Computational Physics Cheat Sheet

Open Book Final Exam Reference
December 10, 2024

Numerical Errors

Types of Errors:

- **Rounding Errors:** Errors in how numbers are stored/manipulated.
- **Approximation Errors:** Errors due to approximations in methods.

Machine Precision:

64-bit float: $\epsilon_M \sim 10^{-16}$, Fractional error: $\sigma = C|x|$

Tips:

- Avoid comparisons like `if (a == b)`; use a tolerance:

$$|\text{a - b}| < \delta$$

Propagation of Errors: Errors propagate statistically, especially in numerical derivatives:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h} \quad \text{(Danger zone!)}$$

Simple Integration Techniques

Trapezoidal Rule:

$$I(a,b) \approx h\left[\frac{f(a)+f(b)}{2} + \sum_{k=1}^{N-1} f(a+kh)\right]$$

where $h = \frac{b-a}{N}$.

Simpson's Rule:

$$I(a,b) \approx \frac{h}{3}\left[f(a)+f(b)+4\sum_{k\text{ odd}} f(a+kh)+2\sum_{k\text{ even}} f(a+kh)\right]$$

Python Implementation:

```
def simpsons(f, a, b, N):
    if N % 2 != 0:
        raise ValueError("N must be even for Simpson's rule.")
    h = (b - a) / N
    integral = f(a) + f(b)
    for k in range(1, N, 2):
        integral += 4 * f(a + k * h)
    for k in range(2, N-1, 2):
        integral += 2 * f(a + k * h)
    return (h / 3) * integral
```

Error Estimation:

- Trapezoidal: $\epsilon = \frac{h^2}{12}\left[f'(a) - f'(b)\right]$
- Simpson: $\epsilon = \frac{h^4}{180}\left[f'''(a) - f'''(b)\right]$

Adaptive Integration:

- Calculate I_N, double N, compute again.
- Error formula for trapezoidal: $\epsilon = \frac{I_{2N} - I_N}{3}$

Gaussian Quadrature

Formula:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N} w_k f(x_k)$$

where x_k are roots of the Legendre polynomial $P_N(x)$, and

$$w_k = \left.\frac{2}{(1-x^2)\left(\frac{dP_N}{dx}\right)^2}\right|_{x=x_k}.$$

Legendre Polynomials: Defined recursively:

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_{N+1}(x) = \frac{(2N+1)xP_N(x) - NP_{N-1}(x)}{N+1}$$

Pros:

- Converges quickly: error decreases as $O(1/N^2)$.
- Accurate for high-order polynomials.

Cons:

- Requires smooth functions.
- Difficult to estimate error.

Python Code Snippets

```
# Trapezoidal Rule
def trapezoidal_rule(f, a, b, N):
    h = (b − a) / N
    integral = 0.5 ∗ (f(a) + f(b))
    for k in range(1, N):
        integral += f(a + k ∗ h)
    return h ∗ integral

# Gaussian Quadrature
from scipy.special import roots_legendre
def gaussian_quadrature(f, a, b, N):
    x, w = roots_legendre(N)
    return sum(w ∗ f(0.5∗(b−a)∗x + 0.5∗(b+a))) ∗ 0.5 ∗ (b−a)
```

Derivatives and Interpolation

Forward Difference Approximation:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad \text{Error: } O(h)$$

Central Difference Approximation:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \quad \text{Error: } O(h^2)$$

Python Code for Numerical Derivatives:

```
# Forward difference
def forward_diff(f, x, h):
    return (f(x + h) - f(x)) / h

# Central difference
def central_diff(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)
```

Fourier Transforms

Discrete Fourier Transform (DFT):

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right)$$

Inverse Discrete Fourier Transform (iDFT):

$$y_n = \frac{1}{N}\sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right)$$

