- Biblioteca de matemáticas simbólicas.

- Su objetivo es convertirse en un sistema de álgebra computacional con todas las funciones , manteniendo al mismo tiempo la código lo más simple posible para que sea comprensible y fácilmente extensible.

Recuperado de https://www.sympy.org.

- Tutorial de 2 horas a cargo de Aaron Meurer: https://youtu.be/FZWevQ6Xz6U?t=3059.

- Módulos destacados: sympy.calculus, integrales simpáticas, serie sympy.fourier, sympy.solvers.oda, y sympy.solvers.pde.

- Última versión 1.13.3 del 18 de septiembre de 2024.

Computación simbólica en Python

```
de sympy importa Derivada como D de
sympy.abc importa U, c, k, r, theta, x, y de sympy.core
importa Eq, Función, I, Símbolo, pi, var de sympy.functions importa
exp de sympy.integrals importa
transformada_de_fourier, transformada_de_fourier_inversa

u = Función("u") laplace

= Eq(lhs=D(U, x, 2) + D(U, y, 2), rhs=0) Δt = Símbolo("Δt") Δx =

Símbolo("Δx") u =
Función("u") taylor =
Eq(u(x + Δx), u(x +
Δx).series(x=Δx, x0=0, n=3).simplify()) expresión1 = (exp(I * theta) + exp(-I * theta)) / 2 expresión2 =

(exp(I * theta) - exp(-I * theta)) / (2 * I) expresión3 = (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, resolver(((Unp1j - Unj) / Δt + c * (Unj - Unjm1) / Δx).subs({Δt: r * Δx / c}), Unp1j)[
        0
    ],
)

a = transformada de Fourier(f=exp(-(x***2)), x=x, k=k)

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

s = series_de_fourier(f=x***2, límites=(x, -pi, pi)) iterador =
s.truncate(n=Ninguno) para n en rango(10):
término = siguiente(iterador)
    print(f"a_{n}:")
    pprint(término.subs(x,
    0))
```

Ecuación de Laplace:

$$\frac{d^2}{Dx^2}(U) + \frac{d^2}{dy^2}(U) = 0$$

Expansión de Taylor:

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

Simplifica:

$$\cos(\theta)$$
$$\sin(\theta)$$
$$i \cdot \sin(\theta)$$

Resolver ecuación:

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

Transformada de Laplace: 2^2

$$\sqrt{\pi} \cdot e^{-\pi \cdot k}$$

Transformada de Laplace inversa:

$$\frac{2}{mi}$$

py-pde: un paquete de Python para resolver ecuaciones diferenciales parciales

- Contiene clases para cuadrículas en las que se pueden definir campos escalares y tensoriales.
  Los operadores diferenciales asociados se calculan utilizando una implementación de diferencias finitas.

  Recuperado de https://www.zwickergroup.org/software.

- Tutorial de 60 minutos por mí: https://youtu.be/2xnK2ubFAto?t=273.

- Destaca tres módulos útiles: pde.pdes.laplace, pde.pdes.difusión y onda pde.pdes.

- Última versión 0.41.0 del 5 de agosto de 2024.

```
Desde matemáticas, importe pi

desde pde importar CartesianGrid, resolver_ecuación_de_laplace

res = resolver_ecuación_de_laplace(
    cuadrícula=CartesianGrid(límites=[[0, 2 * pi]] * 2, forma=2***8),
    bc={"valor": "sin(y)"}, {"valor": "sin(x)"},
)
```

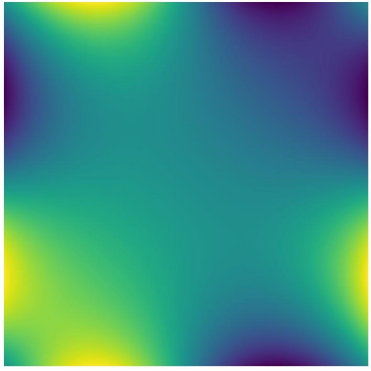


Figura: Solución de la ecuación de Laplace utilizando los métodos de líneas en una cuadrícula rectangular.

DOLFINx: entorno de resolución de problemas FEniCS de próxima generación

- El proyecto FEniCS es un proyecto de investigación y software cuyo objetivo es crear
  Métodos matemáticos y software para resolver ecuaciones en derivadas parciales.

