CEE 251L. Uncertainty, Design and Optimization
Homework 1
Code + reference documents (GitHub): https://github.com/PetersQuinn/HomeworkOne
Answers on next page

Problem 1 (5 points) — Engineering Ethics

Response.
University coursework is more than a checklist toward a degree; it is a sustained experience that develops both personal resilience and individual integrity. Over a semester, students repeatedly face situations that test discipline, honesty, adaptability, and perseverance. These tests are often subtle, but taken together they shape who we become not only as students, but as people.

A key way a university course builds resilience is through continuous academic pressure. Deadlines stack up, expectations rise, and setbacks are inevitable. Exams may not go as planned, projects can take longer than expected, and balancing coursework with other responsibilities can feel overwhelming. In these moments, students must confront discomfort rather than avoid it. Each time I have struggled with a difficult assignment or worked back from a disappointing grade, I have had to choose between giving up and adjusting my approach. Learning to diagnose what went wrong, seek help, and try again is the core of resilience. Repeating that cycle over time turns frustration into persistence and uncertainty into confidence.

Courses also strengthen integrity because honesty is often a personal decision rather than something enforced at every step. Completing work independently, citing sources correctly, and resisting the temptation to cut corners under stress are all ethical choices that appear throughout academic life. These decisions are rarely dramatic, but they are meaningful. Integrity is built through consistency in small actions. When I submit my own work even when no one is watching, I reinforce that my character matters more than a temporary advantage.

Resilience and integrity are tightly connected, especially when pressure is high. It becomes easier to justify shortcuts when time is short or failure feels unacceptable. That is precisely when resilience is most important: it allows a student to respond to stress without compromising values. Instead of cutting corners, students are pushed to develop healthier habits, such as planning earlier, asking questions sooner, and accepting imperfection while continuing to improve. In this way, challenges become opportunities to practice self-control and accountability.

Ultimately, a university course is not only about mastering content. It is a structured environment that mirrors real-world demands, where persistence and ethical judgment matter as much as technical skill. By navigating deadlines, setbacks, collaboration, and independent work, students build the resilience to endure adversity and the integrity to stay grounded in their values. These lessons extend far beyond the classroom and shape how we respond to challenges and how we define success in every area of life.

Reference document (GitHub): EngineeringEthicsQ1.pdf

Key ethical takeaway: Resilience is the ability to keep adapting under pressure, and integrity is the choice to stay honest under that pressure; university coursework builds both by forcing repeated, real decisions.

Problem 2 (5 points) — Martin Luther King Jr. Day Reflection

Given: On Martin Luther King day, read something by Martin Luther King Jr., listen to one of his speeches, or participate in an MLK Day event.

Find: A paragraph reflecting on the experience and personal thoughts.

Collaborators: none.

Reflection.
For Martin Luther King Jr. Day, I took time to listen to one of Dr. King's speeches and reflect on the meaning behind his words. What stood out to me most was how he balanced hope with urgency, reminding listeners that progress does not happen passively but requires courage, sacrifice, and consistent moral action. His message felt especially relevant today, since many of the challenges he spoke about still exist in different forms. The experience made me realize that honoring his legacy is not only about remembering history, but also about examining my own choices and asking whether I am contributing to fairness and respect in the spaces I am part of.

Problem 3 (4 points) — Python Setup Readings

Given: (a) Read *Using the Terminal.* (b) Read *Python language skills and debugging.* (c) Install git, VS Code, and Python from the instructions in section 2.

Find: Confirmation of completion and brief notes on key takeaways or setup details.

Collaborators: none.

All required readings completed and software installed; verified via terminal commands.

Problem 4 (6 points) — Install `multivarious` and Verify Import in VS Code

Given: Clone `multivarious` and install via pip editable mode; verify VS Code can import it using `verify_path_import.py`.

Find: Terminal commands used and evidence that import/path verification succeeded.

Collaborators: none. **Verification in VS Code.**
Opened `multivarious/examples`, ran `verify_path_import.py`, and confirmed that: (1) `PYTHONPATH` was set to the local clone path, and (2) `multivarious` imported successfully.

$$\boxed{\texttt{multivarious}\text{ installed in editable mode and verified import/path in VS Code.}}$$

Problem 5 (10 points) — Make Computing Mistakes in Python and Fix Them

Given: Type the provided commands, inspect results, identify mistakes, and correct them. Also invent two more vector mistakes and two more matrix mistakes (y, z, E, F) and fix them.

Find: Corrected code and brief explanations of what went wrong and why the fix works.

Collaborators: none.

Notes (thinking in words for partial credit).
This exercise highlights common pitfalls when moving between Python/NumPy and other numerical environments: (1) Python lists are not mathematical arrays, (2) array shape matters (1D vs 2D) especially for transpose, (3) `*` is element-wise multiplication while `@` is matrix multiplication, and (4) some operators look familiar but mean different things (e.g., `^` is XOR, not exponentiation).