Fast Fourier Transform (FFT): Uses a divide-and-conquer strategy:

$$c_k = E_k + \omega^k O_k, \quad \omega = \exp\left(-i\frac{2\pi}{N}\right)$$

Python Code for DFT and FFT:

```
import numpy as np

# Compute DFT
def dft(y):
    N = len(y)
    c = np.zeros(N, dtype=complex)
    for k in range(N):
        for n in range(N):
            c[k] += (y[n] *
                     np.exp(-2j * np.pi * k * n / N))
        return c

# Compute FFT
from numpy.fft import fft
def fft_example(y):
    return fft(y)
```

2D Fourier Transform: For a 2D grid y_{mn}:

$$c_{k\ell} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} y_{mn} \exp\left[-i2\pi\left(\frac{km}{M} + \frac{\ell n}{N}\right)\right]$$

Inverse 2D Fourier Transform:

$$y_{mn} = \frac{1}{MN}\sum_{k=0}^{M-1}\sum_{\ell=0}^{N-1} c_{k\ell} \exp\left[i2\pi\left(\frac{km}{M} + \frac{\ell n}{N}\right)\right]$$

Solving Simple ODEs

General Form:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t), \quad \text{with initial condition } x(t=0) = x_0.$$

—

Euler Method

- Start at $t = t_0,\ x = x_0$.
- Discretize into time steps t_i with constant spacing h.
- At each step:

$$x_i = x_{i-1} + hf(x_{i-1}, t_{i-1}).$$

Error:

- Local error: $O(h^2)$.
- Global error: $O(h)$.

Python Code: Euler Method

```
# Euler Method Implementation
import numpy as np
import matplotlib.pyplot as plt

def f(x, t):
    return -x # Example ODE: dx/dt = -x

t0 = 0 # Initial time
x0 = 1 # Initial condition
h = 0.1 # Step size
t_end = 5 # End time

t_points = np.arange(t0, t_end, h)
x_points = []

x = x0
for t in t_points:
    x_points.append(x)
    x += h * f(x, t)

plt.plot(t_points, x_points)
plt.xlabel('Time t')
plt.ylabel('x(t)')
plt.title('Euler Method Solution')
plt.show()
```

—

Runge-Kutta Methods

RK2 (Second-Order Method):

$$\begin{aligned} k_1 &= hf(x,t), \\ k_2 &= hf\left(x + \frac{k_1}{2}, t + \frac{h}{2}\right), \\ x(t+h) &= x(t) + k_2. \end{aligned}$$

Python Code: RK2 Method

```
# Runge-Kutta 2nd Order (RK2) Method Implementation
import numpy as np
import matplotlib.pyplot as plt

def f(x, t):
    return -x # Example ODE: dx/dt = -x

t0 = 0 # Initial time
x0 = 1 # Initial condition
h = 0.1 # Step size
t_end = 5 # End time

t_points = np.arange(t0, t_end, h)
x_points = []

x = x0
for t in t_points:
    x_points.append(x)
    k1 = h * f(x, t)
    k2 = h * f(x + 0.5 * k1, t + 0.5 * h)
    x += k2

plt.plot(t_points, x_points)
plt.xlabel('Time t')
plt.ylabel('x(t)')
plt.title('RK2 Method Solution')
plt.show()
```

RK4 (Fourth-Order Method):

$$\begin{aligned} k_1 &= hf(x,t), \\ k_2 &= hf\left(x + \frac{k_1}{2}, t + \frac{h}{2}\right), \\ k_3 &= hf\left(x + \frac{k_2}{2}, t + \frac{h}{2}\right), \\ k_4 &= hf(x + k_3, t + h), \\ x(t+h) &= x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

Python Code: RK4 Method

```
# Runge-Kutta 4th Order (RK4) Method Implementation
import numpy as np
import matplotlib.pyplot as plt

def f(x, t):
    return -x # Example ODE: dx/dt = -x

t0 = 0 # Initial time
x0 = 1 # Initial condition
h = 0.1 # Step size
t_end = 5 # End time

t_points = np.arange(t0, t_end, h)
x_points = []

x = x0
for t in t_points:
    x_points.append(x)
    k1 = h * f(x, t)
    k2 = h * f(x + 0.5 * k1, t + 0.5 * h)
    k3 = h * f(x + 0.5 * k2, t + 0.5 * h)
    k4 = h * f(x + k3, t + h)
    x += (k1 + 2 * k2 + 2 * k3 + k4) / 6

plt.plot(t_points, x_points)
plt.xlabel('Time t')
plt.ylabel('x(t)')
plt.title('RK4 Method Solution')
plt.show()
```

Error:

- RK2: $O(h^3)$ local, $O(h^2)$ global.
- RK4: $O(h^5)$ local, $O(h^4)$ global.

