Apuntes de Metodos Numericos en Python

Para el examen del 18 de diciembre de 2025

Marco

Diciembre 2024

Índice

1. Derivadas Numericas

1.1. Conceptos Basicos

La derivada numerica aproxima la derivada de una funcion usando diferencias finitas.
Formula basica:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

1.2. Metodos de Diferencias Finitas

1.2.1. Diferencia hacia adelante (Forward)

```python
def derivada_forward(f, x, h=1e-5):
    return (f(x + h) - f(x)) / h
```

- **Formula:** $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
- **Error:** $O(h)$
- **Uso:** Cuando solo puedes evaluar hacia adelante

1.2.2. Diferencia central (MAS PRECISA)

```python
def derivada_central(f, x, h=1e-5):
    return (f(x + h) - f(x - h)) / (2 * h)
```

- **Formula:** $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$
- **Error:** $O(h^2)$ - Mas preciso
- **Uso:** Metodo preferido cuando es posible

1.3. Usando SciPy (RECOMENDADO)

```python
from scipy.misc import derivative

# Derivada de primer orden
dy = derivative(f, x, dx=1e-6)

# Derivada de segundo orden
d2y = derivative(f, x, dx=1e-6, n=2)
```

1.4. Para Datos Discretos

```python
import numpy as np

# Para arrays de datos
x = np.linspace(0, 10, 100)
y = f(x)

# Gradiente (derivada numerica)
```

```
dy_dx = np.gradient(y, x)

# O usando diferencias
dy_dx = np.diff(y) / np.diff(x)
```

1.5. Parametro h (paso)

- **Muy pequeno** ($h < 10^{-10}$): Errores de redondeo
- **Muy grande** ($h > 10^{-3}$): Menor precision
- **Optimo:** $h \approx 10^{-5}$ a 10^{-8}

PUNTOS CLAVE:

- Diferencia central es mas precisa
- Usar SciPy para produccion
- h tipico: 10^{-5} a 10^{-8}
- Evitar h muy pequeno (errores de redondeo)

2. Integracion Numerica

2.1. Regla del Trapecio

```python
def trapecio(f, a, b, n=1000):
    x = np.linspace(a, b, n+1)
    y = f(x)
    h = (b - a) / n
    return h * (0.5*y[0] + np.sum(y[1:-1]) + 0.5*y[-1])
```

Formula:

$$\int_a^b f(x)dx \approx h\left[\frac{y_0}{2} + y_1 + y_2 + \ldots + \frac{y_n}{2}\right]$$

Error: $O(h^2)$

2.2. Regla de Simpson

```python
def simpson(f, a, b, n=1000):
    if n % 2 == 1: n += 1  # n debe ser par
    x = np.linspace(a, b, n+1)
    y = f(x)
    h = (b - a) / n
    return h/3 * (y[0] + 4*np.sum(y[1:-1:2]) +
                  2*np.sum(y[2:-1:2]) + y[-1])
```

Formula:

$$\int_a^b f(x)dx \approx \frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + \ldots + y_n]$$

Error: $O(h^4)$ - Mas preciso
Nota: n debe ser PAR

2.3. Usando SciPy (RECOMENDADO)

```python
from scipy import integrate

# Integral definida
resultado, error = integrate.quad(f, a, b)

# Con limites infinitos
resultado, error = integrate.quad(f, 0, np.inf)

# Integral doble
resultado, error = integrate.dblquad(f, x_min, x_max,
                                     y_min, y_max)
```

2.4. Para Datos Discretos

```python
# Con NumPy
integral = np.trapz(y, x)

# Con SciPy - Trapecio
integral = integrate.trapezoid(y, x)
```

```
6
7 # Con SciPy - Simpson
8 integral = integrate.simpson(y, x)
```

PUNTOS CLAVE:

- Simpson es mas preciso que Trapecio
- `quad()` para funciones, `trapz()` para datos
- Simpson requiere n par
- Usar `quad()` para limites infinitos