Code link (GitHub): errors.py

Two additional vector mistakes and two additional matrix mistakes (with fixes).

```
# errors.py
# Quinn Peters
# CEE 251L HW1 Problem 5

import numpy as np

# -------------------------
# Two MORE vector mistakes
# -------------------------

# y (vector mistake): using ^ expecting exponentiation (it's XOR
    for ints)
y = np.array([1, 2, 3]) ^ 2          # WRONG: bitwise XOR
y_fix = np.array([1, 2, 3]) ** 2     # RIGHT: exponentiation

# z (vector mistake): expecting .append to modify a NumPy array
   "in place"
z = np.array([5, 8, 13])
try:
    z.append(21)                       # WRONG: NumPy arrays have
        no append method
except AttributeError:
    pass
z_fix = np.append(z, 21)               # RIGHT: returns a new
   array


# -------------------------
```

```python
# Two MORE matrix mistakes
# -------------------------

# E (matrix mistake): using * expecting matrix multiplication
E = np.array([[1, 2],
              [3, 4]])
F = np.array([[10],
              [20]])

E_wrong = E * F                       # WRONG: element-wise broadcasting (not matrix multiply)
E_fix = E @ F                         # RIGHT: matrix multiplication (2x2)@(2x1) -> (2x1)

# F (matrix mistake): trying to invert a non-square matrix
G = np.array([[1, 2, 3],
              [4, 5, 6]])             # 2x3 not square
# G_wrong = np.linalg.inv(G)          # WRONG: inv requires a square matrix

# Fix: use pseudoinverse for non-square matrices
G_fix = np.linalg.pinv(G)             # RIGHT: pseudoinverse exists for non-square matrices
```

Additional mistakes demonstrated: `^` vs `**`, NumPy array immutability/`append`, `*` vs `@`, and the requirement that `inv` applies only to square matrices (use `pinv` otherwise).

Problem 6 (10 points) — The Golden Ratio and Fibonacci Module (`fibo.py`)

Given: Fibonacci sequence with $F_0 = 1, F_1 = 1$ and $F_j = F_{j-1} + F_{j-2}$. The ratio of consecutive Fibonacci numbers, F_j/F_{j+1}, converges to the golden ratio value φ as j gets large. The golden ratio solves $\frac{1}{\varphi} = 1 + \varphi$.

Find: Implement `fibo.py` with functions `seqnce(N)` and `plot(N)`, generate plots for $N = 20$, and briefly discuss irrationality and why φ is often called the "most irrational" number.

Collaborators: none.

Code link (GitHub): fibo.py

Implementation (`fibo.py`).

```
# fibo.py
# Purpose: Generate Fibonacci sequences and visualize
    convergence to the golden ratio value.
# Name: Quinn Peters

import numpy as np
import matplotlib.pyplot as plt


def seqnce(N):
    """
    Returns a Fibonacci sequence of length N+1 (F_0 through F_N)
        for N > 1.

    Returns
    -------
    j : np.ndarray
        Indices from 0 to N (inclusive).
    F : np.ndarray
        Fibonacci numbers F_0 ... F_N with F_0 = 1, F_1 = 1.
    """
    if N <= 1:
        raise ValueError("N must be greater than 1.")

    # (a) Compute Fibonacci sequence F_0 ... F_N using only F_0
       =1 and F_1=1
    F = np.zeros(N + 1, dtype=np.int64)
    F[0] = 1
    F[1] = 1
    for k in range(2, N + 1):
        F[k] = F[k - 1] + F[k - 2]

    # (b) Indices j = [0 ... N] using np.linspace
    j = np.linspace(0, N, N + 1).astype(int)
```

```python

    return j, F


def plot(N):
    """
    Plots:
      1) Fibonacci sequence F_j vs j
      2) Ratio sequence r_j = F_j / F_{j+1} vs j, showing convergence to phi

    where phi solves 1/phi = 1 + phi, i.e. phi = (sqrt(5) - 1)/2   0.618.
    """
    # (a) Get j and F from seqnce(N)
    j, F = seqnce(N)

    # (b) Ratio sequence (golden ratio values) from F
    r = F[:-1] / F[1:]
    j_r = j[:-1]

    # Golden ratio value as defined in the prompt: 1/phi = 1 + phi
    phi = (np.sqrt(5) - 1) / 2

    # (c) Plot both sequences with labeled axes
    fig, axes = plt.subplots(2, 1, figsize=(8, 7), sharex=True)

    axes[0].plot(j, F, marker="o")
    axes[0].set_ylabel("Fibonacci number $F_j$")
    axes[0].set_title("Fibonacci sequence")

    axes[1].plot(j_r, r, marker="o", label=r"$F_j/F_{j+1}$")
    axes[1].axhline(phi, linestyle="--", label=rf"$\varphi = (\sqrt{{5}}-1)/2 \approx {phi:.6f}$")
    axes[1].set_xlabel("Index $j$")
    axes[1].set_ylabel("Ratio")
    axes[1].set_title("Convergence of consecutive ratios")
    axes[1].legend()

    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    plot(20)
```

Plot output for $N = 20$.

