

3.5 Problems



Problem 3.1 Write a program that meets the following requirements:

- It constructs a NumPy matrix `A` to represent the following mathematical matrix:

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{bmatrix}.$$

- It defines a function `left_up_sum(A: np.ndarray) -> np.ndarray` that adds the component (element) to the left and the component above (wrapping, if necessary) to each component. The function should pass through the array once, row-by-row, and return a new array. The function should be able to handle any size of matrix.
- It defines a function `left_up_sums(A: np.ndarray, n: int) -> np.ndarray` that executes `left_up_sum()` `n` times and returns a new array.
- It calls `left_up_sums()` on `A` and prints the returned array for the following values of `n`: 0, 1, 4.

Problem 3.2 The inner product of two real n -vectors x and y is defined as

$$\langle x, y \rangle = \sum_{i=0}^{n-1} x_i y_i.$$

The result is a scalar. The `np.inner()` and `np.dot()` functions can be used in NumPy to find the inner product of two vectors of the same size. In this problem, we will write our own function that computes the real inner product even if they are of different sizes. Write a program that meets the following requirements:

- It defines a function

```
| inner_flat_trunc(x: np.ndarray, y: np.ndarray) -> float
```

that returns the truncated inner product of vectors `a` and `b` even if the sizes of the vectors do not match by using a truncated version of the one that is too long. The function should handle any shape of input arrays by using the `flatten()` method before truncating and taking the inner product. If both input arrays do not have `dtype` attribute `np.dtype('float')` or `np.dtype('int')`, the function should raise a **TypeError** exception.

- It calls `inner_flat_trunc()` on the following arrays:
 - A pair of arrays from the lists:
`[-1.1, 3, 2.9, -1, -9.2, 0.1]` and `[1.3, 0.2, 8.3]`

- ii. An array of the integers from 0 through 13 and an array of the integers from 3 through 12
- iii. An array of 21 linearly spaced elements from 0 through 10 and an array of 11 linearly spaced elements from 5 through 25.
- iv. A pair of arrays of elements from the lists `[True, False, True]` and `[0, 1, 2]` (handle the exception in the main script so it runs without raising the exception)

Problem 3.3  Consider the following mathematical matrices and vectors:

$$A = \begin{bmatrix} 2 & 1 & 9 & 0 \\ 0 & -1 & -2 & 3 \\ -3 & 0 & 8 & -4 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 9 & -1 \\ 1 & 0 & 3 \\ 0 & -1 & 1 \end{bmatrix} \quad x = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \quad y = [3 \quad 0 \quad -1]. \quad (3.1)$$

Write a program that meets the following requirements:


- a. It defines NumPy arrays to represent A , B , (column vector) x , and (row vector) y .
- b. It computes and prints the following quantities:
 - i. BA
 - ii. $A^T B - 6J_{4 \times 3}$, where $J_{4 \times 3}$ is the 4×3 matrix of all 1 components
 - iii. $Bx + y^T$
 - iv. $xy + B$
 - v. yx
 - vi. $yB^{-1}x$
 - vii. CB , where C is the 3×3 submatrix of the first three columns of A

Problem 3.4  Consider the array:


```
| a = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]) # 4x3
```

Write a program that performs and prints the results of the following operations on a *without* using `for` loops:


- a. Adds 1 to all elements
- b. Adds 1 to the last column
- c. Flattens a to a vector
- d. Reshapes a into a 3×4 matrix
- e. Adds the vector `[1, 2, 3]` to each row
- f. Adds the vector `[1, 2, 3, 4]` to each column
- g. Reshapes a to a column vector
- h. Reshapes a to a row vector

Problem 3.5  Write vectorized Python functions that operate element-wise on array arguments for the following mathematical functions:


