Topic: Numerical Modelling

Concepts: Finite Difference approximation for derivative; Convergence; Solving an equation numerically using code;

(1) Finite Difference

To calculate the numerical derivative of the function sin(*x*) for *x* in the interval (0,2*pi) and compare it with analytical calculations. Follow through the steps below.

(i) At selected points: (manual calculations + code for plotting)

a. Through manual calculations, evaluate the numerical derivative of sin(x) at three points x=pi/4, x=pi/2, and x=pi using forward, backward, and central difference methods (at each of the points). Use the same step size (say, h) for all three methods.

b. Compare the errors resulting from the three methods. For error calculation, exact derivatives can be evaluated analytically at those points.

c. Repeat the calculations for different step sizes. Make a log-log plot of error vs step size and verify the rates of convergence for the three methods.

(ii) In the full interval of the domain (by discretization): (no manual calculations)

a. Discretize the domain (0,2*pi) with a step size that gives at least a two decimal accuracy (i.e. error <0.01) for the numerical derivative.

b. Through a computer code/program, evaluate the numerical derivative throughout the discretized domain using the three methods. Store the output in three different arrays.

c. Also, evaluate the analytical derivative at all the discrete points of the domain.

d. On a single plot, mark the analytical and numerical derivatives using solid line and markers respectively. (There should be four curves, one for analytical and three for the three numerical approaches. Clearly indicate through a legend in the plot. Use different markers for different methods.)

e. Find the error at all the discrete points for each of the three methods and plot the error curves together. Compare the errors and identify the maximum/minimum values and where they occur.

You may use the code snippets given below by modifying them appropriately for your requirement:

```
import math, numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def f(x): return math.sin(x)
def df_exact(x): return math.cos(x)
```

```
def d_forward(f, x, h):  return (f(x + h) - f(x)) / h

def abs_err(approx, exact): return abs(approx - exact)

def evaluate_at_points(points, h):
    rows = []
    for x in points:
        exact = df_exact(x)
        fwd = d_forward(f, x, h)
        rows.append({
            "x": x, "h": h, "exact": exact,
            "forward": fwd, "err_forward": abs_err(fwd, exact),
        })
    return rows

def convergence_study(points, h_values):
    hs, ef, eb, ec = [], [], [], []
    for h in h_values:
        batch = evaluate_at_points(points, h)
        ef.append(np.mean([r["err_forward"]  for r in batch]))
        hs.append(h)
    return {"h": np.array(hs), "err_forward": np.array(ef),
            "err_backward": np.array(eb), "err_central": np.array(ec)}

def slope_loglog(x, y):
    lx, ly = np.log10(x), np.log10(y)
    m, c = np.polyfit(lx, ly, 1)
    return m, c

res = convergence_study(X_POINTS, H_VALUES)
import pandas as pd
df = pd.DataFrame(res)
df

res = convergence_study(X_POINTS, H_VALUES)
m_f, _ = slope_loglog(res["h"], res["err_forward"])
print(f"Estimated orders: forward≈{m_f:.2f}, backward≈{m_b:.2f}, central≈{m_c:.2f}")

plt.figure()
plt.loglog(res["h"], res["err_forward"], marker="o", linestyle="--", label="Forward")
plt.gca().invert_xaxis()
plt.xlabel("h"); plt.ylabel("Mean abs error")
plt.title("Finite-difference error vs step size (log-log)")
plt.legend(); plt.tight_layout()
plt.savefig("fd_convergence.jpg", dpi=160)
plt.show()
```

```
# Write the code in the cell wherever TODO is mentioned and run the code to check the results and verify by the TAs.
def grid_over_domain(a, b, h):
    n = int(math.floor((b - a) / h)) + 1
    xs = a + np.arange(n)*h
    xs = xs[xs <= b]
```

```
    return xs

def evaluate_on_domain(h):
    a, b = 0.0, 2.0*math.pi
    xs = grid_over_domain(a, b, h)
    fwd = np.zeros_like(xs); bwd = np.zeros_like(xs); cen = np.zeros_like(xs)
    exact = np.cos(xs)
    for i, x in enumerate(xs):
        if i == 0:
            fwd[i] = '''TODO'''; bwd[i] = '''TODO'''; cen[i] = '''TODO'''
        elif i == len(xs)-1:
            fwd[i] = '''TODO'''; bwd[i] = '''TODO'''; cen[i] = '''TODO'''
        else:
            fwd[i] = '''TODO'''; bwd[i] = '''TODO'''; cen[i] = '''TODO'''
    ef = np.abs(fwd - exact); eb = np.abs(bwd - exact); ec = np.abs(cen - exact)
    return xs, exact, fwd, bwd, cen, ef, eb, ec

xs, exact, fwd, bwd, cen, ef, eb, ec = evaluate_on_domain(H_DOMAIN)

fig, axes = plt.subplots(2, 2, figsize=(10, 8), constrained_layout=True)

# (0,0): curves
ax = axes[0, 0]
ax.plot(xs, exact, label="Exact (cos x)")
ax.plot(xs, fwd, marker="o", linestyle="None", label="Forward")
ax.plot(xs, bwd, marker="s", linestyle="None", label="Backward")
ax.plot(xs, cen, marker="^", linestyle="None", label="Central")
ax.set_xlabel("x"); ax.set_ylabel("Derivative"); ax.set_title("Analytical vs
numerical")
ax.legend()

# (0,1): forward error
ax = axes[0, 1]
ax.plot(xs, ef, marker=".", linestyle="--", label="Forward")
ax.set_xlabel("x"); ax.set_ylabel("Absolute error"); ax.set_title("Error (Forward)")
ax.legend()

# (1,0): backward error
ax = axes[1, 0]
ax.plot(xs, eb, marker=".", linestyle="--", label="Backward")
ax.set_xlabel("x"); ax.set_ylabel("Absolute error"); ax.set_title("Error (Backward)")
ax.legend()

# (1,1): central error
ax = axes[1, 1]
ax.plot(xs, ec, marker=".", linestyle="--", label="Central")
ax.set_xlabel("x"); ax.set_ylabel("Absolute error"); ax.set_title("Error (Central)")
ax.legend()

fig.savefig("derivatives_and_errors_grid.jpg", dpi=160)
plt.show()

def summarize_domain_errors(xs, err):
    e = err.copy()
    mask = ~np.isnan(e)
```

```
    if not np.any(mask): return None
    e = e[mask]; x = xs[mask]
    i_max = int(np.argmax(e)); i_min = int(np.argmin(e))
    return {"max_error": float(e[i_max]), "x_at_max": float(x[i_max]),
            "min_error": float(e[i_min]), "x_at_min": float(x[i_min])}

sum_f = summarize_domain_errors(xs, ef)
sum_b = summarize_domain_errors(xs, eb)
sum_c = summarize_domain_errors(xs, ec)
sum_f, sum_b, sum_c
```