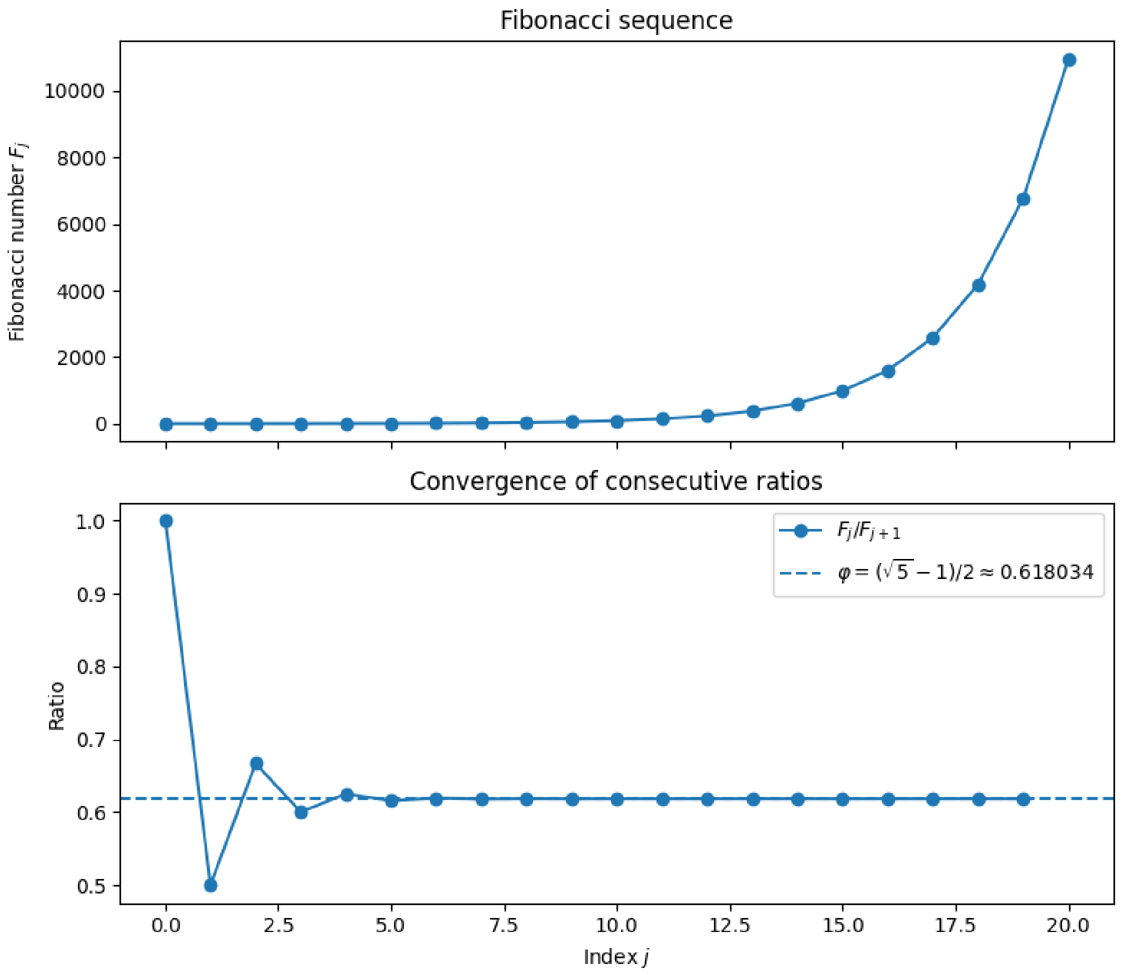

Figure 1: Fibonacci sequence (F_0 through F_{20}) and the ratio sequence F_j/F_{j+1} showing convergence toward φ.

Irrationality discussion (short).
One way to compare how “irrational” two irrational numbers are is to ask how well they can be approximated by rational numbers n/m. Numbers whose continued fraction expansions have small terms tend to have especially good rational approximations. The golden ratio has a continued fraction with all 1’s, which makes it unusually difficult to approximate closely by rationals, so it is commonly described as the “most irrational” number in this approximation sense.

