

# 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

```
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

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

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

Taylor expansion:

$$u(x + \Delta x) = u(x) + \Delta x \cdot \frac{d}{dx}(u(x)) + \frac{\Delta x^2}{2} \cdot \frac{d^2}{dx^2}(u(x)) + O(\Delta x^3)$$

Simplify:

$$\begin{aligned} &\cos(\theta) \\ &\sin(\theta) \\ &i \cdot \sin(\theta) \end{aligned}$$

Solve equation:

$$U^{n+1}_j = -U^n_j \cdot r + U^n_j + U^{n_{j-1}} \cdot r$$

Laplace transform:

$$\frac{2}{-\pi \cdot k}$$

$$\sqrt{\pi} \cdot e$$

Inverse Laplace transform:

$$\frac{2}{-x} \cdot e$$

# 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/2xnK2ubFAt0?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

```
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)"}],
)
```

$\Delta u = 0$  with boundary conditions:  $\sin(y), \sin(x)$

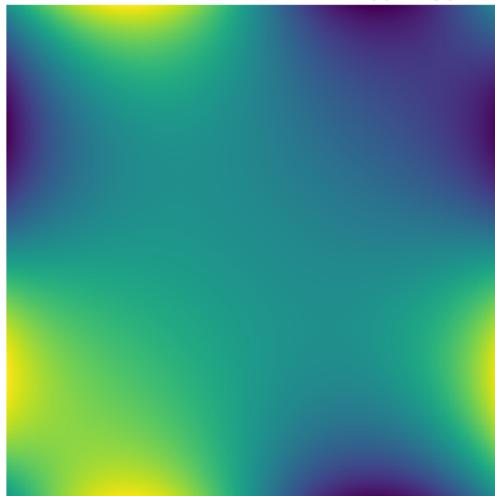


Figure: Solution of Laplace equation using the methods of lines in a rectangular grid.

# 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

```
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,
)
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,
)
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()
```

# DOLFINx: Next generation FEniCS problem solving environment

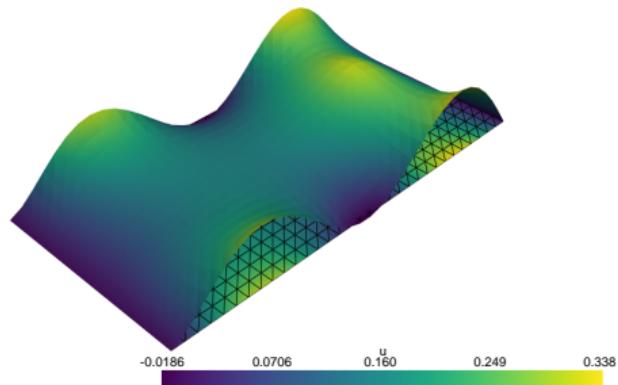


Figure: Warp By Scalar filter over solution.

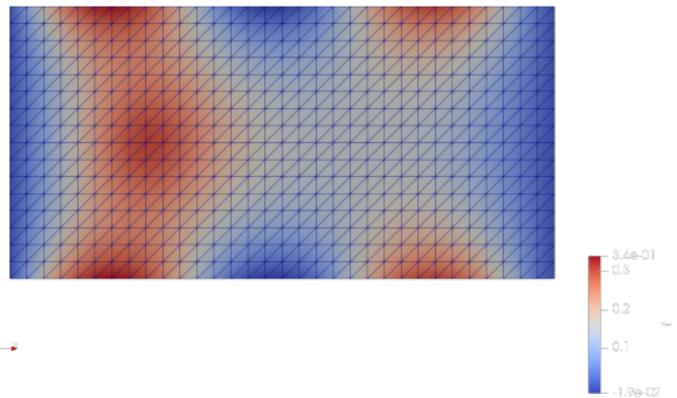


Figure: Solution.

# 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/doctoрат/tutorial\\_clawpack.pdf](https://en.ancey.ch/cours/doctoрат/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)