3. Calculo de Raices

3.1. Conceptos Basicos

Problema: Encontrar valores de x tales que $f(x) = 0$.
Raiz: Un valor x^* donde $f(x^*) = 0$

3.2. Metodo de Biseccion

El metodo mas simple y robusto. Requiere que f(a) y f(b) tengan signos opuestos.

```python
def biseccion(f, a, b, tol=1e-6, max_iter=100):
    """
    Encuentra raiz de f en el intervalo [a, b]
    Requiere: f(a) * f(b) < 0
    """
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) y f(b) deben tener signos opuestos")

    for i in range(max_iter):
        c = (a + b) / 2

        if abs(f(c)) < tol or abs(b - a) < tol:
            return c

        if f(a) * f(c) < 0:
            b = c
        else:
            a = c

    return (a + b) / 2

# Ejemplo: encontrar raiz de x^2 - 2 = 0
def f(x):
    return x**2 - 2

raiz = biseccion(f, 0, 2)
print(f"Raiz: {raiz}")  # aprox 1.414213 (raiz de 2)
```

Ventajas:

- Siempre converge si $f(a) \cdot f(b) < 0$
- Simple de implementar
- No requiere derivadas

Desventajas:

- Convergencia lenta (lineal)
- Requiere intervalo inicial con cambio de signo

3.3. Metodo de Newton-Raphson

Metodo mas rapido pero requiere la derivada.

Formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

```python
def newton_raphson(f, df, x0, tol=1e-6, max_iter=100):
    """
    f: funcion
    df: derivada de f
    x0: estimacion inicial
    """
    x = x0

    for i in range(max_iter):
        fx = f(x)
        dfx = df(x)

        if abs(dfx) < 1e-12:
            raise ValueError("Derivada muy pequena")

        x_new = x - fx / dfx

        if abs(x_new - x) < tol:
            return x_new

        x = x_new

    raise ValueError("No convergio")

# Ejemplo: x^2 - 2 = 0
def f(x):
    return x**2 - 2

def df(x):
    return 2*x

raiz = newton_raphson(f, df, x0=1.5)
```

Ventajas:

- Convergencia cuadratica (muy rapida)
- Pocas iteraciones necesarias

Desventajas:

- Requiere derivada
- Puede diverger con mala estimacion inicial
- Falla si $f'(x) = 0$

3.4. Metodo de la Secante

Similar a Newton pero aproxima la derivada numericamente.

Formula:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

```python
def secante(f, x0, x1, tol=1e-6, max_iter=100):
    """
    x0, x1: dos estimaciones iniciales
    """
    for i in range(max_iter):
        fx0 = f(x0)
        fx1 = f(x1)

        if abs(fx1 - fx0) < 1e-12:
            raise ValueError("Division por cero")

        x_new = x1 - fx1 * (x1 - x0) / (fx1 - fx0)

        if abs(x_new - x1) < tol:
            return x_new

        x0 = x1
        x1 = x_new

    raise ValueError("No convergio")
```

Ventajas:

- No requiere derivada
- Convergencia mas rapida que biseccion

Desventajas:

- Requiere dos estimaciones iniciales
- Puede diverger

3.5. Usando SciPy (RECOMENDADO)

3.5.1. Para una variable

```python
from scipy.optimize import root_scalar

# Definir funcion
def f(x):
    return x**3 - x - 2

# Metodo 1: Biseccion
sol = root_scalar(f, bracket=[1, 2], method='bisect')
print(f"Raiz: {sol.root}")

# Metodo 2: Brent (hibrido, muy bueno)
sol = root_scalar(f, bracket=[1, 2], method='brentq')
print(f"Raiz: {sol.root}")

# Metodo 3: Newton (requiere derivada)
def df(x):
    return 3*x**2 - 1

sol = root_scalar(f, x0=1.5, fprime=df, method='newton')
```

```
print(f"Raiz: {sol.root}")

# Metodo 4: Secante
sol = root_scalar(f, x0=1, x1=2, method='secant')
print(f"Raiz: {sol.root}")
```

