Computational Physics CheatSheet

Samyak Rai

November 09, 2025

1 Root Finding

Bisection Method

```
def bisection(f, a, b, tol=1e-8, maxiter=100):
    fa, fb = f(a), f(b)
    if fa == 0: return a
    if fb == 0: return b
    if fa * fb > 0:
        raise ValueError("f(a) and f(b) must have opposite signs")
    for _ in range(maxiter):
        c = 0.5*(a+b)
        fc = f(c)
    if abs(b-a) < tol or abs(fc) < tol:
        return c
    if fa * fc < 0:
        b, fb = c, fc
    else:
        a, fa = c, fc
    return 0.5*(a+b)
```

Newton Raphson Method

```
def newton(f, df, x0, tol=1e-10, maxiter=50):
    x = x0
    for i in range(maxiter):
        fx = f(x)
        dfx = df(x)
        if abs(dfx) < 1e-16:
            raise ZeroDivisionError("Derivative too small")
        dx = fx/dfx
        x = x - dx
        if abs(dx) < tol:
            return x
    return x

def newton_fd(f, x0, tol=1e-10, maxiter=50, h=1e-6):
    def df(x):
        return (f(x+h)-f(x-h))/(2*h)
    return newton(f, df, x0, tol, maxiter)
```

2 Numerical Integration Techniques

Trapezoidal Rule

```
def trapezoid(f, a, b, n=1000):
    x = np.linspace(a, b, n+1)
    y = f(x)
    h = (b-a)/n
    return (h/2) * (y[0] + 2*np.sum(y[1:-1]) + y[-1])
```

Simpson Rule

```
def simpson(f, n, a, b):
    if n % 2 == 1:
        raise ValueError("n must be even for Simpson's rule")
    h = (b - a) / n
    x = np.linspace(a, b, n + 1)
    y = f(x)
    intgr = h/3*(y[0]+y[-1] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-2:2]))
    return intgr
```

3 Ordinary Differntial Equations

Verlet Method

```
def velocity_verlet(x0, v0, a_func, t):
    t = np.asarray(t)
    n = t.size
    x = np.empty((n,) + np.shape(x0))
    v = np.empty((n,) + np.shape(v0))
    x[0] = x0
    v[0] = v0
    for i in range(n-1):
        dt = t[i+1] - t[i]
        ai = a_func(x[i], t[i])
        x[i+1] = x[i] + v[i]*dt + 0.5*ai*dt*dt
        a_next = a_func(x[i+1], t[i+1])
        v[i+1] = v[i] + 0.5*(ai + a_next)*dt
    return x, v
```

Runga Kutta 2 (Midpoint)

```
def rk2(f, y0, t):
    y0 = np.asarray(y0, dtype=float)
    t = np.asarray(t)
    n = t.size
    y = np.empty((n,) + y0.shape)
    y[0] = y0
    for i in range(n-1):
        dt = t[i+1] - t[i]
        k1 = f(t[i], y[i])
        k2 = f(t[i] + 0.5*dt, y[i] + 0.5*dt*k1)
        y[i+1] = y[i] + dt*k2
    return y
```

RK4

```
def force(function, state):
    dxdt = state[1]
    dvxdt = function.x
    dydt = state[3]
    dvydt = function.y
    return np.array([dxdt, dvxdt, dydt, dvydt])

def rk4_step(f, state, t, dt):
    k1 = f(state, t)
    k2 = f(state + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = f(state + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = f(state + dt * k3, t + dt)
    return state + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

def Solver(f, state_initial, t0, t_final, dt):
    n_steps = int(np.ceil((t_final - t0) / dt))
    t_eval = np.linspace(t0, t_final, n_steps + 1)
    history = np.empty((n_steps + 1, len(state_initial)))
    history[0] = state_initial

    state = state_initial.copy()
    for n in range(n_steps):
        t = t_eval[n]
        state = rk4_step(f, state, t, dt)
        history[n + 1] = state

    return history, t_eval
```

Predictor Corrector

Predictor–corrector methods combine an explicit *predictor* (cheap, possibly less accurate) with an implicit or more accurate *corrector* that uses the predicted value. A typical workflow is PECE (Predict, Evaluate, Correct, Evaluate) where the final Evaluate updates the derivative used in the next step.

Simple Heun (PECE) — one-step example.

$$\text{Predictor:} \quad y_{n+1}^{(0)} = y_n + h\, f(t_n, y_n),$$

$$\text{Evaluate:} \quad f_{n+1}^{(0)} = f(t_{n+1}, y_{n+1}^{(0)}),$$

$$\text{Corrector:} \quad y_{n+1} = y_n + \frac{h}{2}\left(\, f(t_n, y_n) + f_{n+1}^{(0)} \,\right),$$

$$\text{Re-evaluate:} \quad f_{n+1}^{(1)} = f(t_{n+1}, y_{n+1}) \ \texttt{(Optional)}$$

This is PECE: Predictor (Euler) → Evaluate → Correct (trapezoid) → Evaluate.

One may iterate the corrector (use the newly computed y_{n+1} to recompute f_{n+1} and reapply the Corrector formula) until the change is below a tolerance.