`fibo.py` implements `seqnce(N)` and `plot(N)`; using $N = 20$ produces the included plot, and the ratios F_j/F_{j+1} converge toward $\varphi \approx 0.618$.

Problem 7 (10 points) — Sums of Sinusoids Without a For-Loop or Sum

Given: $y(x;n) = \frac{4}{\pi}\sum_{k=1}^{n}\frac{1}{2k-1}\sin((2k-1)x)$. Using a single for-loop compute three vectors of y for $n = 5, 10, 15$. Within the for-loop, create a row vector $k = [1 : n]$ and then calculate y in one line of Python by being clever about transpose (.T), element-wise operations, and vector multiplication (@). Use 500 values of x in the range $-\pi \leq x \leq \pi$ and plot $y(x;5)$, $y(x;10)$, and $y(x;15)$ on the same axes.

Find: Vectorized implementation (no inner for-loop and no explicit summation over k), a labeled plot, and an explanation of why this works.

Collaborators: none.

Code link (GitHub): sinWork.py

Implementation (`sinWork.py`).

```
# sinWork.py
# Purpose: Compute and plot sums of sinusoids (Fourier series
    partial sums) for n=5,10,15.
# Name: Quinn Peters

import numpy as np
import matplotlib.pyplot as plt


def fourier_square_partial(x: np.ndarray, n: int) -> np.ndarray:
    """
    Compute y(x; n) = (4/pi) * sum_{k=1..n} [ 1/(2k-1) * sin((2k
       -1)*x) ]
    without using an explicit sum() or inner for-loop.
    """
    k = np.arange(1, n + 1)                         # k = [1, 2, ..., n]
    m = 2 * k - 1                                   # odd harmonics
    a = 1 / m                                       # coefficients
    y = (4 / np.pi) * (a.reshape(1, -1) @ np.sin(m.reshape(-1,
       1) * x.reshape(1, -1))).ravel()
    return y


def main():
    x = np.linspace(-np.pi, np.pi, 500)

    plt.figure(figsize=(9, 5))

    for n in (5, 10, 15):
        y = fourier_square_partial(x, n)
        plt.plot(x, y, label=f"n = {n}")

    plt.xlabel("x")
    plt.ylabel("y(x; n)")
    plt.title("Fourier series partial sums: sum of sinusoids")
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    main()
```

Plot output (500 points on $[-\pi, \pi]$).

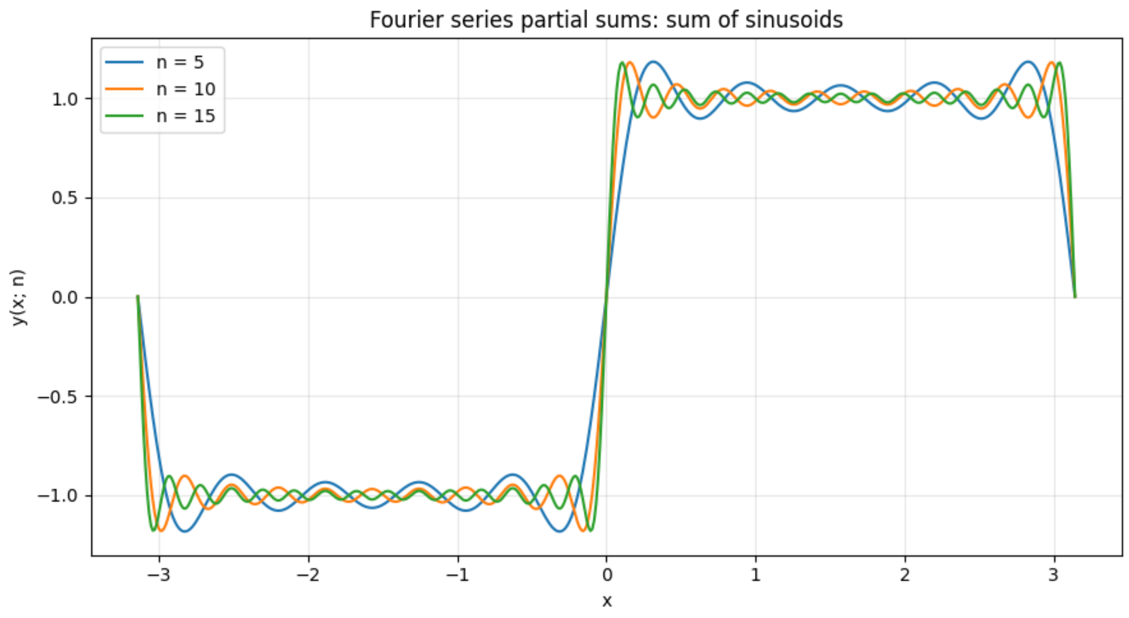

Figure 2: Fourier series partial sums $y(x; n)$ for $n = 5, 10, 15$ computed using vectorized array operations and matrix multiplication.