—

Leapfrog Method

Formula:

$$x_{i+1} = x_{i-1} + 2hf(x_i, t_i)$$

Python Code: Leapfrog Method

```
# Leapfrog Method Implementation
import numpy as np
import matplotlib.pyplot as plt

def f(x, t):
    return -x # Example ODE: dx/dt = -x

t0 = 0 # Initial time
x0 = 1 # Initial condition
h = 0.1 # Step size
t_end = 5 # End time

t_points = np.arange(t0, t_end, h)
x_points = []

# Initialize x at two time steps
x_prev = x0
x = x0 + h * f(x0, t0) # Use Euler's method for the first step

x_points.append(x_prev)
x_points.append(x)

for t in t_points[1:-1]:
    x_next = x_prev + 2 * h * f(x, t)
    x_prev = x
    x = x_next
    x_points.append(x)

plt.plot(t_points, x_points[:len(t_points)])
plt.xlabel('Time t')
plt.ylabel('x(t)')
plt.title('Leapfrog Method Solution')
plt.show()
```

Error: $O(h^2)$ global error.

—

Bulirsch-Stoer Method

- Refines grid spacing to cancel higher-order errors.
- Combines Richardson extrapolation with modified midpoint methods.

Error: Improves by 2 orders for each refinement.

—

Comparison of Methods

- **Euler Method:** Simple but less accurate, larger error, suitable for quick approximations.
- **RK2 and RK4:** Higher accuracy with moderate computational effort, RK4 being more accurate than RK2.
- **Leapfrog Method:** Time-reversible, good for systems where conservation of energy is important.
- **Bulirsch-Stoer:** Very high accuracy, useful for problems requiring very precise solutions.

Solving Simple PDEs

Classification of PDEs

General PDE form:

$$\alpha\frac{\partial^2\phi}{\partial x^2}+\beta\frac{\partial^2\phi}{\partial x\partial y}+\gamma\frac{\partial^2\phi}{\partial y^2}+\delta\frac{\partial\phi}{\partial x}+\varepsilon\frac{\partial\phi}{\partial y}=f.$$

Discriminant:

$$\Delta=\beta^2-4\alpha\gamma$$

- $\Delta=0$: Parabolic PDE (e.g., diffusion equation).
- $\Delta<0$: Elliptic PDE (e.g., Laplace or Poisson equation).
- $\Delta>0$: Hyperbolic PDE (e.g., wave equation).

—

Elliptic PDEs: Solvers for the Poisson Equation

$$\nabla^2\phi=\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}=f.$$

Discretization:

$$\frac{\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)-4\phi(x,y)}{a^2}\approx f(x,y).$$

Jacobi Relaxation:

$$\phi_{\text{new}}(x,y)=\frac{1}{4}\left[\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right].$$

Gauss-Seidel Method: Update ϕ on-the-fly:

$$\phi(x,y)\leftarrow\frac{1}{4}\left[\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right].$$

Overrelaxation:

$$\phi_{\text{new}}(x,y)=(1+\omega)\phi_{\text{Jacobi}}(x,y)-\omega\phi_{\text{old}}(x,y).$$

—

Parabolic PDEs: Heat Equation

Heat equation:

$$\frac{\partial T}{\partial t}=\kappa\frac{\partial^2T}{\partial x^2}.$$

Discretization:

$$\text{Spatial: }\frac{\partial^2T}{\partial x^2}\approx\frac{T_{m+1}-2T_m+T_{m-1}}{a^2},$$

$$\text{Temporal: }\frac{\partial T}{\partial t}\approx\frac{T_m^{n+1}-T_m^n}{h}.$$