4 1D Boundary Value Problems

Newton Shooting

```
def newton_shooting(func, x_start: float, x_end: float, y_start: float,
y_end: float, num_points: int, max_iterations: int):
    slope= 1.0
    dx = (x_end - x_start) / num_points

    for _ in range(max_iterations):
        initial_state = [y_start, slope]
        x_values, y_values = Solver(func, initial_state, x_start, x_end,
dx)

        error = y_values[-1] - y_end

        _, y_values_eps = Solver(func, [y_start, slope+ 1e-6],
        x_start, x_end, dx)
        error_eps = y_values_eps[-1] - y_end

        error_derivative = ((error_eps - error) / 1e-6) + 1e-9
        slope -= error / error_derivative

        if abs(error) < 1e-6:
            print(f"Converged with boundary error: {error:.2e}")
            break

    return x_values, y_values
```

Finte Diference Method

```
def fd_bvp_linear(p, q, r, a, b, alpha, beta, m):
    h = (b-a)/m
    x = np.linspace(a, b, m+1)
    # interior unknowns y1..y_{m-1}
    A = np.zeros((m-1, m-1))
    B = np.zeros(m-1)
    for i in range(1, m):
        xi = x[i]
        ai = 1.0/h**2 - p(xi)/(2*h)
        bi = -2.0/h**2 + q(xi)
        ci = 1.0/h**2 + p(xi)/(2*h)
        idx = i-1
        if idx > 0:
            A[idx, idx-1] = ai
            A[idx, idx] = bi
        if idx < m-2:
            A[idx, idx+1] = ci
            B[idx] = r(xi)

    # incorporate boundary values
    B[0] -= (1.0/h**2 - p(x[1])/(2*h))*alpha
    B[-1] -= (1.0/h**2 + p(x[-2])/(2*h))*beta
    y_interior = np.linalg.solve(A, B)
    y = np.empty(m+1)
    y[0] = alpha
    y[-1] = beta
    y[1:-1] = y_interior
    return x, y
```

5 2D Partial Differntial Equations

5.1 Iterative Methods

Jacobi Iteration

```
def Jacobi(V, s, delta, max_iter=5000, tol=1e-6):
    for it in tqdm(range(max_iter), desc= "Iteration: "):
        V_new = V.copy()
        for n in range(1, N+1):
            for m in range(1, M+1):
                V_new[n,m] = 0.25*(V[n+1,m] + V[n-1,m] + V[n,m+1] +
V[n,m-1] - s[n,m]*delta*delta)
        if np.linalg.norm(V_new-V)/np.linalg.norm(V_new) < tol:
            print(f"Converged at iteration {it}")
            return V_new
        V = V_new
    return V
```

Gauss-Seidel

```
def GaussSeidel(V, s, delta, max_iter=5000, tol=1e-6):
    for it in tqdm(range(max_iter), desc= "Iteration: "):
        V_prev = V.copy()
        for n in range(1, N+1):
            for m in range(1, M+1):
                V[n,m] = 0.25*(V[n+1,m] + V[n-1,m] + V[n,m+1] + V[n,m-1]
- s[n,m]*delta*delta)
        if np.linalg.norm(V - V_prev)/np.linalg.norm(V) < tol:
            print(f"Converged at iteration {it}")
            return V
    return V
```

Running the Algorithm for both cases,

```
N = 100
M = 100
Lx, Ly = 1.0, 1.0
dx = Lx/(N+1)
dy = Ly/(M+1)
V = np.zeros((N+2, M+2))
# setting up Boundary Conditions along y
y = np.linspace(0, Ly, M+2)
V[0, :] = np.sin(2*np.pi*y)
# Constant Term
s = (np.zeros((N+2, M+2)))

V_sol = Solver(V, s, dx) # if dx and dy are different change things a
bit
```

5.2 Matrix Formulation

```
# Define N, M and delta
N = 100
M = 100
dx = 1.0 / (N + 1)
dy = 1.0 / (M + 1)


# Create 1D finite difference matrices
def create_1d_laplacian(n):
    main_diag = np.full(n, -2)
    side_diag = np.full(n - 1, 1)
    return np.diag(main_diag) + np.diag(side_diag, 1) +
np.diag(side_diag, -1)


# 1D matrices
A_x = create_1d_laplacian(N)
A_y = create_1d_laplacian(M)
I_N = np.eye(N)
I_M = np.eye(M)

# 2D Laplacian using Kronecker products
D_x = np.kron(I_M, A_x)  # ∂²/∂x²
D_y = np.kron(A_y, I_N)  # ∂²/∂y²

L = D_x / (dx ** 2) + D_y / (dy ** 2)

# Boundary condition vectors
V_left = np.sin(2* np.pi * np.linspace(0,1,M))  # V(0,:)
V_right = np.zeros(M)  # V(N+1,:)
V_bottom = np.zeros(N)  # V(:,0)
V_top = np.zeros(N)  # V(:,M+1)

# Pre-scale by dx, dy once
V_left /= dx ** 2
V_right /= dx ** 2
V_bottom /= dy ** 2
V_top /= dy ** 2

# Create source term
x = np.linspace(dx, 1 - dx, N)
y = np.linspace(dy, 1 - dy, M)
X, Y = np.meshgrid(x, y, indexing="ij")

f = 0 * X * Y
f_vec = f.flatten()

# Copy to boundary-corrected RHS
f_bc = f_vec.copy()

for j in range(M):
    for i in range(N):
        k = j * N + i  # Flattened index

        if i == 0:  # Left boundary
            f_bc[k] -= V_left[j]
        if i == N - 1:  # Right boundary
            f_bc[k] -= V_right[j]
        if j == 0:  # Bottom boundary
            f_bc[k] -= V_bottom[i]
        if j == M - 1:  # Top boundary
            f_bc[k] -= V_top[i]

# Solve system
u_vec = np.linalg.solve(L, f_bc)
u_numerical = u_vec.reshape((N, M))

# Plot
```

```
plt.contourf(Y, X, u_numerical, levels=25, cmap="viridis")
plt.colorbar()
plt.title("Numerical Solution")
plt.show()
```