Why no inner for-loop or explicit sum over k is needed.
The computation forms all n harmonics at once as an $n \times 500$ matrix $\sin(mx)$ using broadcasting. The coefficients are stored as a $1 \times n$ row vector, and the weighted sum across harmonics is computed in a single matrix multiplication, $(1 \times n)@(n \times 500)$, which produces the 1×500 vector of values for $y(x; n)$. This replaces looping over k with vectorized linear algebra on NumPy arrays.

$$\boxed{\text{Computed and plotted } y(x;5),\ y(x;10), \text{ and } y(x;15) \text{ on } [-\pi,\pi] \text{ using a single loop over } n \text{ and vectorized matrix operations over harmonics.}}$$

Problem 8 (5 points) — 3-D Surface Plot and Contour Plot

Given: $f(x, y) = 2x^2 - 3xy - 4y^2$ for $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$.
Find: Make a surface plot and a contour plot with labeled axes.
Collaborators: none.

Code link (GitHub): 3D_plot.py

Implementation (3D_plot.py).

```
# 3D_plot.py
# Purpose: Create a 3D surface plot and a contour plot of the
   saddle function
#f(x, y) = 2x^2 - 3xy - 4y^2 on -10<=x<=10, -10<=y<=10
# Name: Quinn Peters

import numpy as np
import matplotlib.pyplot as plt


def f(x, y):
    return 2 * x**2 - 3 * x * y - 4 * y**2


def main():
    x = np.linspace(-10, 10, 200)
    y = np.linspace(-10, 10, 200)
    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)

    # -------- 3D surface plot --------
    fig = plt.figure(figsize=(9, 6))
    ax = fig.add_subplot(111, projection="3d")
    ax.plot_surface(X, Y, Z)

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("f(x, y)")
    ax.set_title(r"Surface plot of $f(x,y)=2x^2-3xy-4y^2$")

    plt.tight_layout()
    plt.show()

    # -------- Contour plot --------
    plt.figure(figsize=(8, 6))
    cs = plt.contour(X, Y, Z, levels=25)
    plt.clabel(cs, inline=True, fontsize=8)

    plt.xlabel("x")
    plt.ylabel("y")
    plt.title(r"Contour plot of $f(x,y)=2x^2-3xy-4y^2$")
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    main()
```

Plot outputs.

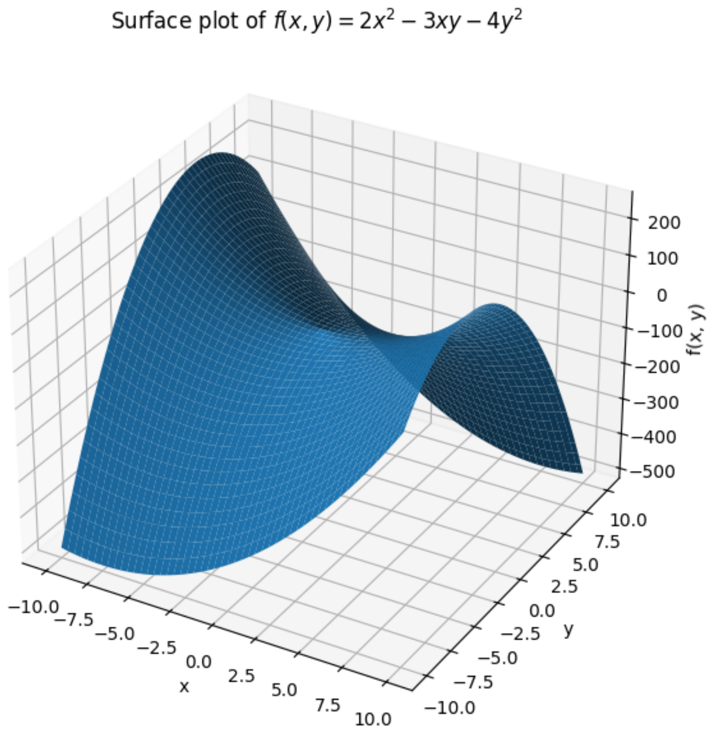

Figure 3: 3D surface plot of $f(x, y) = 2x^2 - 3xy - 4y^2$ on $-10 \le x \le 10$, $-10 \le y \le 10$.