- a. $f(x) = x^2 + 3x + 9$
- b. $g(x) = 1 + \sin^2 x$
- c. $h(x, y) = e^{-3x} + \ln y$
- d. $F(x, y) = \lfloor x/y \rfloor$
- e. $G(x, y) = \begin{cases} x^2 + y^2 & \text{if } x > y \\ 2x & \text{otherwise} \end{cases}$

Problem 3.6  **DN** Write a program that graphs each of the following functions over the specified domain:

- a. $f(x) = \tanh(4 \sin x)$ for $x \in [-5, 8]$
- b. $g(x) = \sin \sqrt{x}$ for $x \in [0, 100]$
- c. $h(x) = \begin{cases} 0 & \text{if } x < 0 \\ e^{-x} \sin(2\pi x) & \text{otherwise} \end{cases}$ for $x \in [-2, 6]$

Problem 3.7  **WF** Write a program that loads and plots ideal gas data with the `engcom.data.ideal_gas()` function in the following way:

- a. The data it loads is over the volume domain: $V \in [0.1, 2.1] \text{ m}^3$
- b. The data it loads has 3 temperatures: $V = 300, 400, 500 \text{ K}$
- c. It plots in a single graphic P versus V for each of the three temperatures
- d. Each data point should be marked with a dot •
- e. Sequential data points should be connected by straight lines
- f. Each plot should be labeled with its corresponding temperature, either next to the plot or in a legend

Problem 3.8  **Y1** Use the data from problem 3.7 to write a program that meets the following requirements:

- a. It loads the pressure-volume-temperature data from problem 3.7.
- b. It estimates the work W done by the gas for each of the three values of temperatures via the integral equation

$$W = - \int_{0.1}^{2.1} P(V) dV.$$

Note: An integral can be estimated from discrete data via the trapezoidal rule, which can be executed with NumPy's `np.trapz()` function.

- c. It generates a bar chart comparing the three values of work (one for each temperature).


Problem 3.9  **KG** Write a program to bin data and create histogram charts that meets the following requirements:

- a. It defines a function

```
| binner(A: np.ndarray, nbins: int) -> (np.ndarray, np.ndarray)
```

that accepts an array `A` of data and returns an array for the frequency of the data in each bin and an array of the bin edges. Consider the following details:

- i. The bin edges should include the left edge and not the right edge, except the rightmost, which should include the right edge (“left” and “right” here mean lesser and greater).
 - ii. The bins should be of equal width.
 - iii. Give a default value (e.g., 10) for the `nbins` argument.
 - iv. Do *not* use the (nice) functions `np.histogram()` or `plt.hist()` for this exercise.
- b. It defines a function `histogram(A: np.ndarray, nbins: int)` that calls `binner()` and `plt.bar()` to generate a histogram chart.
 - c. It loads all of the thermal conductivity data from the `engcom.data` module with the `engcom.data.thermal_conductivity()` function.
 - d. It generates 3 histograms, one for each of the following material categories (key): “Metals”, “Liquids”, and “Gases”. Be sure to properly label the axes.

Problem 3.10  You will now create life. John Conway’s Game of Life is a cellular automata game that explores the notion of life. In this problem, you will write a program for the game, which is played on a 2D grid. The grid is composed of elements called cells, each of which can be either alive or dead at a given moment. The rules of the game are simple (Johnston and Greene 2022):

- If a cell is alive, it survives to the next generation if it has 2 or 3 live neighbors; otherwise it dies.
- If a cell is dead, it comes to life in the next generation if it has exactly 3 live neighbors; otherwise it stays dead.

The neighbors of a cell are those eight cells adjacent to it (including diagonals).

Write a program for playing the game of life that meets the following requirements:

- a. It defines a function

```
| game_of_life(A: np.ndarray)
```

that accepts a matrix `A` that encodes the starting state for the game. Use 1 to signify an alive cell and 0 to signify a dead cell. Consider the following details:

- i. The game is traditionally played on an infinite grid. However, your program should play the game of life on a torus (doughnut) made from sewing the opposite edges of the starting state `A` grid. For

instance, the neighbors above a cell in the top row are on the bottom row (i.e., neighbors wrap).