Explicit FTCS (Forward Time Centered Space):

$$T_m^{n+1}=T_m^n+\frac{\kappa h}{a^2}\left(T_{m+1}^n-2T_m^n+T_{m-1}^n\right).$$

Implicit FTCS: Solve simultaneous equations for stability:

$$\mathbf{A}T^{n+1}=\mathbf{B}T^n,$$

where $\mathbf{A}$ and $\mathbf{B}$ are coefficient matrices.
Crank-Nicolson Method:

$$T_m^{n+1}-\frac{\kappa h}{2a^2}\left(T_{m+1}^{n+1}-2T_m^{n+1}+T_{m-1}^{n+1}\right)=T_m^n+\frac{\kappa h}{2a^2}\left(T_{m+1}^n-2T_m^n+T_{m-1}^n\right)$$

—

Hyperbolic PDEs: Wave Equation

Wave equation:

$$\frac{\partial^2\phi}{\partial t^2}=c^2\frac{\partial^2\phi}{\partial x^2}.$$

Discretization:

$$\frac{\phi_m^{n+1}-2\phi_m^n+\phi_m^{n-1}}{h^2}=c^2\frac{\phi_{m+1}^n-2\phi_m^n+\phi_{m-1}^n}{a^2}.$$

Crank-Nicolson for Stability: Combine explicit and implicit methods to maintain accuracy and stability.

—

Python Code: FTCS Method for Heat Equation

```
import numpy as np

# Parameters
kappa = 1.0 # Thermal conductivity
a = 0.1 # Grid spacing
h = 0.01 # Time step
L = 1.0 # Length of rod
M = int(L / a)
T = np.zeros(M+1)

# FTCS update
for n in range(steps):
    T_new = np.copy(T)
    for m in range(1, M):
        T_new[m] = T[m] + (kappa * h / a**2) * (T[m+1] -
            2*T[m] + T[m-1])
    T = T_new
```

Advanced ODEs and PDEs

Stability and Von Neumann Analysis

For stability, we analyze how a single Fourier mode evolves in time. Growth factors must satisfy:

$$\left|\frac{\widehat{T}_k^{n+1}}{\widehat{T}_k^n}\right|\le 1.$$

FTCS Explicit for Diffusion:

$$T_m^{n+1}=T_m^n+\frac{\kappa h}{a^2}\left(T_{m+1}^n-2T_m^n+T_{m-1}^n\right).$$

Stability criterion:

$$\boxed{h\le\frac{a^2}{2\kappa}}$$

FTCS Explicit for Waves:

$$\frac{\partial^2\phi}{\partial t^2}=c^2\frac{\partial^2\phi}{\partial x^2}.$$

Growth factors:

$$|\lambda_\pm|^2=1+h^2r^2\ge 1.$$

FTCS is always unstable for hyperbolic equations.
FTCS Implicit: Implicit methods improve stability but can lead to decaying solutions.

—

Crank-Nicolson Method

Hyperbolic PDEs:

$$\phi_m^{n+1}-\frac{h}{2}\psi_m^{n+1}=\phi_m^n+\frac{h}{2}\psi_m^n,$$

$$\psi_m^{n+1}-\frac{h}{2}\frac{c^2}{a^2}\left(\phi_{m+1}^{n+1}+\phi_{m-1}^{n+1}-2\phi_m^{n+1}\right)=\psi_m^n+\frac{h}{2}\frac{c^2}{a^2}\left(\phi_{m+1}^n+\phi_{m-1}^n-2\phi_m^n\right).$$

Growth factors:

$$|\lambda_\pm|=1\quad\text{(neither grows nor decays).}$$

Verlet Method

$$\frac{d^2x}{dt^2} = \frac{F(x)}{m}, \quad \frac{dx}{dt} = v.$$

Update:

$$v\left(t + \frac{h}{2}\right) = v(t) + \frac{h}{2}\frac{F(x(t))}{m},$$

$$x(t+h) = x(t) + hv\left(t + \frac{h}{2}\right),$$

$$v(t+h) = v\left(t + \frac{h}{2}\right) + \frac{h}{2}\frac{F(x(t+h))}{m}.$$

—

Spectral Methods

Key Idea: Use an orthogonal function basis (e.g., Fourier series) to project and solve PDEs. Suitable bases include:

- $\sin(n\pi x/L)$ (boundary condition: zero at edges),
- $\cos(n\pi x/L)$ (zero derivative at edges),
- $\exp(in\pi x/L)$ (periodic),
- Chebyshev polynomials (flexible boundary conditions).

Fourier Transform:

$$f(x) = \sum_{n=-\infty}^{\infty} \widehat{f}_n \exp(ik_n x), \quad \frac{\partial f}{\partial x} \to ik_n \widehat{f}_n.$$

Example: Numerical differentiation using FFT:

```
import numpy as np
from numpy.fft import rfft, irfft

L = 2.0 # domain length
Delta = 0.1 # width of Gaussian
nx = 200 # grid points
x = np.linspace(0, L, nx)

# Gaussian function
def f(x): return np.exp(-(x-L/2)**2 / Delta**2)

# FFT and differentiation
fhat = rfft(f(x))
k = np.arange(nx//2+1) * 2 * np.pi / L
df_dx = irfft(1j * k * fhat)
```

Boundary Value Problems

Shooting Method: Use ODE integration and adjust parameters (e.g., initial velocity) to satisfy boundary conditions.

Random Processes and Monte Carlo Techniques

Random Number Generation

Pseudo-Random Number Generators (PRNG):

- Computers can't generate true random numbers but use algorithms to simulate randomness.
- Key properties of PRNGs:
  - Follows a desired distribution.
  - Long period and uncorrelated.
  - Fast to generate.
- Example: Linear Congruential Generator (LCG):

$$x_{i+1} = (ax_i + c) \mod m$$

where a, c, m are constants, and x_0 is the seed.

Python PRNGs:

- Python uses the Mersenne Twister (better than LCG).
- Useful methods:
  - `random()` - Random float in $[0, 1)$.
  - `randrange(m, n)` - Random integer from m to $n-1$.

Monte Carlo Integration

Advantages:

- Works well for high-dimensional integrals.
- Handles complicated domains and pathological functions.
- Convergence: $\epsilon \propto 1/N^{1/2}$, independent of dimensionality.

—

Hit-or-Miss Monte Carlo

- Randomly pick N points in a region.
- Count points k under the curve.
- Integral:

$$I \approx \frac{kA}{N}, \quad A = \text{area of bounding box.}$$

Error Estimate:

$$\sigma = \sqrt{\frac{(A-I)I}{N}}.$$

Python Example:

```
import numpy as np

def f(x): return np.sin(1 / ((2 - x) * x))**2

N = 10000 # number of points
k = 0 # points under the curve
a, b = 0., 2.

for i in range(N):
    x_samp = a + (b - a) * np.random.random()
    y_samp = np.random.random()
    if y_samp <= f(x_samp):
        k += 1

A = (b - a) * 1.0
I = A * k / N
print("I = {:.6e}".format(I))

# Error
sigma_HM = np.sqrt(I * (A - I) / N)
print("Error = {:.6e}".format(sigma_HM))
```

—

Mean Value Monte Carlo

- Randomly sample x_i points.
- Integral:

$$I \approx (b-a)\langle f(x)\rangle, \quad \langle f(x)\rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i).$$

Error Estimate:

$$\sigma = (b-a)\sqrt{\frac{\mathrm{Var}(f)}{N}}, \quad \mathrm{Var}(f) = \langle f^2\rangle - \langle f\rangle^2.$$

Python Example:

```
k, k2 = 0, 0 # mean and variance
for i in range(N):
    x = (b - a) * np.random.random()
    k += f(x)
    k2 += f(x)**2

I = k * (b - a) / N
var = k2 / N - (k / N)**2
sigma_MV = (b - a) * np.sqrt(var / N)
print("I = {:.6e}, Error = {:.6e}".format(I, sigma_MV))
```

—

Importance Sampling

- Useful for functions with divergences or peaks.
- Use a weight function $w(x)$ to bias sampling:

$$I = \int_a^b f(x)\,dx = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \int_a^b w(x)\,dx.$$

- Choose $w(x)$ to match the integrand's behavior.