- La última versión del proyecto FEniCS, FEniCSx, consta de varios bloques de construcción
  bloques, a saber, DOLFINx, UFL, FFCx y Basix.

- DOLFINx, un solucionador de elementos finitos de última generación .

Recuperado de https://jsdokken.com/dolfinx-tutorial.

- Tutorial de 45 minutos a cargo de Antonio Baiano Svizzero:
  https://youtu.be/uQW2wSDtW5k.

- Destaca cinco módulos útiles: dolfinx.mesh.create_rectangle,
  dolfinx.fem.espaciofuncional, dolfinx.mesh.locate_entities_boundary,
  ufl.Función de prueba y ufl.TrialFunction.

- Última versión 0.9.0.post0 del 9 de octubre de 2024.

DOLFINx: entorno de resolución de problemas FEniCS de próxima generación

```
importar numpy como
np desde dolfinx.fem importar dirichletbc, functionspace, locate_dofs_topological desde dolfinx.fem.petsc
importar LinearProblem desde dolfinx.io importar VTKFile
desde dolfinx.mesh importar CellType,
create_rectangle, locate_entities_boundary desde mpi4py importar MPI desde petsc4py.PETSc importar
ScalarType desde ufl importar
(SpatialCoordinate, TestFunction, TrialFunction,
ds, dx,




    exp,
    grad,
    interior,
    pecado,
)

msh = crear_rectángulo(
    comunicación=MPI.COMM_WORLD,
    puntos=((0.0, 0.0), (2.0, 1.0)), n=(32, 16),

    tipo_celda=CellType.triangle,
)
```

```
V = espacio de funciones (malla = msh, elemento = ("Lagrange", 1))
facetas = localizar_límite_entidades( msh =
        msh, dim
        = 1,
        marcador = lambda x: np.logical_or (np.isclose (x[0], 0.0), np.isclose (x[0], 2.0)),
)

dofs = locate_dofs_topological(V=V, entidad_dim=1, entidades=facetas) bc =
dirichletbc(valor=ScalarType(0), dofs=dofs, V=V)

u = FunciónDePrueba(espacio_función=V) v =
FunciónDePrueba(espacio_función=V) x =
CoordenadaEspacial(dominio=msh) f = 10 *
exp(-((x[0] - 0,5) *** 2 + (x[1] - 0,5) *** 2) / 0,02) g = sin(f=5 * x[0]) a =
interior(a=grad(u),
b=grad(v)) * dx L = interior(a=f, b=v) * dx +
interior(a=g, b=v) * ds

problema = ProblemaLineal(
        a=a, L=L, bcs=[bc], opciones_petsc={"ksp_type": "preonly", "pc_type": "lu"}

) uh = problema.resolver()
```

DOLFINx: entorno de resolución de problemas FEniCS de próxima generación

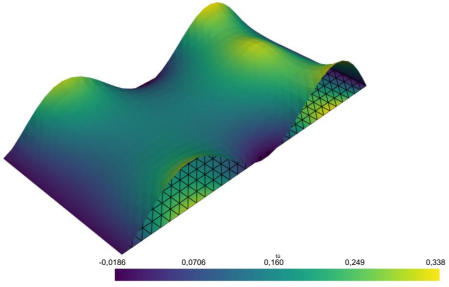


Figura: Deformación por filtro escalar sobre solución.

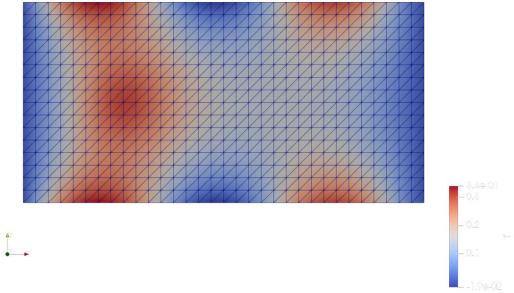


Figura: Solución.

Paquete de leyes de conservación (Clawpack)

- Es una colección de métodos de volumen finito para ecuaciones hiperbólicas lineales y no lineales. sistemas de leyes de conservación.

- Emplea métodos de tipo Godunov de alta resolución con limitadores en un sentido general. Marco aplicable a muchos tipos de ondas.

Recuperado de https://www.clawpack.org.

- Tutorial de 15 minutos de Felix Köhler: https://youtu.be/tr348El2A4Q.