- ii. A visualization is required. A very useful Matplotlib function here is `plt.matshow(A)`, which will display the numerical values of a matrix in a grid. For instance, try the following:

```
| plt.matshow([[0,1,1],[1,0,1],[0,0,1]])
```

- iii. Strongly consider using additional functions to define operations like “evolve one generation,” “kill,” “animate,” and “visualize.”
- b. It calls `game_of_life()` on matrices corresponding to the following starting states:
 - i. A 5×5 grid of cells with the following pattern (blinker):


$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- ii. A 20×20 grid of cells, all dead (0) except a group near the center with the following pattern (glider):

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- iii. A 40×40 grid of cells, all dead (0) except a group near the center with the pattern that can be loaded as a list from the `engcom.data` module with the function call

```
| engcom.data.game_of_life_starts("gosper_glider")
```

Problem 3.11  **R9** In robotic path planning, it is often important to know if a given point (e.g., a potential location of the robot) is inside of a given polygon (e.g., a shape representing an obstacle). On a plane, a polygon can be defined by a list of n points (x_i, y_i) representing the vertices of the polygon P . One algorithm for determining if a given point R is in P is called the **winding number algorithm**, which computes the winding number ω as the sum of the angles θ_i between the vectors from R to consecutive vertices P_i and P_{i+1} of the polygon, denoted r_i and r_{i+1} , as shown in figure 3.7. In other words,

$$\omega = \frac{1}{2\pi} \sum_{i=0}^{n-1} \theta_i. \quad (3.2)$$

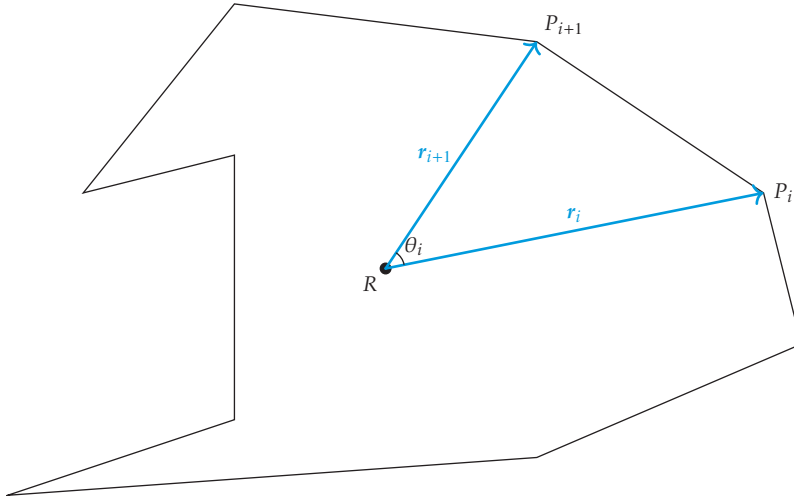


Figure 3.7. A polygon and vectors from R to two consecutive vertices.

It can be shown that if the winding number is 0, then R is outside the polygon; otherwise, it is inside. The angle ϕ_i of vector $r_i = [r_{ix}, r_{iy}]$ is

$$\phi_i = \arctan(r_{iy}/r_{ix}),$$

where we should use `np.atan2(riy, rix)` for computation. The difference between the angles of two consecutive vectors is

$$\theta_i = \phi_{i+1} - \phi_i \text{ where } |\theta_i| \leq \pi.$$


The bound $|\theta_i| \leq \pi$ must be enforced because the acute angle is used in equation (3.2), so if $\phi_{i+1} - \phi_i < -\pi$, we should add 2π and if $\phi_{i+1} - \phi_i > \pi$, we should subtract 2π .