(2) Solving an equation

Using a computer program implementing the Newton-Raphson method, find the roots of the function sin(x) in the domain (0,4*pi). Follow through the steps below.

(i) Find the numerical solution by using an initial guess of x in between pi/2 and 3*pi/2. Note the number of iterations required to find the solution to an accuracy of 5 decimal places (i.e. error less than 1e-5).

(ii) Try different initial guess points in the given domain and see how the solution changes and also note the number of iterations in each case.

(iii) Try providing an initial guess close enough to the root and see if the required number of iterations reduce.

(iv) See how using different finite difference approximations affects the solution or the number of iterations taken (related to convergence).

You may use the code snippet given below by modifying it appropriately for your requirement:

```
# Write the code in the cell wherever TODO is mentioned and run the code to check the
results and verify by the TAs.
def newton_exact(x0, tol=1e-5, maxiter=100):
    x = float(x0)
    for k in range(maxiter):
        fx = '''TODO Define exact function'''; dfx = '''TODO Define exact function
derivative'''
        if abs(dfx) < 1e-14: return None, k, "derivative ~ 0"
        xnew = ## TODO write the function
        if abs(xnew - x) < tol: return xnew, k+1, "ok"
        x = ## TODO write the function
    return x, maxiter, "maxiter"

def newton_fd(x0, h, method="central", tol=1e-5, maxiter=100):
    x = float(x0)
    for k in range(maxiter):
        fx = math.sin(x)
        if method == "forward":
            dfx = ## TODO write the expression for the backward finite difference
method
        elif method == "backward":
            dfx = ## TODO write the expression for the backward finite difference
method
        else:
```

```
            dfx = ## TODO write the expression for the backward finite difference
method
        if abs(dfx) < 1e-14: return None, k, "derivative ~ 0"
        xnew = ## TODO write the function
        if abs(xnew - x) < tol: return xnew, k+1, "ok"
        x = ## TODO write the function
    return x, maxiter, "maxiter"

a_nr, b_nr = 0.0, 4.0*math.pi
records = []
for x0 in INITIAL_GUESSES:
    if not (a_nr <= x0 <= b_nr): continue
    root_e, it_e, status_e = newton_exact(x0)
    root_cf, it_cf, status_cf = newton_fd(x0, H_NEWTON, "forward")
    root_cb, it_cb, status_cb = newton_fd(x0, H_NEWTON, "backward")
    root_cc, it_cc, status_cc = newton_fd(x0, H_NEWTON, "central")
    records.append({
        "x0": x0,
        "exact_root": root_e, "exact_iters": it_e, "exact_status": status_e,
        "fd_forward_root": root_cf, "fd_forward_iters": it_cf, "fd_forward_status":
status_cf,
        "fd_backward_root": root_cb, "fd_backward_iters": it_cb, "fd_backward_status":
status_cb,
        "fd_central_root": root_cc, "fd_central_iters": it_cc, "fd_central_status":
status_cc,
    })

pd.DataFrame.from_records(records)
```