```
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

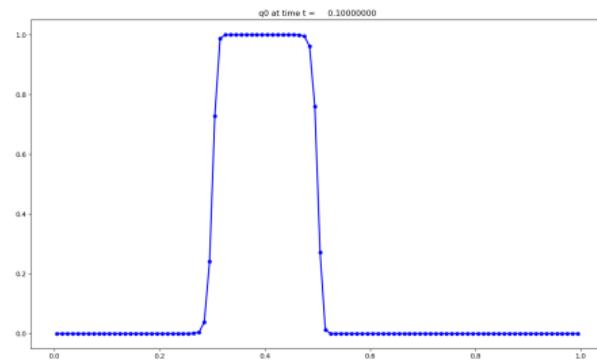
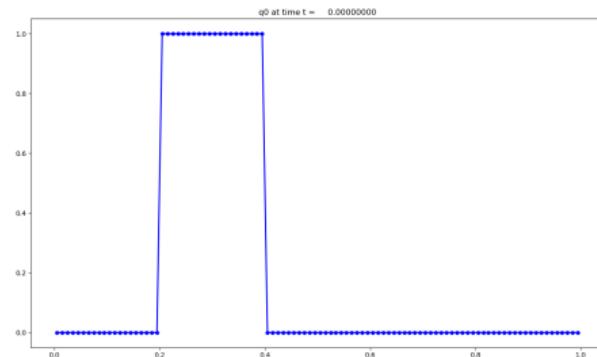
# the library
solver = ClawSolver1D(riemann_solver=advection_1D)
solver.bc_lower[0] = BC.periodic
solver.bc_upper[0] = BC.periodic

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

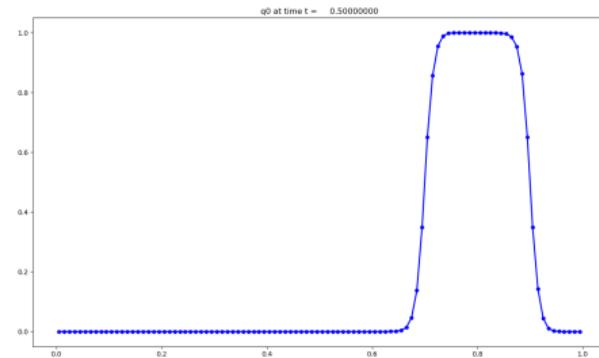
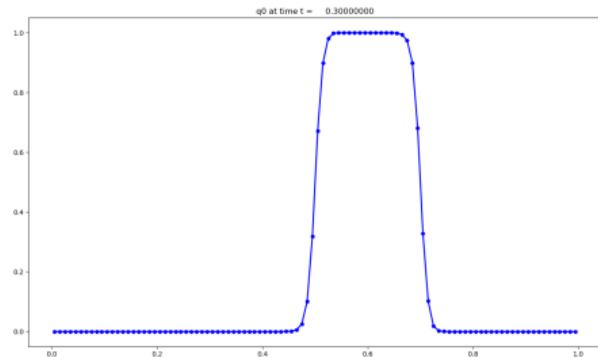
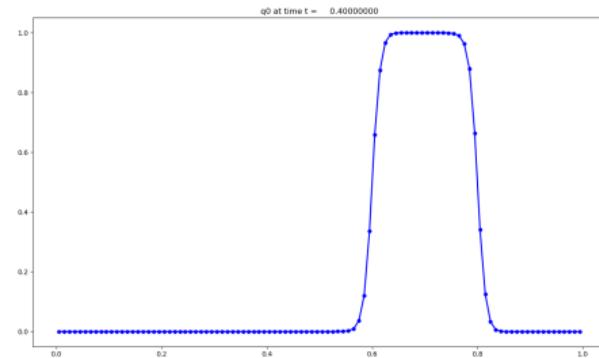
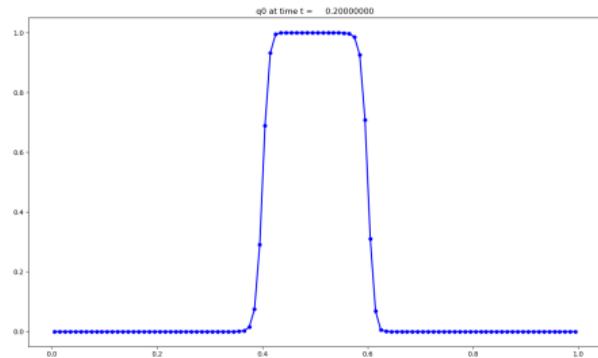
solution = Solution(solver.num_eqn, domain)