3.5.2. Para sistemas de ecuaciones

```
from scipy.optimize import fsolve, root

# Sistema de ecuaciones:
# x^2 + y^2 = 4
# x - y = 1

def equations(vars):
    x, y = vars
    eq1 = x**2 + y**2 - 4
    eq2 = x - y - 1
    return [eq1, eq2]

# Usando fsolve
sol = fsolve(equations, [1, 1])
print(f"Solucion: x={sol[0]:.4f}, y={sol[1]:.4f}")

# Usando root (mas control)
result = root(equations, [1, 1], method='hybr')
print(f"Solucion: {result.x}")
print(f"Convergio?: {result.success}")
```

3.6. Raices de Polinomios

```
import numpy as np

# Para polinomios, numpy tiene funcion especializada
# P(x) = x^3 - 6x^2 + 11x - 6 = (x-1)(x-2)(x-3)

coef = [1, -6, 11, -6]  # Coeficientes de mayor a menor grado
raices = np.roots(coef)

print("Raices del polinomio:")
print(raices)  # [3, 2, 1]

# Verificar
p = np.poly1d(coef)
for r in raices:
    print(f"P({r}) = {p(r)}")
```

3.7. Optimizacion (minimos/maximos)

```
from scipy.optimize import minimize_scalar, minimize

# Encontrar minimo de una funcion de una variable
```

```python
def f(x):
    return (x - 2)**2 + 1

result = minimize_scalar(f, bounds=(0, 4), method='bounded')
print(f"Minimo en x = {result.x:.6f}, f(x) = {result.fun:.6f}")

# Encontrar minimo de funcion multivariable
def f_multi(x):
    return (x[0] - 1)**2 + (x[1] - 2.5)**2

result = minimize(f_multi, x0=[0, 0], method='BFGS')
print(f"Minimo en {result.x}, f = {result.fun:.6f}")

# Maximo = minimo de -f(x)
def f_max(x):
    return -(x**2 - 4*x + 3)

result = minimize_scalar(f_max)
print(f"Maximo en x = {result.x:.6f}")
```

PUNTOS CLAVE:

- Biseccion: Siempre converge si $f(a) \cdot f(b) < 0$, robusto pero lento
- Newton-Raphson: Muy rapido (convergencia cuadratica), requiere derivada
- Secante: No requiere derivada, convergencia rapida
- SciPy: Usar `root_scalar` para una variable, `fsolve`/`root` para sistemas
- Polinomios: Usar `np.roots()`
- Optimizacion: `minimize_scalar` o `minimize`

4. Ecuaciones Diferenciales

4.1. EDOs con Condiciones Iniciales (IVP)

```python
from scipy.integrate import solve_ivp

# Definir ecuacion: dy/dt = f(t, y)
def ecuacion(t, y):
    return -2 * y  # Ejemplo: dy/dt = -2y

# Condiciones iniciales
t_span = (0, 5)  # Intervalo de tiempo
y0 = [1]         # y(0) = 1

# Resolver
sol = solve_ivp(ecuacion, t_span, y0, dense_output=True)

# Evaluar solucion
t = np.linspace(0, 5, 100)
y = sol.sol(t)
```

4.2. Metodos disponibles

- **RK45:** Runge-Kutta 4-5 (default) - Problemas generales
- **RK23:** Runge-Kutta 2-3 - Problemas simples
- **DOP853:** Runge-Kutta 8 - Alta precision
- **Radau:** Para sistemas rigidos
- **BDF:** Para sistemas rigidos
- **LSODA:** Adaptativo