6 Time Dependent PDEs

Consider the continuity equation,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0$$

In one dimensional,

$$\frac{\partial \rho(x,t)}{\partial t} + \frac{\partial \rho u(x,t)}{\partial \rho} = 0$$

Forward Euler (Explicit) Scheme

Taking $u(x,t) = u$ constant we get

$$\rho_j^{n+1} = \rho_j^n - u\frac{\Delta t}{2\Delta x}\left(\rho_{j+1}^n - \rho_{j-1}^n\right)$$

This method is unstable as the eigenvalue for the P Matrix is always more than 1, thus norm increases to infinity.

```
def forward_euler(rho0, u, dx, dt, steps):
    rho = rho0.copy()
    N = len(rho)
    for _ in range(steps):
        rho_new = rho.copy()
        rho_new[1:-1] = rho[1:-1] - u*dt/(2*dx)*(rho[2:] - rho[:-2])
        rho = rho_new
    return rho
```

Implicit Euler Method

$$\rho_j^{n+1} = \rho_j^n - u\frac{\Delta t}{2\Delta x}\left(\rho_{j+1}^{n+1} - \rho_{j-1}^{n+1}\right)$$

Thus we can write it as,

$$\boldsymbol{\rho}^n = T\boldsymbol{\rho}^{n+1} + B\boldsymbol{\rho}^{n+1}$$

$$T = \begin{bmatrix} 1 & a & 0 & \dots & 0 \\ -a & 1 & a & \dots & 0 \\ \vdots & \vdots & & & \\ 0 & \cdots & 0 & -a & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & \dots & -a \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & & & \\ a & \cdots & 0 & 0 & 0 \end{bmatrix}$$

where $a = \frac{u\Delta t}{2\Delta x}$

```
def implicit_euler_periodic(rho0, u, dx, dt, steps):
    rho = rho0.copy()
    N = rho.size
    a = u * dt / (2.0 * dx)   # as defined in the notes
    # Build A = T + B (dense)
    A = np.eye(N, dtype=float)

    if N > 1:
        off = a * np.ones(N - 1, dtype=float)
        A += np.diag(off, 1)       # +a on superdiagonal
        A -= np.diag(off, -1)      # -a on subdiagonal
        # periodic wrap contributions (the B matrix)
        A[0, -1] = -a              # top-right corner
        A[-1, 0] =  a              # bottom-left corner
    for _ in range(steps):
        rho = np.linalg.solve(A, rho)

    return rho
```

Leapfrog Method

By taking centre difference in not just space but also time,

$$\rho_j^{n+1} = \rho_j^{n-1} - u\frac{\Delta t}{\Delta x}\left(\rho_{j+1}^{n+1} - \rho_{j-1}^{n+1}\right)$$

```
def leapfrog(rho_prev, rho_curr, u, dx, dt, steps):
    rho_nm1 = rho_prev.copy()
    rho_n = rho_curr.copy()
    for _ in range(steps):
        rho_np1 = rho_nm1 - u * dt / dx * (rho_n[2:] - rho_n[:-2]) / 2
        rho_nm1, rho_n = rho_n, np.concatenate(([0], rho_np1, [0]))
    return rho_n
```

Lax-Friedrich Method

Current step and position is taken as an average of neighbouring positions,

$$\boldsymbol{\rho}^{n+1} = \frac{1}{2}\begin{bmatrix} 0 & 1-u\frac{\Delta t}{\Delta x} & 0 & \cdots & 0 \\ 1+u\frac{\Delta t}{\Delta x} & 0 & 1-u\frac{\Delta t}{\Delta x} & \cdots & 0 \\ \vdots & & & & \\ 0 & \cdots & 0 & 1+u\frac{\Delta t}{\Delta x} & 0 \end{bmatrix}\boldsymbol{\rho}^n$$

```
def lax_friedrichs_T(N, u, dx, dt):
    c = u * dt / (2.0 * dx)
    upper = 0.5 - c   # weight for rho_{j+1}
    lower = 0.5 + c   # weight for rho_{j-1}
    T = np.zeros((N, N), dtype=float)
    for j in range(N):
        if j - 1 >= 0:
            T[j, j - 1] = lower
        if j + 1 < N:
            T[j, j + 1] = upper

    return T

def evolve_matrix(rho0, T, steps, periodic=False, u=None, dx=None,
dt=None):
    rho = rho0.copy()
    N = rho.size

    if periodic:
        c = u * dt / (2.0 * dx)
        T[0, -1] = 0.5 - c    # contribution from rho_{N-1} into rho_0
        T[-1, 0] = 0.5 + c    # contribution from rho_0 into rho_{N-1}
        P = T + B
    else:
        P = T

    for _ in range(steps):
        rho = P @ rho

N = 200
x = np.linspace(0, 1, N, endpoint=False)
dx = x[1] - x[0]
u = 0.1
dt = 0.8 * dx / u
steps = 100

rho0 = np.exp(-((x - 0.5)**2) / 0.02)
T = lax_friedrichs(N, u, dx, dt)
rho_p  = evolve_matrix(rho0, T, steps, periodic=True, u=u, dx=dx, dt=dt)
```