- Última versión 5.11.0 del 26 de agosto de 2024.

- PyClaw: versión Python de los solucionadores de EDP hiperbólicas que permite resolver la Problema en Python sin utilizar explícitamente ningún código Fortran.

- Tutorial amigable: https://en.ancey.ch/cours/doctorat/tutorial_clawpack.pdf

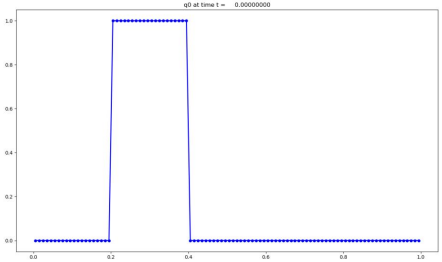
- Randall J. LeVeque y otros escribieron dos libros sobre este paquete:

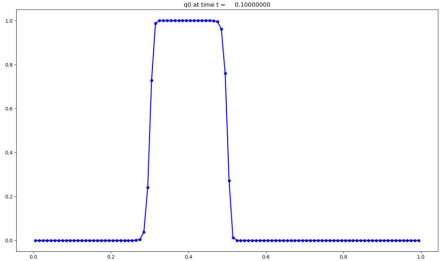
  - Métodos de volumen finito para problemas hiperbólicos (2002).

  - Problemas de Riemann y soluciones de Jupyter (2020).

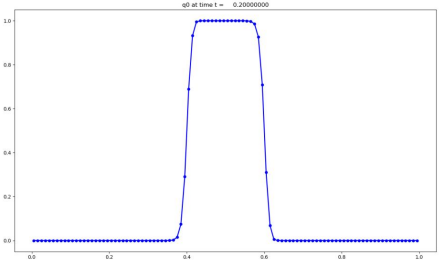
Paquete de leyes de conservación (Clawpack)

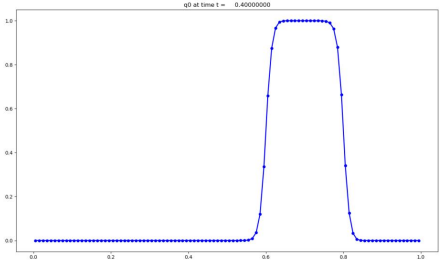
```
importar numpy como
np desde clawpack.pyclaw importar BC, ClawSolver1D, Controlador, Dimensión, Dominio, Solución
desde clawpack.pyclaw.plot importar interactive_plot desde
clawpack.riemann importar advection_1D

# el solucionador de
la biblioteca = ClawSolver1D(riemann_solver=advection_1D) solver.bc_lower[0]
= BC.periodic solver.bc_upper[0] = BC.periodic



dimensión_x = Dimensión(inferior=0.0, superior=1.0, num_celdas=100) dominio =
Dominio(dimensión_x)


solución = Solución(solver.num_eqn, dominio)


estado = solución.estado
coordenadas_del_centro_de_la_celda = estado.cuadrícula.p_centros[0]
estado.q[0, :] = np.donde(
    (coordenadas del centro de la celda > 0,2) y (coordenadas del centro de la celda < 0,4),
    1.0,
    0.0,
)


estado.problema_datos["u"] = 1.0


controlador = Controlador()
controlador.solucion = solucion
controlador.solver = solucionador
controlador.tfinal = 1.0
```

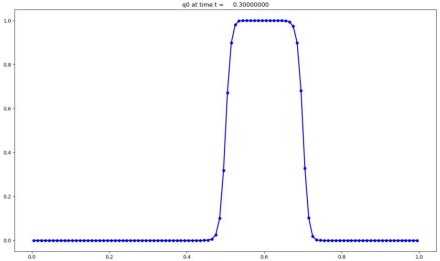


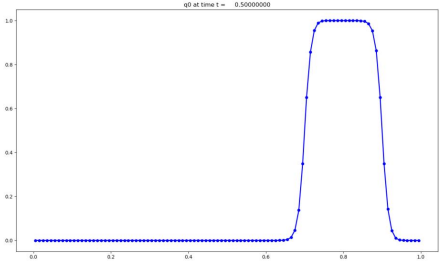


Paquete de leyes de conservación (Clawpack)