Write a program that meets the following requirements:

- It defines a `Polygon` class that is constructed with instance attribute `vertices`, a list of (x_i, y_i) coordinate tuples defining the vertices of the polygon.
- The `Polygon` class has a method `plot()` that plots the polygon as a closed curve. If a point R is passed to the `plot()` method, it should appear as a single point on the plot.
- The `Polygon` class has a method `is_inside(R)` that checks if the point R (a tuple) is inside the polygon using a winding number algorithm. The method should return **True** if R is inside the polygon and **False** otherwise. Additional methods can be added to help with the computation of angles and other intermediate quantities.

- d. It tests the Polygon class with the following polygons and points, testing if the points are inside the polygon and plotting the polygon with the points:
- $P = [(5, 1), (2, 3), (-2, 3.5), (-4, 1), (-2, 1.5), (-2, -2), (-5, -3), (2, -2.5), (5.5, -1)]$ and the points $R_1 = (0, 0)$ and $R_2 = (-4, 0)$
 - $P = [(4, 1), (1, 2), (-1, 1), (-4, 2), (-5, -2), (-3, -2), (-5, -3), (2, -2), (5, -2)]$ and the points $R_1 = (0, 0)$ and $R_2 = (-4, 0)$


Restriction: Use only the NumPy and Matplotlib packages.

Problem 3.12  Create and test the Polygon class described in problem 3.11 with the following augmentation: instead of using the tangent function to compute the angle θ_i between consecutive vectors, use vector products.

The magnitude of the angle can be found from the dot product (`np.dot()`) expression

$$\cos \theta_i = \frac{\mathbf{r}_i \cdot \mathbf{r}_{i+1}}{|\mathbf{r}_i| |\mathbf{r}_{i+1}|},$$

where $|\cdot|$ is the vector length and \cdot is the dot product. To determine the sign of the angle, the cross product (`np.cross()`) can be used because $\mathbf{r}_i \times \mathbf{r}_{i+1} = -\mathbf{r}_{i+1} \times \mathbf{r}_i$. In other words, the direction of rotation between \mathbf{r}_i and \mathbf{r}_{i+1} can be found from the sign of their cross product.

Problem 3.13  In this problem, we will develop a model of a mass's trajectory on a 2D rectangular region of space with walls off which the mass bounces elastically (i.e., with no loss of kinetic energy). We divide time into discrete moments t_0, t_1, \dots, t_{n-1} with equal intervals Δt . The mass's position at time t_i is given by the vector $\mathbf{r}_i = [x_i \ y_i]^\top$. In discrete time, the mass's velocity at time t_i is given by the vector

$$\mathbf{v}_i = \frac{\mathbf{r}_{i+1} - \mathbf{r}_i}{\Delta t},$$

which approximates the derivative of the position vector with respect to time.


We will model the trajectories of several masses. In this problem, there will be no forces acting on the masses, so they will move at a constant velocity. (See problem 3.14 for an extension with force fields.)

Write a program that meets the following requirements:

- Define a class `State` that represents the state of a mass at a given time. The class should have the following attributes:
 - `position`, a NumPy array representing the position at the current time
 - `velocity`, a NumPy array representing the velocity at the current time