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

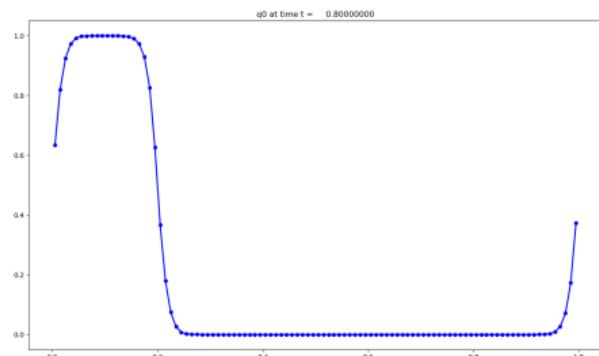
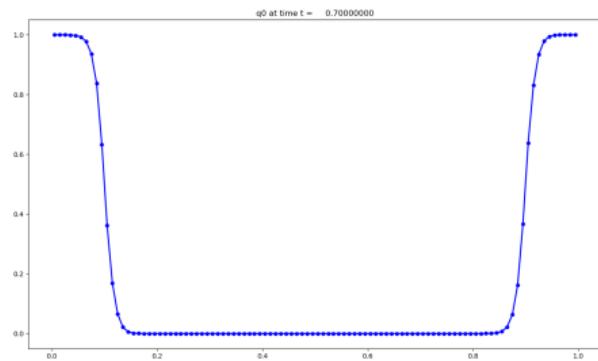
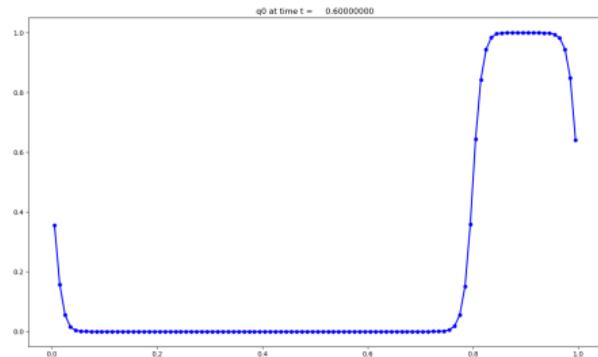
controller = Controller()
controller.solution = solution
controller.solver = solver
controller.tfinal = 1.0
```



# 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)

```
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, Ax = 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 / Ax**2
off_diagonal_entry = -1.0 * TIME_STEP_LENGTH / Ax**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)
```



# A python hydrodynamics code for teaching and prototyping (pyro2)

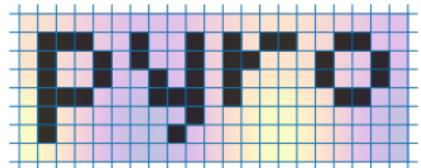
- It is designed to provide a tutorial for students in computational astrophysics (and hydrodynamics in general) and for easily prototyping new methods.

- 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.



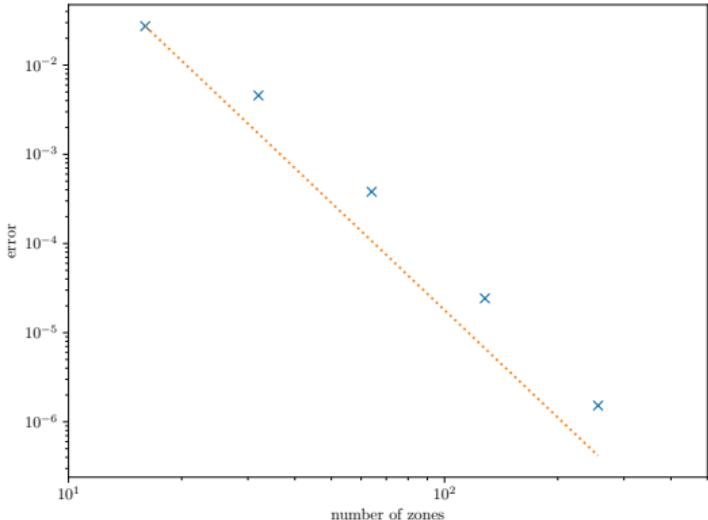
# A python hydrodynamics code for teaching and prototyping (pyro2)