4.3. Sistema de EDOs

```python
# Ejemplo: d^2y/dt^2 = -y (oscilador armonico)
# Convertir: y1 = y, y2 = dy/dt
def sistema(t, Y):
    y1, y2 = Y
    dy1_dt = y2
    dy2_dt = -y1
    return [dy1_dt, dy2_dt]

# Condiciones: y(0) = 1, dy/dt(0) = 0
y0 = [1, 0]
sol = solve_ivp(sistema, (0, 10), y0, dense_output=True)
```

4.4. EDOs con Condiciones de Frontera (BVP)

```python
from scipy.integrate import solve_bvp

# Ecuacion: y'' + y = 0
# Sistema: y1 = y, y2 = y'
def ecuacion(x, y):
    return np.vstack((y[1], -y[0]))

# Condiciones: y(0) = 0, y(pi) = 1
def bc(ya, yb):
    return np.array([ya[0] - 0,     # y(0) = 0
                     yb[0] - 1])    # y(pi) = 1

x = np.linspace(0, np.pi, 10)
y = np.zeros((2, x.size))
sol = solve_bvp(ecuacion, bc, x, y)
```

4.5. EDPs - Ecuacion de Calor 1D

Ecuacion:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

```python
# Discretizacion
# u[i,n+1] = u[i,n] + r(u[i+1,n] - 2u[i,n] + u[i-1,n])
# donde r = alpha*dt/dx^2

L = 1.0; T = 0.5; alpha = 0.01
nx = 50; nt = 1000

dx = L/(nx-1); dt = T/nt
r = alpha * dt / dx**2  # IMPORTANTE: r <= 0.5 para estabilidad

x = np.linspace(0, L, nx)
u = np.sin(np.pi * x / L)  # Condicion inicial

for n in range(nt):
    u_new = u.copy()
    for i in range(1, nx-1):
        u_new[i] = u[i] + r*(u[i+1] - 2*u[i] + u[i-1])
    u = u_new
    u[0] = 0; u[-1] = 0  # Condiciones de frontera
```

PUNTOS CLAVE:

- `solve_ivp`: condiciones iniciales
- `solve_bvp`: condiciones de frontera
- RK45 normal, Radau/BDF rigidos
- Convertir EDOs orden n a sistema de n EDOs primer orden
- EDPs: Verificar estabilidad $r \leq 0{,}5$, donde $r = \alpha dt/dx^2$

5. Interpolacion y Minimos Cuadrados

5.1. Interpolacion 1D

```python
from scipy import interpolate

x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([0, 0.8, 0.9, 0.1, -0.8, -1])

# Interpolacion lineal
f_lineal = interpolate.interp1d(x, y, kind='linear')

# Interpolacion cubica (recomendado)
f_cubica = interpolate.interp1d(x, y, kind='cubic')

# Spline cubico
tck = interpolate.splrep(x, y, s=0)
y_spline = interpolate.splev(x_nuevo, tck)
```

Tipos: 'linear', 'quadratic', 'cubic' (recomendado), 'nearest'

5.2. Interpolacion 2D

```python
from scipy.interpolate import griddata

# Datos dispersos
x = np.random.rand(100) * 4
y = np.random.rand(100) * 4
z = np.sin(x) * np.cos(y)

# Grilla regular
xi = np.linspace(0, 4, 100)
yi = np.linspace(0, 4, 100)
Xi, Yi = np.meshgrid(xi, yi)

# Interpolar
zi = griddata((x, y), z, (Xi, Yi), method='cubic')
```

5.3. Ajuste Lineal Simple

```python
from scipy import stats

slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)

# Ecuacion: y = slope * x + intercept
# R^2: r_value**2
# Error estandar: std_err
```

5.4. Ajuste Polinomial

```python
# Ajustar
grado = 2
coef = np.polyfit(x, y, grado)
```

```
p = np.poly1d(coef)
y_ajuste = p(x)

# Calcular R^2
y_pred = p(x)
ss_res = np.sum((y - y_pred)**2)
ss_tot = np.sum((y - np.mean(y))**2)
r2 = 1 - (ss_res / ss_tot)
```