Lax-Wendroff

```
def lax_wendroff_matrix(N, u, dx, dt):
    c = u * dt / (2 * dx)
    P = np.zeros((N, N))
    for j in range(N):
        P[j, (j - 1) % N] = c + c**2
        P[j, j] = 1 - 2 * c**2
        P[j, (j + 1) % N] = -c + c**2
    return P
```

6.1 Non-Linear Equations

Richtmyer 2-step Lax Wendroff

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

N = 1000
a, b = 0, 1
x_full = np.linspace(a, b, N+2, dtype=np.float64)
x = x_full[1:-1]
dx = x[1] - x[0]

u = 0.01 * np.sin(2*np.pi*x)
t_max = 20.0
time = 0.0

fig, axes = plt.subplots(4, 5, sharex=True, sharey=True,
figsize=(15,12))

snapshots = 20
snapshot_times = np.linspace(0, t_max, snapshots)
s = 0
for snap_t in snapshot_times:
    while time < snap_t:
        u_max = np.max(np.abs(u)) + 1e-9
        dt = 0.6 * dx / u_max
        dt_dx = dt / dx

        # p
        f = 0.5 * u**2
        u_half = 0.5*(u[1:] + u[:-1]) - 0.5*dt_dx*(f[1:] - f[:-1])

        f_half = 0.5 * u_half**2

        u[1:-1] = u[1:-1] - dt_dx * (f_half[1:] - f_half[:-1])

        # periodic wrap for ghost cells
        u[0]  = u[-2]
        u[-1] = u[1]

        time += dt

    r = s // 5
    c = s % 5
    ax = axes[r, c]
    ax.plot(x, u, color=f'#{8*s:06X}')
    ax.set_title(f"t ≈ {time:.2f}s")
    ax.grid(True)
    s += 1

plt.tight_layout()
plt.show()
```

Schrodinger Equation

```
def crank_nicolson(psi0, V, dx, dt, hbar=1, m=1, steps=1000):
    N = len(psi0)
    D = np.diag(-2*np.ones(N)) + np.diag(np.ones(N-1), 1) +
np.diag(np.ones(N-1), -1)
    P = np.diag(V)
    H = -D + P

    A = np.eye(N) + 1j * dt / (2) * H
    B = np.eye(N) - 1j * dt / (2) * H

    psi = psi0.copy()
    for _ in range(steps):
        psi = np.linalg.solve(A, B @ psi)
        psi /= np.linalg.norm(psi)  # normalization
    return psi
```

7 Monte Carlo Methods

7.1 Integration

Area of Circle

```
N = 10000
rng = np.random.default_rng(seed=67)
x = rng.random(N)
y = rng.random(N)

inside = (x*x + y*y) < 1
cum_inside = np.cumsum(inside)
trials = np.arange(1, N+1)
value_circle = 4 * (cum_inside / trials)
print(value_circle[-1])
```

Volume of N-dim Sphere

```
N = 1000000
dimension = 10

def N_dim_Sphere(N, dimension):
    rng = np.random.default_rng(seed=70)
    points = rng.random((N, dimension))
    points = points*points
    r = np.sum(points, axis=1)

    inside = np.where(r <= 1, 1, 0)
    cum_inside_me = np.cumsum(inside)
    trials_ndim = np.arange(1, N+1)
    value_circle = (2**dimension) * (cum_inside_me/ trials_ndim)
    print(f"Monte Carlo Estimate: {value_circle[-1]:.5f}")
    true_value = (np.pi**(dimension/2)) / math.gamma(dimension/2 + 1)  #
π^{d/2}/Γ(d/2+1)
    print(f"True Value: {true_value:.5f}")
```

Area under Function

```
def Monte_Carlo_Integration(f, x_lim: np.ndarray, y_lim: np.ndarray, N:
int = 10000, seed: int = 42):
    np.random.seed(seed)
    points = np.random.random((N, 2))

    # Making Random Points go into range of x_lim and y_lim
    points[:, 0] *= x_lim[1] - x_lim[0]
    points[:, 1] *= y_lim[1] - y_lim[0]
    points[:, 0] += x_lim[0]
    points[:, 1] += y_lim[0]

    f_points = f(points[:, 0]) # Array of evaluated values on given
random points

    positive_region = (f_points >= 0) & (points[:, 1] >= 0) & (points[:,
1] <= f_points)
    negative_region = (f_points < 0) & (points[:, 1] < 0) & (points[:,
1] >= f_points)

    point_contributions = positive_region.astype(int) -
negative_region.astype(int)

    cumulative_contributions = np.cumsum(point_contributions)