Example: For $I = \int_0^1 \frac{x^{-1/2}}{1+\exp(x)}\,dx$:

- Let $w(x) = x^{-1/2}$. Transform the integral:

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{1+\exp(x_i)}.$$

Simulated Annealing and Global Optimization

Simulated Annealing: Overview

Simulated annealing is a Monte Carlo-based algorithm for finding the **global minimum** of a function by simulating a cooling process.
Boltzmann Probability: For a physical system in equilibrium at temperature T:

$$P(E_i) = \frac{\exp\left(-\beta E_i\right)}{Z}, \quad Z = \sum_i \exp\left(-\beta E_i\right), \quad \beta = \frac{1}{k_B T}.$$

As $T \to 0$, the system converges to the ground state:

$$P(E_i) = \begin{cases} 1 & \text{if } E_i = 0, \\ 0 & \text{if } E_i > 0. \end{cases}$$

Key Idea:

- Simulate the system at high temperature and gradually cool it down.
- Allow the system to escape local minima by accepting some high-energy states with nonzero probability.
- Gradually decrease temperature T (slow cooling) to ensure convergence to the global minimum.

—

Algorithm

1. Initialize the system at temperature T_0 and state (x_0, y_0). 2. Generate a trial state (x', y') using random Gaussian steps:

$$(x', y') = (x + \delta x, y + \delta y),$$

where $\delta x, \delta y \sim \mathcal{N}(0, \sigma^2)$. 3. Accept the move with probability:

$$P = \exp\left(-\frac{\Delta E}{k_B T}\right), \quad \Delta E = E(x', y') - E(x, y).$$

4. Gradually lower the temperature:

$$T = T_0 \exp\left(-\frac{t}{\tau}\right),$$

where τ is the time constant for cooling. 5. Stop when $T \leq T_{\text{final}}$ or the state stops changing.
—

Python Code: Simulated Annealing

```
import numpy as np

# Define the function to minimize
def f(x, y):
    return x**2 − np.cos(4 * np.pi * x) + (y − 1)**2

# Simulated Annealing functions
def draw_normal(sigma):
    theta = 2 * np.pi * np.random.random()
    z = np.random.random()
    r = (−2 * sigma**2 * np.log(1 − z))**0.5
    return r * np.cos(theta), r * np.sin(theta)

def get_temp(T0, tau, t):
    return T0 * np.exp(−t / tau)

def decide(new_val, old_val, temp):
    if np.random.random() > np.exp(−(new_val − old_val) /
        temp):
        return 0 # reject
    return 1 # accept

# Parameters
tau = 10000
T_start = 1.0
T_final = 0.001
X_start, Y_start = 2.0, 2.0

# Simulated Annealing loop
x, y = [X_start], [Y_start]
T = T_start
time = 0

while T > T_final:
    time += 1
    T = get_temp(T_start, tau, time)
    dx, dy = draw_normal(1.0)
    x_new, y_new = x[−1] + dx, y[−1] + dy
    if decide(f(x_new, y_new), f(x[−1], y[−1]), T):
        x.append(x_new)
        y.append(y_new)
    else:
        x.append(x[−1])
        y.append(y[−1])

print(f"The global minimum is estimated at (x,y) = ({x[−1]},
    {y[−1]})")
print(f"f({x[−1]:.2f},{y[−1]:.2f}) = {f(x[−1],y[−1]):.2f}")
```