```
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  $u: \mathbb{R}^d \rightarrow \mathbb{R}$  be a sufficiently smooth function and  $x \in \mathbb{R}^d$ .

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

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:

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

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.,

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

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

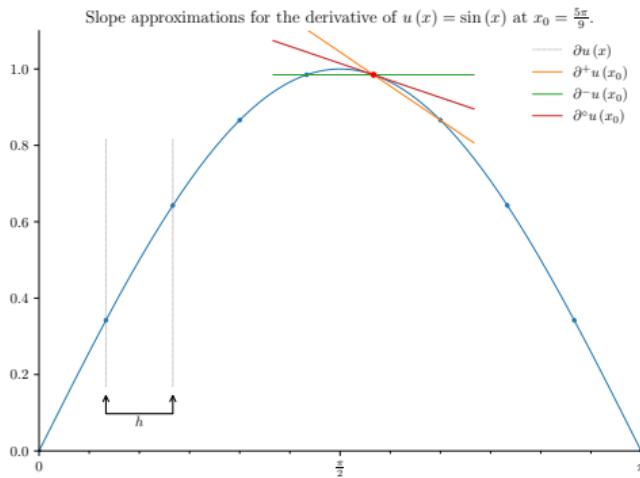
$$\partial_x^2 u(x) = \frac{u(x - \Delta x) - 2u(x) + u(x + \Delta x)}{(\Delta x)^2} + O((\Delta x)^2).$$

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.

$$\begin{aligned}\partial_x u(x) &\approx \begin{cases} \partial^+ u(x) := \frac{u_{i+1} - u_i}{\Delta x}, & i = 0, \dots, n-1. \\ \partial^- u(x) := \frac{u_i - u_{i-1}}{\Delta x}, & i = 1, \dots, n. \\ \partial^\circ u(x) := \frac{u_{i+1} - u_{i-1}}{2\Delta x}, & i = 1, \dots, n-1. \end{cases} \\ \partial_x^2 u(x) &\approx \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}, \quad i = 1, \dots, n-1. \\ \partial_x^3 u(x) &\approx \frac{u_{i+2} - 2u_{i+1} + 2u_{i-1} - u_{i-2}}{2(\Delta x)^3}, \quad i = 2, \dots, n-2. \\ \partial_x^4 u(x) &\approx \frac{u_{i+2} - 4u_{i+1} + 6u_i - 4u_{i-1} + u_{i-2}}{(\Delta x)^4}, \quad i = 2, \dots, n-2.\end{aligned}$$

## Mixed derivatives

Suppose that

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

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

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

Then,

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

## Theorem (Properties of difference operators)

Let be  $u_1, u_2: \mathbb{R} \rightarrow \mathbb{R}$  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  $u: \mathbb{R}^d \rightarrow \mathbb{R}$  a function.