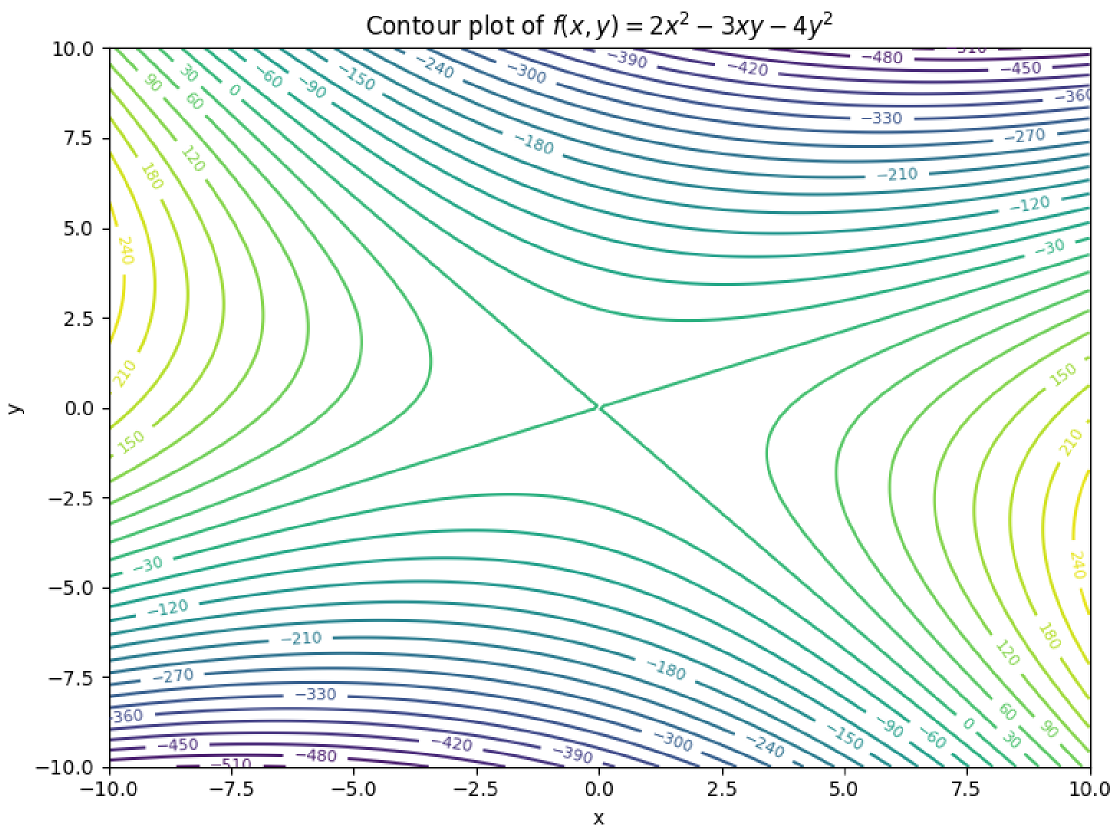

Figure 4: Contour plot of $f(x, y) = 2x^2 - 3xy - 4y^2$ on $-10 \le x \le 10$, $-10 \le y \le 10$.

$\boxed{\text{Surface and contour plots generated for } f(x, y) = 2x^2 - 3xy - 4y^2 \text{ on } [-10, 10]^2.}$

Problem 9 (5 points) — Maximize Beam End Slope for a Simply Supported Beam

Given: Simply supported beam with point load F located a distance a from the left end, span L. From beam deflection tables, use the expression for the right-end slope/rotation θ_2. Compute the dimensionless end-rotation $\frac{EI\theta_2}{FL^2}$ for 100 values of $a/L \in [0, 1]$ in one line (without a for-loop), use `argmax` to find the maximizing a/L, and plot the curve with a marker at the maximum.

Find: The dimensionless expression, the maximizing location a/L, and a labeled plot.

Collaborators: none.

Code link (GitHub): beam_def.py

Beam-table equation and dimensionless form.
For a simply supported beam with a point load F at distance a from the left end, let $b = L-a$. From the beam tables, the right-end rotation is