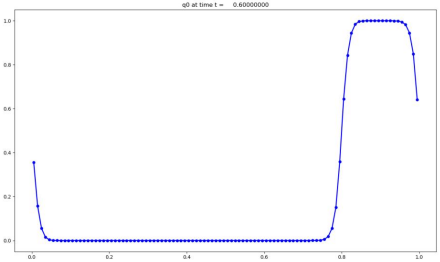


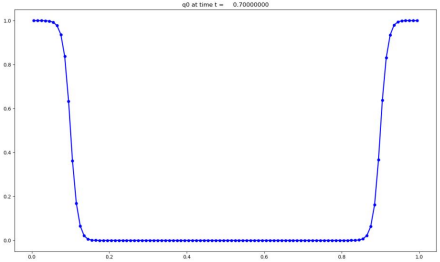


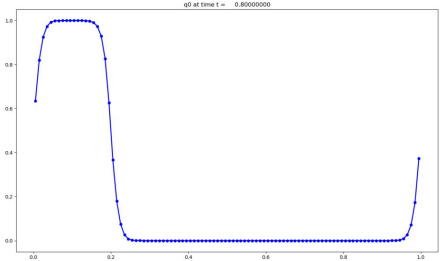




Paquete de leyes de conservación (Clawpack)







Kit de herramientas portátil y extensible para computación científica (PETSc)

- es un conjunto de estructuras de datos y rutinas para la solución escalable (paralela) de Aplicaciones científicas modeladas por ecuaciones diferenciales parciales.

- Utiliza el estándar MPI para todas las comunicaciones de paso de mensajes.

Recuperado de https://petsc.org/release/petsc4py.

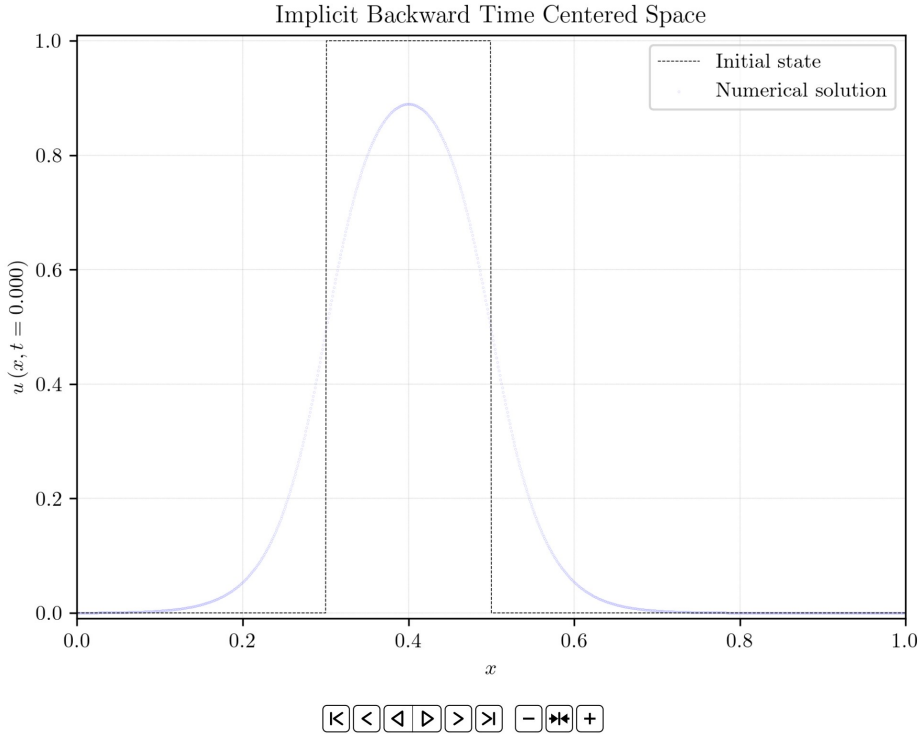
- Tutorial de 25 minutos de Felix Köhler: https://youtu.be/oqxPyRZKOu4.

- Última versión 3.22.0 del 28 de septiembre de 2024.

- Demostración: Resolver un problema de Poisson de coeficiente constante en una cuadrícula regular.
- Libro: PETSc para ecuaciones diferenciales parciales: soluciones numéricas en C y Pitón (2020).