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

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

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

# Finite differences as approximations of partial derivatives

## Remark

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

```
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	16	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`.

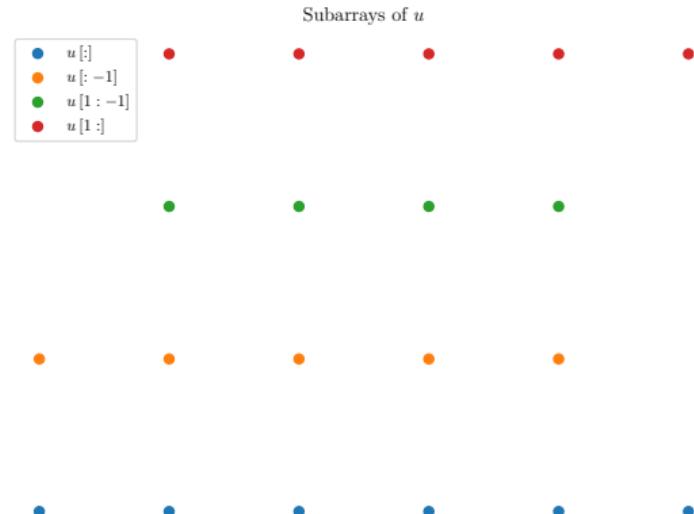


Figure: Point diagram of subarrays of  $u$ .

## Finite differences as approximations of partial derivatives

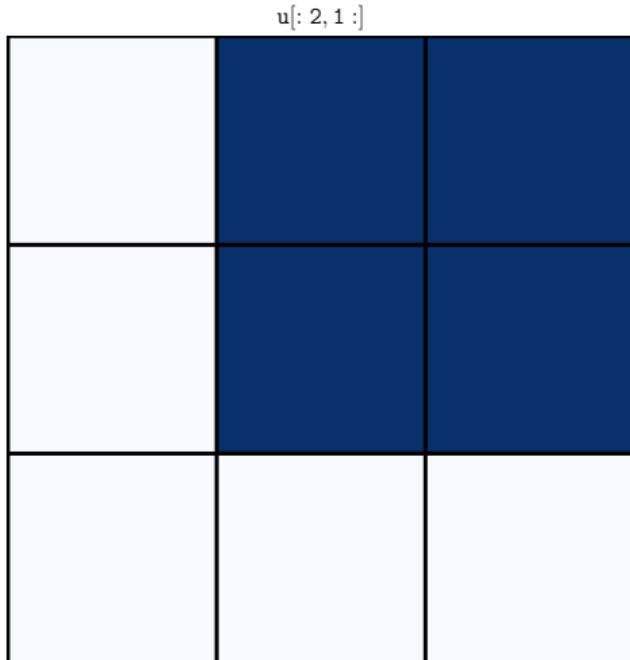


Figure: Select from the **first row to the second** and from the **second column to the end**.

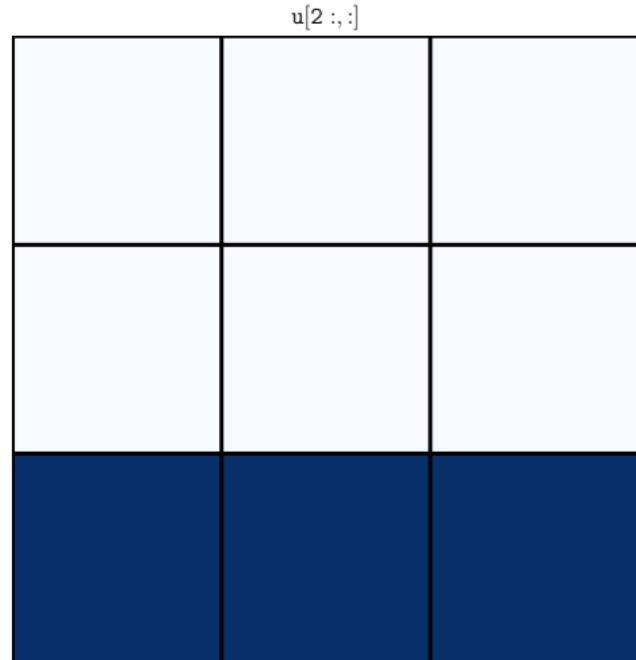


Figure: Select from the **third row to the end** and **every all columns of u**.