    # Running estimate at each iteration
    box_area = (x_lim[1] - x_lim[0]) * (y_lim[1] - y_lim[0])
    iterations = np.arange(1, N + 1)
    convergence = (cumulative_contributions / iterations) * box_area
    return convergence[-1]

limits_x = np.array([0, 3*np.pi/2])
limits_y = np.array([np.exp(0), np.exp(3*np.pi/2)]) # Min and Max Value
of the function in entire range
N = 10000
seed = 67
print(f"Monte Carlo: {Monte_Carlo_Integration(function, limits_x,
limits_y, N, seed):.5f}")
```

7.2 Casting Distributions

To generate a function a distribution $f(y)$, then we need to create a mapping from $x \mapsto y$ such that

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{f(y)}$$

Then for a probability distribution $\rho(x)$, we can find the integral by,

$$\int f(x)\,\mathrm{d}x = \left\langle \frac{f(x)}{\rho(x)} \right\rangle$$

Closed Form Integrals

Given $f(y) = \sin(y)$, we can say $\sin(y)\,\mathrm{d}y = \mathrm{d}x \Rightarrow y = \arccos(1-x)$.

```
N = 1000000
x = np.random.rand(N)              # Uniform [0, 1)
y = np.arccos(1 - x)               # Inverse transform sampling
plt.hist(y, bins=100, density=True, alpha=0.6, label='Sampled')
plt.plot(np.linspace(0, np.pi/2, 100), np.sin(np.linspace(0, np.pi/2,
100))/1, 'r-', lw=2, label='sin(y)')
plt.legend(); plt.show()
```

Box Muller Transform

To generate the standard normal distribution,

$$f(y) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{1}{2}\frac{(y-\mu)^2}{\sigma^2}\right] \quad \text{where} \quad \mu = 0, \sigma = 1$$

There exists no closed form solution to the required integral to obtain the map $x \mapsto y$ so we convert to polar coordinates,

$$f(x)f(y)\,\mathrm{d}x\,\mathrm{d}y \mapsto \frac{1}{2\pi}\exp\left(-\frac{r^2}{2}\right)r\,\mathrm{d}r\,\mathrm{d}\theta$$

Now to generate the probability distribution above,

$$g(r) = \exp\left(-\frac{r^2}{2}\right)r \Rightarrow \mathrm{d}x = g(r)\,\mathrm{d}r \Rightarrow r = -\sqrt{2\ln(x)}$$

$$\theta = x \times (2\pi)$$

Thus, we get 2 normal distributions

$$y_1 = r\cos(\theta) \quad y_2 = r\sin(\theta)$$

```
N = 100000

x_1 = np.random.rand(N)
x_2 = np.random.rand(N)

r = np.sqrt(-2*np.log(x_1))
theta = 2*np.pi*x_2

y_1 = r * np.cos(theta)
y_2 = r * np.sin(theta)
normal = lambda x: np.exp(-x**2/2)/(np.sqrt(2*np.pi))
compare = np.linspace(-3, 3, N)

counts1, bins1 = np.histogram(y_1, bins=50)
deltax1 = bins1[1]-bins1[0]
pdf1 = (counts1/N)/deltax1
bin_mid1 = (bins1[:-1]+bins1[1:])/2

counts2, bins2 = np.histogram(y_2, bins=50)
deltax2 = bins2[1]-bins2[0]
pdf2 = (counts2/N)/deltax2
bin_mid2 = (bins2[:-1]+bins2[1:])/2

plt.hist(y_1, bins=50, alpha=0.5, label='cos', color='r')
plt.hist(y_2, bins=50, alpha=0.5, label='sin', color='g')
plt.legend()
plt.show()
plt.clf()

plt.plot(bin_mid1, pdf1, label='cos', color='r', alpha = 0.5)
plt.plot(compare, normal(compare), label='normal', color='b')
plt.plot(bin_mid2, pdf2, label='sin', color='g', alpha = 0.5)
plt.legend()
plt.show()
plt.clf()
```

7.3 Random Walks

Metropolis Hastings

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

# Initialize parameters
n_samples = 100000  # total number of samples to generate
samples = []  # list to store accepted samples
x_current = 0.5  # start point in the middle of [0,1]
sigma = 0.1  # proposal step size (standard deviation)

# Run the Metropolis algorithm
for i in range(n_samples):
    # Propose a new candidate from a normal distribution centered at
current x
    x_proposed = x_current + np.random.normal(0, sigma)

    # Reflect if the proposed value goes out of [0,1] (to stay within
bounds)
    if x_proposed < 0:
        x_proposed = -x_proposed
    elif x_proposed > 1:
        x_proposed = 2 - x_proposed

    # Compute acceptance ratio (unnormalized)
    p_current = - np.sin(x_current) * np.log(x_current)
    p_proposed = - np.sin(x_proposed) * np.log(x_proposed)

    # Avoid divide-by-zero errors
    if p_current == 0:
        acceptance_ratio = 1
    else:
        acceptance_ratio = p_proposed / p_current

    # Accept or reject based on Metropolis criterion
    if np.random.rand() < min(1, acceptance_ratio):
        x_current = x_proposed  # accept move

    # Store the current sample
    samples.append(x_current)

# Convert to NumPy array for analysis
samples = np.array(samples)

# Plot the sampled distribution
plt.figure(figsize=(8, 5))
plt.hist(samples, bins=100, density=True, alpha=0.6, color='skyblue',
label='Sampled distribution')

# Plot the true (unnormalized) function for comparison
x = np.linspace(0.001, 0.999, 500)
norm_const = np.trapezoid(-np.sin(x)*np.log(x), x)
plt.plot(x, -np.sin(x)*np.log(x) / norm_const, 'r-', lw=2, label='True
(normalized) function')
plt.xlabel('x')
plt.ylabel('Probability density')
plt.legend()
plt.title('Sampling from f(x) = -sin(x) ln(x) using Metropolis
algorithm')
plt.show()
```