5.5. Ajuste No Lineal

```
from scipy.optimize import curve_fit

# Definir funcion modelo
def modelo(x, a, b, c):
    return a * np.exp(-b * x) + c

# Ajustar (IMPORTANTE: estimacion inicial p0)
p0 = [1, 1, 1]
popt, pcov = curve_fit(modelo, x, y, p0=p0)

# popt: parametros optimizados
# pcov: matriz de covarianza

# Incertidumbres
perr = np.sqrt(np.diag(pcov))
```

5.6. Incertidumbres

5.6.1. Mediciones directas

```
from scipy import stats

mediciones = np.array([10.2, 10.5, 10.3, 10.1, 10.4])

media = np.mean(mediciones)
desviacion = np.std(mediciones, ddof=1)  # ddof=1 para muestra
error_estandar = stats.sem(mediciones)

# Intervalo de confianza 95%
confianza = 0.95
grados_libertad = len(mediciones) - 1
intervalo = stats.t.interval(confianza, grados_libertad,
                             loc=media, scale=error_estandar)
```

5.6.2. Propagacion de errores

Formulas:
Suma/Resta: $z = x \pm y$

$$\delta z = \sqrt{\delta x^2 + \delta y^2}$$

Multiplicacion/Division: $z = x \times y$ o $z = x/y$

$$\frac{\delta z}{z} = \sqrt{\left(\frac{\delta x}{x}\right)^2 + \left(\frac{\delta y}{y}\right)^2}$$

Potencia: $z = x^n$

$$\delta z = |n \cdot x^{n-1}| \cdot \delta x$$

5.7. Chi-cuadrado reducido

```python
# Para evaluar calidad del ajuste
y_pred = modelo(x, *popt)
chi2 = np.sum(((y - y_pred) / sigma_y)**2)
chi2_red = chi2 / (len(x) - len(popt))

# Si chi^2/dof aprox 1: buen ajuste
# Si chi^2/dof >> 1: modelo no describe bien
```

5.8. Bootstrap

```python
# Muestreo con reemplazo para estimar incertidumbres
n_bootstrap = 1000
parametros = []

for _ in range(n_bootstrap):
    indices = np.random.randint(0, len(x), len(x))
    x_boot = x[indices]
    y_boot = y[indices]

    popt, _ = curve_fit(modelo, x_boot, y_boot)
    parametros.append(popt)

parametros = np.array(parametros)
param_media = np.mean(parametros, axis=0)
param_std = np.std(parametros, axis=0)
```

PUNTOS CLAVE:

- Cubica: suave y precisa
- `linregress()`: lineal simple
- `polyfit()`: polinomios
- `curve_fit()`: no lineal
- Bootstrap: incertidumbres robustas
- Extrapolacion puede ser peligrosa
- `curve_fit` requiere p0

6. Metodo Monte Carlo

6.1. Que es

Definicion: Tecnica que usa muestreo aleatorio repetido para resolver problemas numericos.
Idea basica:

1. Generar muchas muestras aleatorias
2. Simular el proceso para cada muestra
3. Promediar los resultados

6.2. Estimacion de pi

```
def estimar_pi(n_puntos):
    # Puntos aleatorios en [0,1] x [0,1]
    x = np.random.uniform(0, 1, n_puntos)
    y = np.random.uniform(0, 1, n_puntos)

    # Estan dentro del circulo de radio 1?
    dentro = (x**2 + y**2) <= 1

    # pi aprox 4 * (puntos dentro / total)
    return 4 * np.sum(dentro) / n_puntos
```

Por que funciona:

- Area del cuadrado = 1
- Area del cuarto de circulo = $\pi/4$
- Razon = $\pi/4$
- Por tanto: $\pi \approx 4\times$ (puntos dentro / total)