Kit de herramientas portátil y extensible para computación científica (PETSc)

```
importar numpy como
np importar petsc4py

desde petsc4py.PETSc importa KSP, Mat, Vec

N_PUNTOS = 1001
LONGITUD_PASO_DE_TIEMPO
= 0,001
N_PASOS_DE_TIEMPO = 100 mallas, Δx = np.linspace(inicio=0,0, fin=1,0, num=N_PUNTOS, retstep=Verdadero)

# Crea una nueva matriz PETSc dispersa, rellénala y luego ensámblala A =
Mat().createAIJ([N_POINTS, N_POINTS])
A.setUp()

entrada_diagonal = 1.0 + 2.0 * LONGITUD_PASO_TIEMPO / Δx***2
entrada_diagonal_desactivada = -1.0 * LONGITUD_PASO_TIEMPO / Δx***2

A.setValue(0, 0, 1.0)
A.setValue(N_PUNTOS - 1, N_PUNTOS - 1, 1.0) para i en

rango(1, N_PUNTOS - 1):
    A.setValue(i, i, entrada_diagonal)
    A.setValue(i, i - 1, fuera de la entrada diagonal)
    A.setValue(i, i + 1, fuera de la entrada diagonal)

A.ensamblar()

# Definir la condición inicial
condición_inicial = np.donde(
    (malla > 0,3) y (malla < 0,5), 1,0, 0,0,


)

# Ensamble el rhs inicial al sistema lineal b =
Vec().createSeq(N_POINTS)
b.setArray(initial_condition) b.setValue(0,
0.0) b.setValue(N_POINTS
- 1, 0.0)

# Asigna un vector PETSc que almacena la solución al sistema lineal x =
Vec().createSeq(N_POINTS)

# Crear una instancia de un solucionador lineal: solucionador iterativo lineal del subespacio de Krylov
ksp = KSP().create()
ksp.setOperators(A)
ksp.setFromOptions()

chosen_solver = ksp.getType()
print(f"Resolver con {chosen_solver:}")

def animate(ti):
    print(f"Cuadro: {ti + 1}") plt.clf()
    plt.plot( malla,


    condición_inicial, color="negro",
        etiqueta="Estado
        inicial", ancho_de_línea=0.4,

        estilo_de_línea="discontinua",

    ) ksp.solve(b, x)

    #Vuelve a ensamblar el lado derecho para avanzar en el tiempo.
    solución_actual = x.getArray()
    b.setArray(solución_actual) b.setValue(0,
    0.0) b.setValue(N_POINTS
    - 1, 0.0)
```

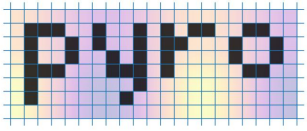


Un código de hidrodinámica en Python para la enseñanza y la creación de prototipos (pyro2)

- Está diseñado para proporcionar un tutorial para estudiantes en astrofísica computacional. (y la hidrodinámica en general) y para prototipar fácilmente nuevos métodos.

- Se basa en un marco de volumen finito para resolver ecuaciones en derivadas parciales.

Recuperado de https://python-hydro.github.io/pyro2.



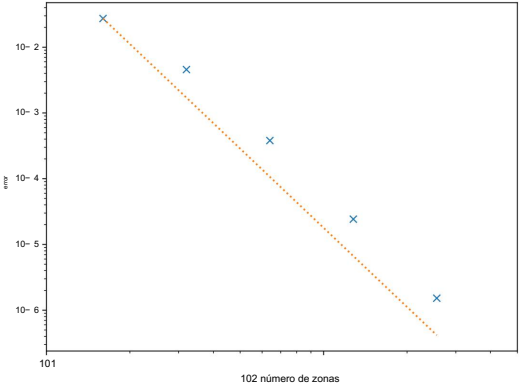
- Última versión 4.4.0 del 21 de septiembre de 2024.

- Michael Zingale escribió un texto abierto Introducción a la computación Hidrodinámica astrofísica que introduce los métodos básicos de volumen finito utilizado en códigos de simulación astrofísica.