8 Neural Networks

Linear model

- We can see that A is

$$A = \begin{bmatrix} 0 & 0 \\ 2 & 1 \end{bmatrix} \tag{4}$$

- This matrix is uniquely determined by the first two conditions
- But then it does not satisfy the third condition
- This means a linear model will not work in this case
- In general, as the number of cases increases, you will get more conditions, while the matrix will have maximum rank of n
- So the system will be overdetermined and we cannot fit the linear model exactly
- One way to overcome this drawback is to go to a nonlinear model

Nonlinear model

- Take the output of a linear model and apply a nonlinear function to it

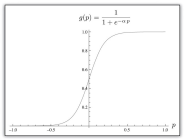
$$\boldsymbol{z} = g(\boldsymbol{y}) = \begin{bmatrix} g(y_1) \\ g(y_2) \\ . \\ . \\ g(y_m) \end{bmatrix} \tag{5}$$

- Defining $g(p)$ gives extra flexibility to fit all the required conditions, ex. the third condition above
- The input and outputs can be standardized for ease of operation
- Set $g(p)$ to lie between 0 and 1

Sigmoid function

- The sigmoid function is a continuous version of the step function

$$g(p) = \frac{1}{1 + e^{-\alpha p}} \tag{6}$$



- α sets the width of the sigmoid function, higher α makes the function sharper

Setting α

- The elements of the nonlinear output $\boldsymbol{z}$ are

$$z_i = g((A\boldsymbol{x})_i) = g[\sum_{j=1}^{n} A_{ij}x_j] \tag{7}$$

- Now the argument of function g should span the range of the width of the sigmoid function, so that the output g takes the full range of 0 to 1
- Assume that the elements of matrix A will be of the order unity
- Then the maximum element of $A\boldsymbol{x}$ will be approximately $n\max(x_i)$
- Then take α as

$$\alpha = \frac{10}{n\max(x_i)} \tag{8}$$

- Also, we can shift the input numbers $x_i \to x_i - med(x_i)$ such that the maximum and minimum are equal and opposite in signs
- This way we can handle input with any magnitude $\boldsymbol{x}$

Training

- Now the next step is to find the matrix A
- This process is called as training. In this we provide training examples to the model
- These examples are basically pairs of $\{\boldsymbol{x}^k, \boldsymbol{y}^k\}_{k=1}^{T}$ where $\boldsymbol{x}^k$ is the k-th input vector and $\boldsymbol{y}^k$ is its desired output vector
- There are T training examples, labelled by k, and our goal is to set A such that we get $\boldsymbol{y}^k$ for given $\boldsymbol{x}^k$
- For this we define an error function f

$$f^k = ||g(A\boldsymbol{x}^k) - \boldsymbol{y}^k||^2 \text{ for each case} j = 1, 2, ..., T \tag{9}$$

- The norm of a vector can be defined as the square root of the sum of the squares of its elements, i.e.,

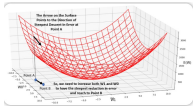
$$f^k = \sum_{i=1}^{m}[g(\sum_{s=1}^{n} A_{is}x_s^k) - y_i^k]^2 \tag{10}$$

Training

- This function f can be thought of as a function of the matrix A_{ij} because that is the only variable, rest all quantities come from the training set
- Hence, the objective then is the adjust A_{ij} such that we can minimize f for each training example
- f is a positive definite quantity, and this problem is of minimization of f
- A well-known method is known as the steepest descent (gradient descent) method

Gradient descent

- Consider a function of 2 variables, this can be represented as a surface
- This function contains has a minima



- If we have a function $f(\boldsymbol{x})$ of which we want to find minimum, then starting from some position $\boldsymbol{x}_0 = (x_0, y_0)$

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta\nabla f \tag{11}$$

- η is the step size, also called as "rate of learning"

Steepest descent

- Ex. find the minimum of $f(x, y) = x^2 + 4y^2 - 4x - 24y + 40$
- This actually comes from $(x-2)^2 + 4(y-3)^2$, so the minimum is at $(x, y) = (2, 3)$
- Learning rate can be faster with higher η
- However, too high η might lead to method not converging or instability
- Also called steepest descent because change is maximum in direction of gradient
- Convergence is determined when the steps become smaller than some threshold

