lax-Friedrichs scheme is oscillates.

Fourier stability analysis

Example (Leapfrog scheme)

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

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

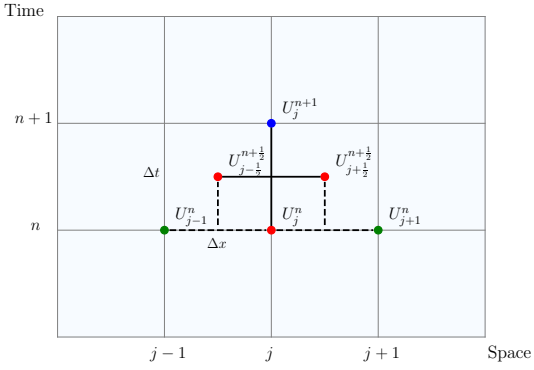


$$\left|U_j^{n+1}\right| = \left|U_j^{n-1} - r\frac{U_{j+1}^n - U_{j-1}^n}{2}\right| = \left|0 - r\frac{e^{ikj\Delta x}e^{ik\Delta x} - e^{ikj\Delta x}e^{-ik\Delta x}}{2}\right|.$$

Example (Lax-Wendroff scheme)

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

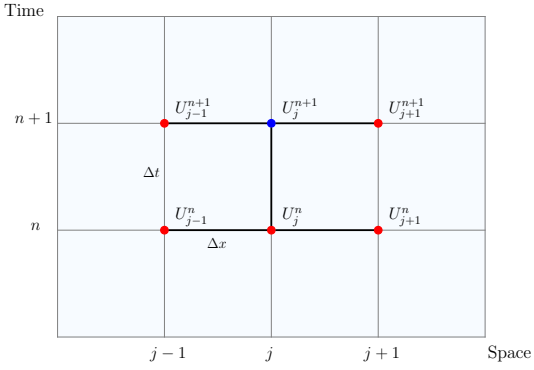
$$U_j^{n+1} = U_j^n - \frac{c\Delta t}{\Delta x}\left(U_{j+\frac{1}{2}}^{n+\frac{1}{2}} - U_{j-\frac{1}{2}}^{n+\frac{1}{2}}\right).$$



Example (Crank-Nicolson scheme)

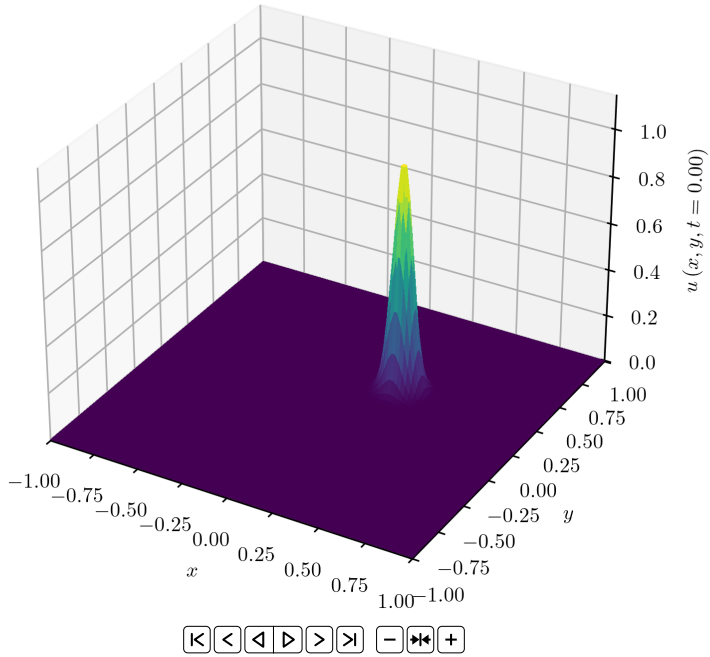
$$0 = \frac{U_i^{n+1} - U_i^{n-1}}{2\Delta t} + c\frac{U_{i+1}^n - U_{i-1}^n}{2\Delta x}.$$

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



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.


1.0
0.8
0.6
0.4
0.2
0.0
$u\,(x, y, t = 0.00)$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

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

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

then it follows that

$$\frac{\mathrm{d}u(\xi(t), t)}{\mathrm{d}t} = 0,$$

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