Un código de hidrodinámica en Python para la enseñanza y la creación de prototipos (pyro2)

```
de pyro importación Pyro

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

para N en nzones:
    params = {"mesh.nx": N, "mesh.ny": N} p =
    Pyro("advección_fv4")
    p.inicializar_problema(nombre_problema="suave", entradas_dict=params | params_all) a_init =
    p.obtener_var("densidad").copiar() p.ejecutar_sim()
    imprimir(f"N =
    {N}, número de pasos = {p.sim.n}") a = p.obtener_var("densidad")
    err.agregar((a - a_init).norm())
```

```
N = 16, número de pasos = 64
N = 32, número de pasos = 128
N = 64, número de pasos = 256
N = 128, número de pasos = 512
N = 256, número de pasos = 1024
```



Diferencias finitas como aproximaciones de derivadas parciales

Definición (derivadas parciales)

Sea u: Rd →R una función suficientemente suave y x Rd.

$$i = 1, \ldots, d : \quad \frac{\partial u\,(x)}{\partial xi} := \text{límite}_{\Delta x \to 0} \quad \frac{u\,(x + \Delta x ei) - u\,(x)}{\Delta x}$$

donde ei es el i-ésimo vector de la base canónica de Rd.

Observación

El método de diferencias finitas (MDF) surge al aproximar la derivada de una función u mediante una expresión de diferencias
en sus valores en ciertos puntos discretos cercanos, y por lo tanto, convertimos una ecuación diferencial en un sistema finito de ecuaciones algebraicas.
ecuaciones que se pueden resolver en la computadora.

La elección de esta “diferencia finita” debería ser

Consistente La aproximación debe ser lo más precisa posible y encontrar una aproximación de diferencia finita de la
derivadas que sean consistentes con el orden más alto posible.

Estable No sólo con respecto a las perturbaciones de datos, sino en versiones discretas de las mismas reglas donde la solución
El problema continuo tiene sus propias propiedades de estabilidad.

Aproximación de diferencias finitas de segundo orden para $\partial^2 xu$

Para obtener una aproximación de diferencia finita de una derivada de orden superior, necesitamos truncar la serie de Taylor a un orden superior al de la derivada. Por ejemplo, para obtener una aproximación de diferencia finita de $\partial^2 xu$, necesitamos la serie de Taylor de tercer orden hacia adelante y hacia atrás:

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

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

Queremos aproximar $\partial^2 xu(x)$. Dado que estos términos tienen signos opuestos en las expansiones de Taylor hacia adelante y hacia atrás, podemos hacer esto sumando los dos, es decir,

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

Reordenando para despejar $\partial^2 xu(x)$, obtenemos:

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

Esta es la aproximación diferencial simétrica de segundo orden de $\partial^2 xu(x)$.

Observación

Al sumar las expansiones de Taylor hacia adelante y hacia atrás, los términos de orden impar se cancelan. Esto significa que el uso de una expansión de Taylor de orden impar dará como resultado una aproximación de precisión de primer orden.

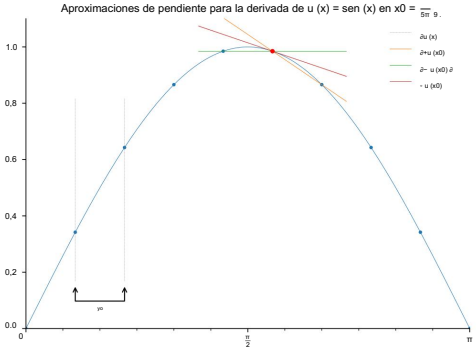


Figura: Diferencia hacia adelante ∂ +u (x), diferencia hacia atrás ∂ − u (x), diferencia centrada ∂ ◦ u (x) Python .

Las diferencias finitas son análogas a las derivadas parciales en varias variables.

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

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

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

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

$$3\,\partial xu\,(x) \approx 2(\Delta x)\,ui+2$$

$$\frac{- 4ui+1 + 6ui - 4ui - 1 + ui - 2\,4\,\partial xu\,(x) \approx (\Delta x)}{3}, \quad i = 2, \ldots, n - 2.$$

$$\frac{}{4}, \quad i = 2, \ldots, n - 2.$$

Derivados mixtos

Supongamos que

$$\partial x \partial y u\,(x, y) = \partial y\, \partial x u\,(x, y)\,.$$

Sean Δx y Δy los tamaños de paso para las variables x e y, respectivamente, y utilizando una aproximación hacia adelante $\partial +$ para $\partial x u\,(x, y)$ y $\partial y u\,(x, y)$:

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

$$\approx \partial + u\,(x, y) = \Delta y\,.$$

Entonces,

$$\partial x \partial y u\,(x, y) \approx \partial + \; u\,(x, y) = \frac{\partial + u\,(x + \Delta x, y) - \partial + u\,(x, y)\,\partial +}{\Delta x}\,.$$