—

SciPy Implementation

Simulated annealing is also available in SciPy via the `dual_annealing()` function.
Example:

```
from scipy.optimize import dual_annealing

# Function to minimize
def fForSciPy(v):
    x_, y_ = v[0], v[1]
    return x_**2 − np.cos(4 * np.pi * x_) + (y_ − 1)**2

# Define bounds
bounds = [[−2.0, 2.0], [0.0, 4.0]]

# Perform simulated annealing
result = dual_annealing(fForSciPy, bounds)

# Summarize the result
solution = result['x']
print(f"Solution: f({solution}) = {fForSciPy(solution):.5f}")
```

—

Comparison with Other Global Optimization Methods

Other methods available in SciPy:

- Basin Hopping: `scipy.optimize.basinhopping`
- Differential Evolution: `scipy.optimize.differential_evolution`
- Simplicial Homology Global Optimization: `scipy.optimize.shgo`
- Brute Force Grid Search: `scipy.optimize.brute`

Example Implementations

1. Adaptive Trapezoidal Rule

Description: Performs numerical integration with automatic error refinement.

```
def adaptive_trapezoidal(f, a, b, tol=1e−6):
    N = 1
    h = (b − a)
    integral = h * (f(a) + f(b)) / 2
    error = float('inf')

    while error > tol:
        h /= 2
        N *= 2
```

```
        new_integral = 0.5 * integral
        for k in range(1, N, 2):
            new_integral += h * f(a + k * h)
        error = abs(new_integral - integral) / 3
        integral = new_integral

    return integral
```

—

2. RK4 for Coupled ODEs

Description: Solves a system of coupled first-order ODEs.

```
def rk4(f, x, t, h):
    k1 = h * f(x, t)
    k2 = h * f(x + 0.5 * k1, t + 0.5 * h)
    k3 = h * f(x + 0.5 * k2, t + 0.5 * h)
    k4 = h * f(x + k3, t + h)
    return x + (k1 + 2 * k2 + 2 * k3 + k4) / 6

# Example: Solving dx/dt = -x, dy/dt = -2y
def coupled_eqs(X, t):
    x, y = X
    return np.array([-x, -2 * y])

t = 0
h = 0.1
x = np.array([1, 1]) # Initial values for x, y
for _ in range(10):
    x = rk4(coupled_eqs, x, t, h)
    t += h
    print(t, x)
```

—

3. FFT of a Gaussian Function

Description: Computes the FFT and its inverse for a Gaussian function.

```
import numpy as np
import matplotlib.pyplot as plt

L, N = 10, 256
x = np.linspace(-L / 2, L / 2, N)
f = np.exp(-x**2)

# FFT
fhat = np.fft.fftshift(np.fft.fft(f))
k = np.fft.fftshift(np.fft.fftfreq(N, d=(x[1] - x[0])))

# Plot
plt.plot(k, np.abs(fhat), label="FFT Magnitude")
plt.legend()
plt.show()
```

—

4. Jacobi Method for Poisson Equation

Description: Solves $\nabla^2\phi = f(x, y)$ on a 2D grid.

```
def jacobi_2d(f, phi, h, tol=1e-6):
    error = float('inf')
    while error > tol:
        phi_new = phi.copy()
        phi_new[1:-1, 1:-1] = 0.25 * (
            phi[:-2, 1:-1] + phi[2:, 1:-1] +
            phi[1:-1, :-2] + phi[1:-1, 2:] -
            h**2 * f[1:-1, 1:-1])
        error = np.max(np.abs(phi_new - phi))
        phi = phi_new
    return phi
```

—

5. Monte Carlo Integration: Hit-or-Miss

Description: Computes the integral of a function using random sampling.

```
def hit_or_miss(f, a, b, N):
    hits = 0
    for _ in range(N):
        x = np.random.uniform(a, b)
        y = np.random.uniform(0, 1)
        if y <= f(x):
            hits += 1
    return (b - a) * hits / N
```

—

6. Simulated Annealing

Description: Finds the global minimum of a 2D function.

```
def simulated_annealing(f, x0, T_start, T_end, cooling_rate):
    x = x0
    T = T_start
    while T > T_end:
        x_new = x + np.random.normal(scale=T)
        delta_E = f(x_new) - f(x)
        if delta_E < 0 or np.random.random() < np.exp(-delta_E
            / T):
            x = x_new
        T *= cooling_rate
    return x

# Example: f(x) = x**2
min_x = simulated_annealing(lambda x: x**2, x0=10, T_start=1,
    T_end=1e-6, cooling_rate=0.9)
print(f"Minimum x = {min_x}")
```