6.3. Integracion Monte Carlo

```
def integral_mc(f, a, b, n_puntos=10000):
    # Puntos aleatorios en [a, b]
    x = np.random.uniform(a, b, n_puntos)

    # Evaluar funcion
    y = f(x)

    # Integral aprox (b-a) x promedio de f(x)
    integral = (b - a) * np.mean(y)
    error = (b - a) * np.std(y) / np.sqrt(n_puntos)

    return integral, error
```

Formula:

$$\int_a^b f(x)dx \approx (b-a) \times \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

6.4. Caminata Aleatoria

```python
def caminata_1d(n_pasos):
    # Cada paso: +1 o -1
    pasos = np.random.choice([-1, 1], n_pasos)
    posicion = np.cumsum(pasos)
    return posicion
```

Propiedades:

- Posicion media: 0
- Desviacion estandar: $\sqrt{n}$
- Distribucion final: Gaussiana

6.5. Difusion en 2D

```python
def difusion_2d(n_particulas, n_pasos):
    x = np.zeros((n_particulas, n_pasos+1))
    y = np.zeros((n_particulas, n_pasos+1))

    for paso in range(n_pasos):
        dx = np.random.normal(0, 1, n_particulas)
        dy = np.random.normal(0, 1, n_particulas)

        x[:, paso+1] = x[:, paso] + dx
        y[:, paso+1] = y[:, paso] + dy

    return x, y
```

Ley de difusion:

$$\langle r^2 \rangle = 2Dt$$

donde D es el coeficiente de difusion.

6.6. Simulacion de Precios (Black-Scholes)

```python
def simular_precio(S0, mu, sigma, T, n_pasos):
    """
    S0: precio inicial
    mu: retorno esperado
    sigma: volatilidad
    T: tiempo total
    """
    dt = T / n_pasos
    S = np.zeros(n_pasos+1)
    S[0] = S0

    for t in range(1, n_pasos+1):
        Z = np.random.standard_normal()
        S[t] = S[t-1] * np.exp((mu - 0.5*sigma**2)*dt +
                               sigma*np.sqrt(dt)*Z)
    return S
```

6.7. Valuacion de Opciones

```python
def valor_call_mc(S0, K, T, r, sigma, n_sim=100000):
    """
    S0: precio actual
    K: strike
    T: tiempo hasta vencimiento
    r: tasa libre de riesgo
    sigma: volatilidad
    """
    # Precio final
    Z = np.random.standard_normal(n_sim)
    ST = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)

    # Payoff: max(ST - K, 0)
    payoffs = np.maximum(ST - K, 0)

    # Valor presente
    valor = np.exp(-r*T) * np.mean(payoffs)
    return valor
```

6.8. Convergencia y Error

Ley de los Grandes Numeros:

$$\text{Error} \propto \frac{1}{\sqrt{N}}$$

Para reducir el error a la mitad, necesitas $4\times$ mas simulaciones.

```python
# Error estandar
error = std / np.sqrt(n_simulaciones)
```

PUNTOS CLAVE:

- Error disminuye como $1/\sqrt{N}$
- Util para problemas de alta dimension
- Flexible para diferentes distribuciones
- Puede requerir muchas simulaciones
- Convergencia puede ser lenta

Cuando usar Monte Carlo:

- Integrales multidimensionales complejas
- Sistemas con muchas variables aleatorias
- Cuando no hay solucion analitica
- Simulacion de procesos estocasticos
- Analisis de riesgo financiero