$$\theta_2 = \frac{F\,a\,b\,(L+a)}{6\,L\,E\,I}.$$

Then the required dimensionless quantity is

$$\frac{EI\theta_2}{FL^2} = \frac{a\,b\,(L+a)}{6L^3}.$$

Let $r = a/L$, so $b = L - a = L(1-r)$ and $L + a = L(1+r)$. Substituting gives

$$\frac{EI\theta_2}{FL^2} = \frac{r(1-r)(1+r)}{6}.$$

Implementation (`beam_def.py`).

```python
# beam_def.py
# Purpose: Find where a point load on a simply supported beam
    maximizes the right-end rotation.
# Name: Quinn Peters

import numpy as np
import matplotlib.pyplot as plt


def main():
    # 100 values of r = a/L in [0, 1]
    r = np.linspace(0, 1, 100)

    # Dimensionless end rotation:
    # theta2 = F*a*b*(L+a)/(6*L*E*I), b = L-a
    # (E*I*theta2)/(F*L^2) = (a*b*(L+a)) / (6*L^3) = (r*(1-r)
        *(1+r))/6
    dim = (r * (1 - r) * (1 + r)) / 6  # one line, element-wise ops only

    i_max = np.argmax(dim)
    r_max = r[i_max]
    dim_max = dim[i_max]

    # Plot
    plt.figure(figsize=(8, 5))
    plt.plot(r, dim, label=r"$(EI\theta_2)/(F L^2)$")
    plt.plot(r_max, dim_max, "ro", label="max")

    plt.xlabel(r"$a/L$")
    plt.ylabel(r"$(EI\theta_2)/(F L^2)$")
    plt.title("Dimensionless right-end rotation vs load location")
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    print(f"Max at a/L = {r_max:.6f}")
    print(f"Max (EI*theta2)/(F*L^2) = {dim_max:.6f}")


if __name__ == "__main__":
    main()
```

Plot output.

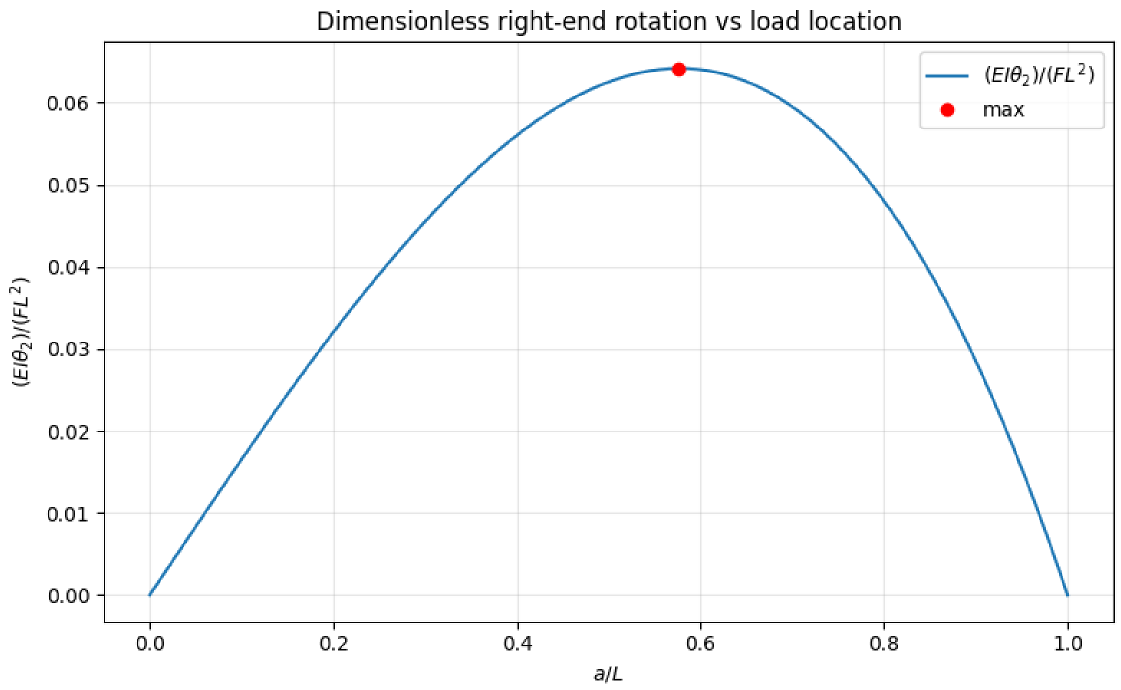

Figure 5: Dimensionless right-end rotation $(EI\theta_2)/(FL^2)$ versus load location a/L, with the maximizing point marked.

Maximum: $a/L = 0.575758$ and $\max\left((EI\theta_2)/(FL^2)\right) = 0.064149.$

Problem 10 (25 points) — Guess-and-Check Logic to Minimize an Objective Function

Given: Objective function

$$f(v_1, v_2) = 2 + \frac{v_1}{40} + \frac{v_2}{30} + \cos\left(\frac{v_1 v_2}{20}\right),$$

with domain $-10 \le v_1 \le 10$ and $-10 \le v_2 \le 10$. Write a Python file containing a function `fv(v)` that computes $f([v_1, v_2])$ and prints the value in the specified format. Starting from an initial guess, use a smart guess-and-check strategy (based on recent guesses) to iteratively reduce $f(v)$.

Find: The function implementation, a guess-and-check update rule, sample evaluations, and an explanation of why the logic works (or where it can fail).

Collaborators: none.