- b. Define a class `Region` that represents the 2D rectangle in which the mass moves. Use Cartesian coordinates with the origin at the lower-left corner of the rectangle, the x -axis increasing to the right, and the y -axis increasing upwards. The class should have the following attributes:
 - i. `width`, the width of the region
 - ii. `height`, the height of the region
- c. Define a class `Trajectory` that represents the trajectory of a mass through time. The class should have the following attributes:
 - i. `initial_state`, a `State` instance representing the mass's initial state (required to initialize)
 - ii. `time`, a NumPy array representing the corresponding times (required to initialize)
 - iii. `region`, a `Region` instance representing the region in which the mass moves (required to initialize)
 - iv. `states`, a list of `State` instances representing the mass's states at each time step
- d. Define an instance method `bounce(s: State) -> State` for the `Trajectory` class that updates the mass's position and velocity when it bounces off a wall. (The angle of incidence equals the angle of reflection.)
- e. Define an instance method `step_state()` for the `Trajectory` class that advances the mass's state by one time step. Wall bouncing should be handled here by calling `bounce()`.
- f. Define an instance method `compute()` for the `Trajectory` class that advances the mass's state through time until a specified end time. This method should call `step_state()` repeatedly and populate the `states` attribute.
- g. Define class instance methods `get_x()` and `get_y()` for the `Trajectory` class that return NumPy arrays of the x and y components of the mass's positions at each time step.
- h. Define a class `Trajectories` that represents a collection of `Trajectory` instances. The class should have the attribute `trajectories`, a list of `Trajectory` instances.
- i. Define an instance method `plot()` for the `Trajectories` class that plots the trajectories of all masses in the collection. The method should plot the trajectories as differently colored lines and the initial positions as points.
- j. Test the classes and methods with the following parameters:
 - i. A time interval of $[0, 10]$ s with 101 time steps
 - ii. A region with width 10 and height 5. (Use units of meters.)
 - iii. A collection of 3 trajectories with the following initial states:

- i. Mass 0 at (1, 1) m with velocity (1, 0) m/s
- ii. Mass 1 at (2, 2) m with velocity (0, 1) m/s
- iii. Mass 2 at (3, 3) m with velocity (1, -1) m/s
- iv. Mass 3 at (4, 4) m with velocity (0.2, 0.6) m/s

Problem 3.14  **FORCEFIELDS** In this problem, we will extend the model of a mass's trajectory on a 2D rectangular region of space with walls from problem 3.13 to include force fields acting on the masses. The force field is a vector field $F(\mathbf{r})$ that gives the force acting on the mass at position \mathbf{r} . The acceleration of the mass at position \mathbf{r} is given by Newton's second law:

$$\mathbf{a}(\mathbf{r}) = \mathbf{F}(\mathbf{r})/m,$$

where m is the mass of the object. In discrete time, the mass's velocity at time t_i is given by the vector

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t,$$

where $\mathbf{a}_i = \mathbf{F}(\mathbf{r}_i)/m$.

A force field can be represented by a function that takes a position vector as input and returns a force vector. Consider the following force fields:

$$\mathbf{F}_0(\mathbf{r}) = c_0 \frac{\mathbf{r} - \mathbf{r}_0}{|\mathbf{r} - \mathbf{r}_0|^2} \text{ and} \quad (3.3)$$

$$\mathbf{F}_1(\mathbf{r}) = c_1 \frac{\mathbf{r}_1 - \mathbf{r}}{|\mathbf{r}_1 - \mathbf{r}|^2}, \quad (3.4)$$

where $\mathbf{r}_0, \mathbf{r}_1$ are fixed positions vectors, $|\cdot|$ is the vector length, and c_0, c_1 are positive constants with units N·m that determine the strength of the force fields. \mathbf{F}_0 points radially outward from \mathbf{r}_0 so it is a “repeller,” and \mathbf{F}_1 points radially inward toward \mathbf{r}_1 so it is an “attractor.”

Write a program that meets the following requirements, starting with the classes from problem 3.13:

- a. Extend the Trajectory class to include a mass attribute that represents the mass of the object.
- b. Extend the Trajectory class to include a list of force fields as an attribute. The force fields should be functions that take a position vector as input and return a force vector.
- c. Extend the `step_state()` method of the Trajectory class to include the acceleration due to the sum of the force fields acting on the mass.
- d. Extend the `step_state()` method of the Trajectory class to include a damping force given by