Training model

- Going back to the case of finding the matrix A in our model
- The function to be minimized is

$$f(A_{ij}) = \sum_{i=1}^{m}\left[g(\sum_{j=1}^{n} A_{ij}x_j) - y_i\right]^2 \tag{12}$$

- given any training pair $(\boldsymbol{x}^k, \boldsymbol{y}^k)$ - suppressing the superscript k for simplicity
- Define $b_i = \sum_{j=1}^{n} A_{ij}x_j$ and $z_i = g(b_i)$, i goes from 1 to m
- Now we want to find the values of matrix elements A_{pq} which minimizes the error f
- For using gradient descent we need

$$\frac{\partial f}{\partial A_{pq}} = \sum_{i=1}^{m} 2(z_i - y_i)\frac{\partial z_i}{\partial A_{pq}} \tag{13}$$

Training model

- We have

$$\frac{\partial z_i}{\partial A_{pq}} = g'(b_i)\frac{\partial b_i}{\partial A_{pq}} \tag{14}$$

$$\frac{\partial b_i}{\partial A_{pq}} = \frac{\partial}{\partial A_{pq}}\sum_{j=1}^{m} A_{ij}x_i \tag{15}$$

$$= \sum_{j=1}^{m}\delta_{ip}\delta_{jq}x_i \tag{16}$$

$$= \delta_{ip}x_q \tag{17}$$

- Also can be shown that

$$g'(p) = \alpha g(p)(1 - g(p)) \tag{18}$$

Training model

- Putting this together we get

$$\frac{\partial f}{\partial A_{pq}} = \sum_{i=1}^{m} 2(z_i - y_i)\alpha g(b_i)(1 - g(b_i))\delta_{ip}x_q \tag{19}$$

- which gives

$$\frac{\partial f}{\partial A_{pq}} = 2(z_p - y_p)\alpha z_p(1 - z_p)x_q \tag{20}$$

- This makes the calculation of the matrix element easy

Algorithm

- Initialize random A of order unity
- Loop over each training pair
- Calculate z using the existing A matrix
- Loop over all elements of A_{pq} and update with

$$A_{pq} \rightarrow A_{pq} - \eta\left[2(z_p - y_p)\alpha z_p(1-z_p)x_q\right] \tag{21}$$

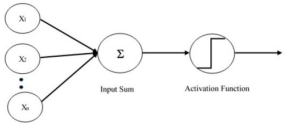
- Taking one gradient descent step over all the training examples is called an "epoch"
- Run the training for several epochs till the error is reduced below some threshold or the accuracy of the model reaches desired level
- The accuracy can be defined as the percentage of predictions it gets right
- After going through all training pairs and epochs, the model is trained and can be used for new data

Training model

- This training is not expected to produce exact results
- The outcome of the training will depend on the order in which the test cases are given
- Ideally after giving the training examples multiple times and in random order, the neural network A should converge
- However, this may not be the case always and it may not give good agreement for all the training examples
- After the model is trained on the training set, it can be tested on a testing set to check how well it performs
- If the performance if good, it can be used on new and untested data

Hidden layers

- If we are given an input vector with 10 elements, i.e., $n = 10$ and the output if just a single element, i.e., $m = 1$ then
- The matrix A will be 10×1 matrix with 10 parameters to adjust
- This can be represented diagramatically as



Hidden layers

- If there is a lot of data which is making the problem overconstrained, then we require more freedom in the neural network model
- We can add an intermediate layer of size k
- The input $\boldsymbol{x} \in R^n$ gets multiplied by matrix B which is $k \times n$ giving a $\tilde{\boldsymbol{z}} \in R^k$

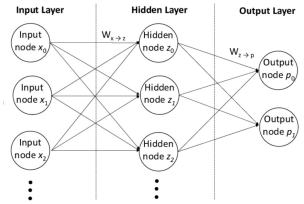
$$\tilde{\boldsymbol{z}} = g(B\boldsymbol{x}) - \frac{1}{2} \tag{22}$$

- Since the nonlinear function g is applied here, the $-1/2$ changes the mapping from $0-1$ to $-1/2$ to $+1/2$
- Now the matrix A of size $m \times k$ is applied on $\tilde{\boldsymbol{z}}$ to get the final output

$$\boldsymbol{z} = \tilde{g}(A\tilde{\boldsymbol{z}}) \tag{23}$$

Hidden layers

- The hidden layers can be diagrammatically represented as



Training

- Now both the B and A matrices have to found by training the model
- The error is again defined as

$$f(A_{ls}, B_{ij}) = \sum_{l=1}^{m}(z_l - y_l)^2 \tag{24}$$

- Using the steepest gradient technique as before, we can calculate the partial derivative of this w.r.t. the matrix elements A_{ls} and B_{ij}
- It can be shown that

$$\frac{\partial f}{\partial A_{pq}} = 2\tilde{\alpha}(z_p - y_p)z_p(1-z_p)\tilde{z}_q \tag{25}$$