Code link (GitHub): finalProblem.py

Implementation (`finalProblem.py`).

```python
# finalProblem.py
# Purpose: Objective function + simple guess-and-check
    minimization
# Name: Quinn Peters

import numpy as np


def fv(v):
    """
    Compute the objective function:
        f(v1, v2) = 2 + v1/40 + v2/30 + cos(v1*v2/20)
    where v is a list or array like [v1, v2].

    Returns
    -------
    f : float
        Objective value
    """
    v1 = float(v[0])
    v2 = float(v[1])
    f = 2.0 + v1 / 40.0 + v2 / 30.0 + np.cos((v1 * v2) / 20.0)

    # Required print format (matches the assignment style)
    print(f" f([ {v1:7.4f} , {v2:7.4f} ]) = {f:7.4f} ")
    return f


def clip_to_domain(v, lo=-10.0, hi=10.0):
    return np.clip(v, lo, hi)


def guess_and_check(v0, step0=2.0, iters=15):
    """
    A simple   smart   guess-and-check optimizer:

    - Keep a move direction d (2D vector).
    - If the last move improved f, keep moving in that direction (momentum).
    - If it got worse, reverse direction and shrink the step (backtracking).
    - Also tries coordinate nudges if stuck.

    This is basically a derivative-free pattern search.
    """
    vkm2 = clip_to_domain(np.array(v0, dtype=float))
    fkm2 = fv(vkm2)

    # Make an initial second guess by nudging v1
    d = np.array([1.0, 0.0])
    step = float(step0)

    vkm1 = clip_to_domain(vkm2 + step * d)
    fkm1 = fv(vkm1)

    # Choose a third guess based on whether that helped
    if fkm1 < fkm2:
        vk = clip_to_domain(vkm1 + step * d)   # keep going
    else:
        step *= 0.5
        d = -d
        vk = clip_to_domain(vkm2 + step * d)   # reverse + shrink

    fk = fv(vk)

    for _ in range(iters):
        #   logic   for v(k+1):
        # if f(k) < f(k-1): continue along last successful displacement
        # else: backtrack (reverse & shrink)
        last_disp = vk - vkm1

        if fk < fkm1:
            d = last_disp if np.linalg.norm(last_disp) > 1e-12 else d
            candidate = clip_to_domain(vk + d)  # same direction, same size as last move
        else:
            step *= 0.5
            d = -last_disp if np.linalg.norm(last_disp) > 1e-12 else -d
            candidate = clip_to_domain(vk + step * (d / (np.linalg.norm(d) + 1e-12)))

        fc = fv(candidate)

        # If we didnt improve, try small coordinate moves
        if fc >= fk:
            e1 = np.array([1.0, 0.0])
            e2 = np.array([0.0, 1.0])

            c1 = clip_to_domain(vk + step * e1)
            f1 = fv(c1)

            c2 = clip_to_domain(vk - step * e1)
            f2 = fv(c2)

            c3 = clip_to_domain(vk + step * e2)
            f3 = fv(c3)

            c4 = clip_to_domain(vk - step * e2)
            f4 = fv(c4)

            # pick best among candidate and coordinate nudges
            options = [(candidate, fc), (c1, f1), (c2, f2), (c3, f3), (c4, f4)]
            best_v, best_f = min(options, key=lambda t: t[1])
            candidate, fc = best_v, best_f

        # shift the history (k-2 <- k-1 <- k <- k+1)
        vkm2, fkm2 = vkm1, fkm1
        vkm1, fkm1 = vk, fk
        vk, fk = candidate, fc

    return vk, fk


if __name__ == "__main__":
    # Example run
    v_best, f_best = guess_and_check([3.0, -4.0], step0=2.0, iters=10)
    v1, v2 = v_best[0], v_best[1]
    f = f_best
    print(f" f([ {v1:7.4f} , {v2:7.4f} ]) = {f:7.4f} ")
```

Sample run and what the updates are doing (thinking in words).
Starting from an initial guess, the algorithm evaluates f at a new guess that moves in a current direction. If f improves, it continues moving in the same direction (a simple momentum idea). If f gets worse, it reverses direction and shrinks the step (backtracking). When it appears stuck, it tries small coordinate nudges in $\pm v_1$ and $\pm v_2$ directions and picks the best option. This logic is derivative-free, and it uses recent function values to decide which direction is likely to decrease f.

Result (best found in the provided run).
From the printed outputs, the smallest objective value obtained was

$$f(9.0000,\ -7.0000) = 0.9917,$$

which occurred at $v_1 = 9.0$, $v_2 = -7.0$.

$$\boxed{\textbf{Best guess found: } v_1 = 9.0000,\ v_2 = -7.0000, \quad \textbf{with} \quad f(v_1, v_2) = 0.9917.}$$

Appendix: Code and References (GitHub)

All code files used for Problems 4–10, along with any reference documents or screenshots used to verify installs and outputs, are located here:
https://github.com/PetersQuinn/HomeworkOne