Common NumPy and Pyplot Methods

NumPy Methods:

- `numpy.random.normal`: Generate random samples from a normal (Gaussian) distribution.

```
import numpy as np
data = np.random.normal(loc=0, scale=1, size=1000) # Mean=0,
    Std=1
print(data[:5])
```

- `numpy.loadtxt`: Load data from a text file and optionally unpack columns.

```
# Example file: "example.txt" with two columns
col1, col2 = np.loadtxt("example.txt", delimiter=",",
    unpack=True)
print(col1[:5], col2[:5])
```

- `numpy.arange` vs `numpy.linspace`:

```
np.arange(0, 1, 0.2) # Steps by 0.2
# Output: [0. , 0.2, 0.4, 0.6, 0.8]

np.linspace(0, 1, 5) # Divides into 5 points
# Output: [0. , 0.25, 0.5, 0.75, 1. ]
```

Pyplot Methods:

- `pyplot.bar`: Create bar plots.

```
import matplotlib.pyplot as plt
labels = ['A', 'B', 'C']
values = [10, 20, 15]
plt.bar(labels, values)
plt.show()
```

- `pyplot.hist`: Plot a histogram.

```
plt.hist(data, bins=30, alpha=0.5, edgecolor="black")
plt.show()
```

Variable Transformations

Variable transformations involve replacing one variable with a function of another to simplify problems or change the domain.
Example: Transform $x \in [0, 1]$ to $u \in [-1, 1]$:

$$x = \frac{b-a}{2}u + \frac{b+a}{2}$$

Code Example:

```
import numpy as np

# Transform u in [-1, 1] to x in [0, 1]
u = np.linspace(-1, 1, 100)
a, b = 0, 1
x = 0.5 * (b - a) * u + 0.5 * (b + a)

# Back-transform
u_recovered = 2 * (x - 0.5 * (b + a)) / (b - a)
```

Example: Polar Coordinates

$$x = r\cos\theta, \quad y = r\sin\theta$$

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.linspace(0, 2 * np.pi, 100)
r = np.sqrt(theta) # Example: r depends on theta
x = r * np.cos(theta)
y = r * np.sin(theta)

plt.plot(x, y)
plt.title("Polar to Cartesian Transformation")
plt.show()
```

Interpolation

Interpolation allows us to estimate values between data points. Here is an example using both linear interpolation and cubic spline interpolation for the function:

$$y = \frac{1}{2+x^2}$$

```
from scipy.interpolate import CubicSpline, interp1d
import matplotlib.pyplot as plt
import numpy as np

# Fake data to interpolate from
x = np.arange(-10, 10)
y = 1./(2.+x**2)

# Points at which we want to interpolate
xs = np.arange(-9, 9, 0.1)

# Apply Linear interpolation
linear_int = interp1d(x, y)
ys_lin = linear_int(xs)

# Apply cubic spline interpolation
cs = CubicSpline(x, y)
ys_cub = cs(xs)

# Plot interpolations and data
plt.plot(xs, ys_lin, 'o', label='linear')
plt.plot(xs, ys_cub, 'o', label='cubic spline')
plt.plot(x, y, '*', label='data')
plt.legend()
plt.show()
```

Steps in the Code:

- `np.arange(-10, 10)`: Generates the original data points (x and y).
- `interp1d(x, y)`: Creates a linear interpolation function.
- `CubicSpline(x, y)`: Creates a cubic spline interpolation function.
- `xs`: The points where the interpolation is evaluated.
- Plot: The interpolated points (`ys_lin` and `ys_cub`) are plotted along with the original data.

Applications:

- Use `interp1d` for faster and simpler interpolations.
- Use `CubicSpline` for smoother curves that respect the shape of the data.