7. Tabla de Referencia Rapida

7.1. Librerias Principales

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate, interpolate, optimize, stats
from scipy.integrate import solve_ivp, solve_bvp, odeint
from scipy.misc import derivative
```

7.2. Funciones Esenciales

Tarea	Funcion	Libreria
DERIVADAS		
Derivada numerica	`derivative(f, x)`	scipy.misc
Gradiente array	`np.gradient(y, x)`	numpy
Diferencias	`np.diff(y)/np.diff(x)`	numpy
INTEGRALES		
Integral definida	`integrate.quad(f, a, b)`	scipy.integrate
Datos - Trapecio	`np.trapz(y, x)`	numpy
Datos - Simpson	`integrate.simpson(y, x)`	scipy.integrate
EDOs		
Cond. iniciales	`solve_ivp(f, t_span, y0)`	scipy.integrate
Cond. frontera	`solve_bvp(f, bc, x, y)`	scipy.integrate
INTERPOLACION		
1D	`interpolate.interp1d(x, y)`	scipy.interpolate
2D dispersa	`interpolate.griddata(...)`	scipy.interpolate
AJUSTES		
Lineal	`stats.linregress(x, y)`	scipy.stats
Polinomial	`np.polyfit(x, y, grado)`	numpy
No lineal	`optimize.curve_fit(f, x, y)`	scipy.optimize
ESTADISTICA		
Media	`np.mean(datos)`	numpy
Desviacion	`np.std(datos, ddof=1)`	numpy
Error estandar	`stats.sem(datos)`	scipy.stats
IC 95 %	`stats.t.interval(0.95, ...)`	scipy.stats
MONTE CARLO		
Uniforme [0,1]	`np.random.uniform(0, 1, n)`	numpy
Normal (0,1)	`np.random.standard_normal(n)`	numpy
Normal (mu,sigma)	`np.random.normal(mu, s, n)`	numpy
Eleccion	`np.random.choice([...], n)`	numpy

7.3. Formulas Importantes

7.3.1. Diferencias Finitas

$$\begin{aligned}\text{Forward:} \quad & f'(x) \approx \frac{f(x+h) - f(x)}{h} \\ \text{Backward:} \quad & f'(x) \approx \frac{f(x) - f(x-h)}{h} \\ \text{Central:} \quad & f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}\end{aligned}$$

7.3.2. Integracion

$$\begin{aligned}\text{Trapecio:} \quad & \int f(x)dx \approx h\left[\frac{f_0}{2} + f_1 + \ldots + \frac{f_n}{2}\right] \\ \text{Simpson:} \quad & \int f(x)dx \approx \frac{h}{3}[f_0 + 4f_1 + 2f_2 + \ldots + f_n]\end{aligned}$$

7.3.3. Propagacion de Errores

$$\begin{aligned}\text{Suma:} \quad & \delta(x \pm y) = \sqrt{\delta x^2 + \delta y^2} \\ \text{Producto:} \quad & \frac{\delta(xy)}{xy} = \sqrt{\left(\frac{\delta x}{x}\right)^2 + \left(\frac{\delta y}{y}\right)^2} \\ \text{Potencia:} \quad & \delta(x^n) = |n \cdot x^{n-1}| \cdot \delta x\end{aligned}$$

7.3.4. Monte Carlo

$$\text{Error} \propto \frac{1}{\sqrt{N}} \tag{1}$$

Para reducir error a la mitad: $N \to 4N$

7.4. Criterios de Estabilidad (EDPs)

Ecuacion de Calor:

$$r = \frac{\alpha \cdot dt}{dx^2} \leq 0{,}5 \quad \text{para estabilidad}$$

7.5. Parametros Tipicos

7.6. Metodos para EDOs

Parametro	Valor Tipico	Notas
h (derivadas)	10^{-5} a 10^{-8}	Muy pequeno causa errores
n (integracion)	1000-10000	Depende suavidad
n (Monte Carlo)	10000-100000	Mas = mejor pero lento
Confianza (IC)	0.95	95 % estandar
ddof (std)	1	Para muestras

Metodo	Cuando usar
RK45	Problemas generales (default)
RK23	Problemas simples
DOP853	Alta precision requerida
Radau	Sistemas rigidos
BDF	Sistemas rigidos
LSODA	Adaptativo (rigidos/no rigidos)

8. Consejos para el Examen

8.1. Estrategia General

1. **Lee todo primero** - Identifica problemas faciles
2. **Gestiona tiempo** - No te atasques
3. **Muestra trabajo** - Codigo comentado vale mas
4. **Verifica resultados** - Tiene sentido?

8.2. Errores Comunes

- Olvidar importar librerias
- Simpson con n impar (necesita n par)
- No verificar estabilidad EDPs ($r \leq 0{,}5$)
- Usar h muy pequeno en derivadas
- No dar p0 en curve_fit
- Olvidar ddof=1 en std()
- Invertir args odeint vs solve_ivp

8.3. Template Rapido

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate, interpolate, optimize, stats

# Datos
x = np.array([...])
y = np.array([...])
```

```
8
9  # Procesamiento
10 resultado = ...
11
12 # Visualizacion
13 plt.plot(x, y, 'o-')
14 plt.xlabel('x'); plt.ylabel('y')
15 plt.title('Titulo'); plt.grid(True)
16 plt.show()
17
18 print(f"Resultado: {resultado}")
```