Training

- Similarly, differentiating w.r.t. B_{pq} it can be shown that

$$\frac{\partial f}{\partial B_{pq}} = \sum_{l=1}^{m}\sigma_l A_{lp}\alpha(\frac{1}{2}+\tilde{z}_p)(\frac{1}{2}-\tilde{z}_p)x_q \tag{26}$$

- where

$$\sigma_l = 2\tilde{\alpha}(z_l - y_l)z_l(1-z_l) \tag{27}$$

- Now we can just modify the update algorithm

$$A_{pq} \rightarrow A_{pq} - \eta\frac{\partial f}{\partial A_{pq}} \tag{28}$$

$$B_{pq} \rightarrow B_{pq} - \eta\frac{\partial f}{\partial B_{pq}} \tag{29}$$

- More generally, the gradient w.r.t. the weights of the neural network nodes are estimated by a method of backward propagation

Different models

- A model with a large number of hidden layers is a deep network
- To generate mode data, noise can be added to the data
- The images can also be tranformed, like left-right inversion, or time inversion, etc.
- Convolutional neural network - image data is processed through some convolution in order to account for neighboring/related pixels
- Taking a large hidden layer (high k) versus many hidden layers - mostly based on experience and experimentation
- Steepest descent leads to the local minimim, might not always work or find the best possible network - therefore need to try many different starting points and train multiple times

Physics informed neural networks

- Neural networks can also be used to solve differential equations describing physical laws
- For example - suppose we experimentally measure the displacement of a particle as a function of time

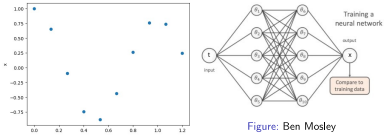


Figure: Ben Mosley

- Now we need to design a neural network that can fit a function $f(t)$ that will approximately give us the value of the displacement $x(t)$

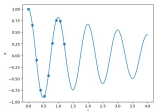
$$f(t) \approx u(t) \tag{37}$$

Physics informed neural networks

- With limited data, the neural network will not be able to extrapolate the displacement to later times
- Rather, if we know that the displacement obeys a differential equation then we can get the full solution

$$m\frac{d^2x}{dt^2} + \mu\frac{dx}{dt} + kx = 0 \tag{38}$$

- This is the equation of a damped harmonic oscillator (SHM if $\mu = 0$) and its solution looks as below



PINN

- Consider the damped harmonic oscillator differential equation

$$m\frac{d^2x}{dt^2} + \mu\frac{dx}{dt} + kx = 0 \tag{39}$$

- with the initial condition $x(t=0) = 1$ and $x'(t=0) = 0$, i.e. an oscillator starting from the peak displacement
- Setup a neural network NN that takes t as an input and gives x as output
- We have to train the network weight factors $\boldsymbol{\theta}$ such that

$$NN(t, \boldsymbol{\theta}) \approx x(t) \tag{40}$$

- We do not need any experimental data to train this network
- The neural network will start generating data itself and the differential equation and boundary conditions will help in training

Loss function

- The network is initialized with random weights
- In each epoch a few input collocation times (t_i, i=1,2,..,N_p) are chosen randomly over the range of interest (including the boundary point) - it is important to train the model over random time points in order to avoid bias towards early or later times
- The network makes predictions $NN(t_i, \boldsymbol{\theta})$
- The loss function consists of two parts, the boundary loss term

$$f_b = \lambda_1\left(NN(t=0,\theta) - 1\right)^2 + \lambda_2\left[\frac{dNN}{dt}(t=0,\theta) - 0\right]^2 \tag{41}$$

- and the differential equation loss is (also called physics-loss)

$$f_p = \frac{\lambda_3}{N_p}\sum_{i=1}^{N_p}\left[m\frac{d^2NN(t_i)}{dt^2} + \mu\frac{dNN(t_i)}{dt} + kNN(t_i)\right]^2 \tag{42}$$

Grad function

- The weights of the different loss terms $\lambda_1, \lambda_2, \lambda_3$ can be adjusted
- The derivatives dNN/dt and d^2NN/dt^2 need to be calculated

```
from torch.autograd import Variable
t_collocation = np.random.uniform(0,4,size=(500,1))
pt_t_collocation = Variable(torch.from_numpy\n(t_collocation).float(), requires_grad=True)
```

- After defining a network $NN(time)$ which takes an input $time$ tensor, it's derivative can be calculated

```
xn=NN(time)
dxndt=torch.autograd.grad(xn,time)
```

- Similarly, getting the second derivative will involve taking one more grad on the $dxndt$ variable

Wave equation

- In order to solve the wave equation for a quantity $u(x,t)$

$$\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = 0 \tag{43}$$

- For this we will define a $NN(x,t)$ and the physics loss will be

$$f_p = \frac{1}{N_p}\sum_i\left[\frac{\partial^2 NN(x_i,t_i)}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 NN(x_i,t_i)}{\partial t^2}\right] \tag{44}$$

- Where the summation is over all collocation points
- For periodic boundary condition the loss is

$$f_b = \sum_j[NN(x_j,t_j) - NN(x_j + L, t_j)]^2 \tag{45}$$

- where x_j is a boundary points and L is the period