$$\mathbf{F}_{\text{damp}} = -0.3|\mathbf{v}|\mathbf{v}.$$

- e. Extend the `Trajectories` class constructor to accept a list of mass values and a list of force fields for each trajectory.
- f. Define the force fields F_0 and F_1 as functions and test the program with the following parameters:
 - i. Force field vectors $r_0 = (10, 6)$ and $r_1 = (10, 4)$ and strength constants $c_0 = c_1 = 0.1 \text{ N}\cdot\text{m}$
 - ii. A time interval of $[0, 700]$ s with 701 time steps
 - iii. A region with width 20 and height 10. (Use units of meters.)
 - iv. A collection of trajectories with the following masses and initial states:
 - i. Mass $m_0 = 1 \text{ kg}$ at $(3, 4.98) \text{ m}$ and at rest
 - ii. Mass $m_1 = 1 \text{ kg}$ at $(3, 4.99) \text{ m}$ and at rest
 - iii. Mass $m_2 = 1 \text{ kg}$ at $(3, 5.01) \text{ m}$ and at rest
 - iv. Mass $m_3 = 1 \text{ kg}$ at $(3, 5.02) \text{ m}$ and at rest

4 Symbolic Analysis and Design



A **symbolic analysis**, sometimes called “analytic” as opposed to “numerical,” is one that manipulates symbols called **symbolic variables**, which represent quantities. In symbolic analysis, variables of interest are solved for by means of techniques from all branches of mathematics. For engineering symbolic analysis, of particular importance are the mathematical techniques of geometry, algebra, calculus, analysis,¹ discrete mathematics, logic, set theory, probability, and statistics.

Applied to an engineering problem, the techniques of these branches of mathematics often yield **symbolic solutions** (also called “analytic” solutions), exact solutions for symbolic variables. However, there are many problems for which symbolic solutions do not exist, are unknown, are difficult to obtain, or would yield little insight into the problem (e.g., when the solution cannot be expressed simply). In such cases, the techniques of numerical analysis (chapter 5) are indicated. For those problems with nice symbolic solutions (i.e., those that can be expressed simply and can be obtained without exorbitant work), there are distinct advantages to finding symbolic solutions:

1. Symbolic solutions have provable properties (e.g., stability and bounds)
2. Symbolic solutions give designers insight into the ways design parameters affect performance (e.g., increasing the mass of this component will reduce a vibration output)
3. Symbolic solutions are much more general than numerical solutions, which are only valid for a specific set of parameters, initial conditions, boundary conditions, etc.

Computers have the ability to manipulate symbolic variables and the expressions and functions associated with them. Software designed for this purpose is called a **computer algebra system (CAS)**. Many of the techniques from the mathematics curriculum of an engineering degree are available in CASs. Popular CASs include

1. The mathematical field of analysis includes real analysis, complex analysis, differential equations, and vector analysis. Analysis developed from calculus.

Mathematica, Maple, the Symbolic Math Toolbox of MATLAB, SageMath, and the SymPy package of Python. Although most of these have an application programming interface (API) for Python, the only one that is exclusively written in and for Python is the SymPy package, and therefore we will use this as our CAS.

The SymPy package is available in the base Anaconda environment. It can be imported in a program with the following statement:

```
| import sympy as sp
```

We use the alias `sp` throughout the text.

4.1 Symbolic Expressions, Variables, and Functions



In SymPy, a **symbolic expression** is comprised of SymPy objects. Unlike numerical expressions, these are not automatically evaluated to integer or floating-point numbers. For instance, using the standard library `math` module, the expression `math.sqrt(3)/2` immediately evaluates to the floating-point approximation of about `0.866`. However, in SymPy, something else happens:²

```
| sp.sqrt(3) / 2
```

$$\frac{\sqrt{3}}{2}$$

This is an *exact* representation of the mathematical expression, as opposed to the approximation obtained previously.

A symbolic expression can be represented as an **expression tree**:

```
| sp.srepr(sp.sqrt(3) / 2) # Show expression tree representation
```

```
| 'Mul(Rational(1, 2), Pow(Integer(3), Rational(1, 2)))'
```

This can be visualized as a tree graph like that shown in figure 4.1.

2. We are pretty printing results that are mathematical expressions.