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

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

Teorema (Propiedades de los operadores diferenciales)

Sean u1, u2 : R →R dos funciones.

$$\partial - (u1u2)(x) = \partial - u1(x)\,u2(x) + u1(x - h)\,\partial + u2(x - h).$$

$$\partial + u1(kh)\,u2(kh) = u1(Nh)\,u2(Nh) - u1(0)\,u2(0) - h \sum_{k=0}^{N-1} u1(kh)\,\partial - u2(kh).$$

$$\partial - \partial + u(x) = \partial + \partial - u(x).$$

Gradiente y arpillera

Sea u: Rd →R una función.

$$j = 1, \ldots, d : [u(x)]j \approx \partial + u(x).$$

$$j, k = 1, \ldots, d : \quad 2u(x) = \partial xj\,\partial xk\,u(x) \approx \partial + [u(x)]j.$$

$$2u(x) \approx \frac{u(x + \Delta xe\hat{}i + \Delta xe\hat{}j) - u(x - \Delta xe\hat{}i + \Delta xe\hat{}j) - u(x + \Delta xe\hat{}i - \Delta xe\hat{}j) + u(x - \Delta xe\hat{}i - \Delta xe\hat{}j)}{4(\Delta x)^2}.$$

Diferencias finitas como aproximaciones de derivadas parciales

Observación

Utilice np.roll En FDM, realiza un desplazamiento periódico de una matriz.

```
importar numpy como np

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

índice	tú[:]	tu[1:]	tu[1:-1]	tu[:-1]
0	0	4	yaya	yaya
1	4	8	4	0
2	8	12	8	4
3	12	16	12	8
4	16	20	16	12
5	20	yaya	yaya	16

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

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

Programa Python : subarrays.py.

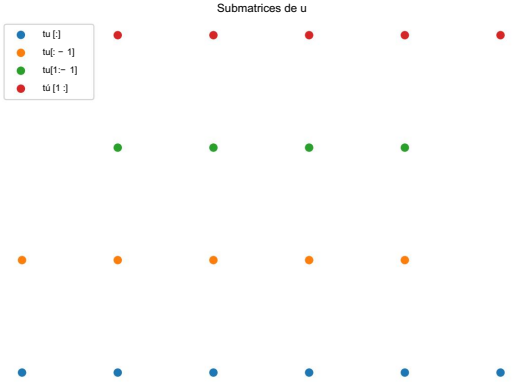


Figura: Diagrama de puntos de subconjuntos de u.

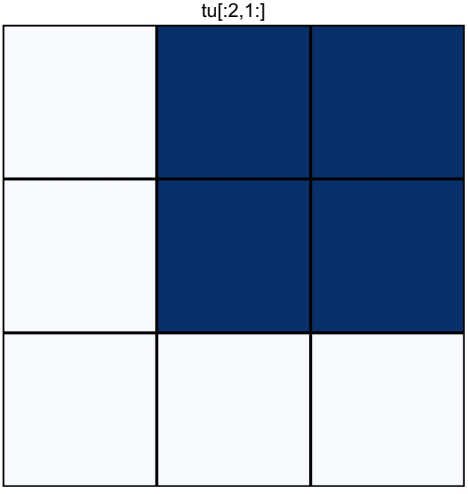


Figura: Seleccione de la primera fila a la segunda y de la Segunda columna hasta el final.

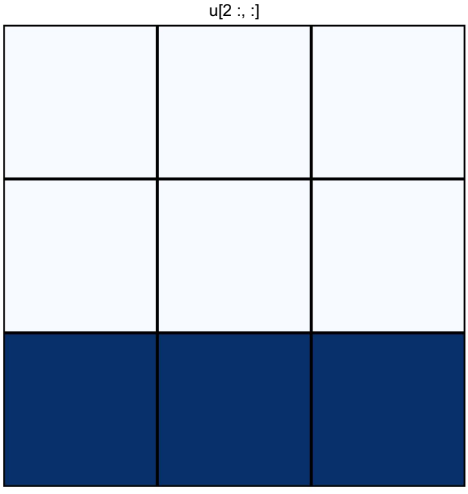


Figura: Seleccione desde la tercera fila hasta el final y todas las columnas de ti.