9. Ejercicios de Practica

9.1. Ejercicio 1: Derivadas

Calcular derivada de $f(x) = x^3 + 2x^2 - 5x + 1$ en $x = 2$

```
def f(x):
    return x**3 + 2*x**2 - 5*x + 1

from scipy.misc import derivative
dy = derivative(f, 2, dx=1e-6)
# Verificar: f'(2) = 3(4) + 4(2) - 5 = 15
```

9.2. Ejercicio 2: Integracion

Calcular $\int_0^{\pi} \sin(x)dx$

```
from scipy import integrate
resultado, error = integrate.quad(np.sin, 0, np.pi)
# Valor exacto: 2.0
```

9.3. Ejercicio 3: EDO

Resolver $dy/dt = -y$, $y(0) = 1$

```
from scipy.integrate import solve_ivp

def f(t, y):
    return -y

sol = solve_ivp(f, (0, 5), [1], dense_output=True)
# Solucion exacta: y = e^(-t)
```

9.4. Ejercicio 4: Ajuste

Ajustar $y = ax + b$

```
from scipy import stats

x = np.array([1, 2, 3, 4, 5])
y = np.array([2.1, 4.2, 5.9, 8.1, 10.2])

slope, intercept, r_value, p, stderr = stats.linregress(x, y)
```

9.5. Ejercicio 5: Monte Carlo

Estimar π con 100,000 puntos

```
n = 100000
x = np.random.uniform(0, 1, n)
y = np.random.uniform(0, 1, n)
dentro = (x**2 + y**2) <= 1
pi_est = 4 * np.sum(dentro) / n
```

10. Checklist Pre-Examen

DERIVADAS:

☐ Forward/Central

☐ derivative()

☐ Gradiente

☐ h=1e-5 a 1e-8

INTEGRALES:

☐ Trapecio/Simpson

☐ quad()

☐ trapz()

☐ Dobles/Triples

EDOs:

☐ solve_ivp

☐ solve_bvp

☐ Convertir orden n

☐ RK45/Radau/BDF

EDPs:

☐ Diferencias finitas

☐ r ≤ 0.5

☐ Cond. iniciales/frontera

INTERPOLACION:

☐ Lineal/Cubica

☐ interp1d()

☐ griddata()

AJUSTES:

☐ linregress()

☐ polyfit()

☐ curve_fit()

☐ R^2

☐ χ^2

INCERTIDUMBRES:

☐ sem()

☐ t.interval()

☐ Propagacion

☐ Bootstrap

MONTE CARLO:

☐ Muestreo

☐ Error $\sim 1/\sqrt{N}$

☐ Integracion

☐ Generadores

MUCHO EXITO EN TU EXAMEN DEL 18/12/2025!

Creado: Diciembre 2024
Ultima actualizacion: 10/12/2024