Programming 2025/26 — Assignment 2

1 About the assignment

You will write Python functions to make various calculations and draw various plots.

To support your programming, you will also need to do some mathematics.

Your mark for the assignment will be based on both the programming and mathematical parts. This is a programming module and so the programming is the main part of this assignment. Nonetheless, where there are relevant mathematical tasks, the idea is that your solution to a programming task should be based on them. Tasks where some maths is expected have "(Math)" after the task number. While it is possible to get full marks just from your Python code, additional partial credit marks can be given for your mathematical working for these tasks if your Python code if not fully correct.

It's important that you test your functions before submitting them. The Moodle server will not perform any checks on your code. Therefore, pay close attention to examples given in relation to the earlier tasks and make sure that your functions work for those.

The deadline for submission is noon on January 21.

1.1 Rules

The assignment should be your own work without help from others. You may view and use any pages or code linked from Blackboard, as well as web pages referenced below (just click on the link which is formatted as underlined green text) and may look up online or in books the meaning of technical terms used in the assignment but should not seek further assistance from books or online sources. You may reuse, or modify, any code that you have written yourself in previous circumstances. You may ask generative AI for help with technical terms and general concepts (though you will likely find more accurate information elsewhere), but you must not submit any code that was written by generative AI.

You may import the math and numpy modules or individual items from those modules. You may also make the usual import of the pyplot part of matplotlib as plt and this is assumed in examples given below. Other imports are not allowed, specifically not other parts of matplotlib such as matplotlib.patches and matplotlib. collections .

Some marks will be reserved for the quality of plots produced, for example for sensible use of colour to distinguish different objects appearing in a plot. Marks will also be reserved for quality of code. In particular, there is a certain hierarchy to some of the tasks where a function written for an earlier task would naturally be called for a later task and doing so may be rewarded in marking.

Your submission will consist of two files:

- A single .py file of Python code which contains only imports, function definitions and comments. This should be submitted to the Moodle server. Please note that it is very important that your submission is a **single .py file**. Please do not submit multiple files zipped together and note that I will not be able to read any files that are submitted in a proprietory Apple format. If you submit your code as a Word document or a PDF, I will try my best to extract your code, but the conversion will likely cause formatting errors which will severely impact your

mark. Functions may call each other and you may include additional functions to support the ones required as solutions to the assignment. Feel free to include comments and to leave commented-out code or commented-out print statements as examples of how your functions would be used. Do not call plt.show() in functions which are submitted as solutions to tasks.

- A single PDF file of your mathematical work, clearly identifying to which task each part of your file relates. This should be submitted to Gradescope having followed the link from Blackboard.

1.2 Difficulty of the tasks and time required

The tasks are ordered in a way that seems natural for the material. Tasks 1, 2, 3, 5 and 6 are deliberately clearly sign-posted. Tasks 4 and 7 are similar but you have to work out more for yourself. These tasks together will account for 50% of the marks.

Tasks 8 to 13 deal with the more challenging topic of simulating a individual particle bouncing off obstacles. Tasks 14 and 15 present a significant extra challenge of finding the tracks a particle can take between a specified startpoint and endpoint. These questions are meant to be open ended – there are multiple different correct approaches. Part of the assessment is being able to chose the correct one, and so I will not be able to advise you what is the best approach to take, nor how the code should behave in 'edge cases'.

Only you can decide how much time to spend on the assignment between now and the deadline. However, I do not expect you to spend days on this assignment or to make yourself miserable trying to answer questions for which you are not ready. The more challenging tasks are provided to stimulate and provide interest for students who have been enjoying programming. If you are taking more than five hours in total to do this assignment (it may take many of you much less time), I would recommend that you consider how best to 'wrap up' the questions you are working on – it is still possible to get partial credit for incomplete answers.

You have been given a very long amount of time to do the assignment as you have your holidays coming up, and you will have commitments from your other modules, but I strongly recommend that you start working on the assignment as soon as possible.

2 The assignment

2.1 Definitions of objects in the plane and representations in Python

As we saw during the practicals, a point in the plane can be represented in Python by a list of two numbers, the coordinates of the point. For example the point $(5, 4)$ would be represented by [5, 4] (or of course by various other forms such as [5.0, 4.0]).

An equation of the form $ax + by + c = 0$, where a, b and c are real numbers, defines a line in the plane. In Python, we can represent such an equation by a Python list containing a, b and c. For example, the equation $-3x + 2y + 5 = 0$ would be represented by [-3, 2, 5]. We can also understand this to represent the line defined by the equation (noting that infinitely many equations define the same line).

Any two distinct points in the plane define a line segment connecting the points. In Python, this can be represented as a list containing the two points, each point being represented as a list of two numbers. For example the line segment connecting $(1.5, 2.4)$ to $(5, 4)$ would be represented as [[1.5, 2.4], [5, 4]].

A directed line segment is a line segment going from one point in the plane to another. This can be represented in the same way as a line segment but now the order of the points matters: [[1, 3], [2, -1]] is the same line segment as [[2, -1], [1, 3]] but they are different directed line segments. The first represents $(1, 3) \rightarrow (2, -1)$ whereas the second represents $(2, -1) \rightarrow (1, 3)$.

A ray is a part of a line that starts at a point in the plane and extends to infinity in a particular direction from that point. A ray can be represented in Python by a list with two elements: the first element is a list of two numbers representing a point and the second element is an angle (in radians) indicating the direction in the usual definition for polar coordinates. For example, [[0, 0], 0] denotes the ray starting at the origin and heading along the x-axis in the direction of increasing x, and [[1.2, 2.1], math.pi/2] denotes the ray starting at $(1.2, 2.1)$ and heading away parallel to the y-axis in the direction of increasing y.

A path in the plane is a sequence of directed line segments where the end point of one segment is the starting point of the next segment. A path can be represented in Python as a list of points. For example the path $(1, 2) \rightarrow (3, -1) \rightarrow (4.5, 6)$ consists of the directed line segments $(1, 2) \rightarrow (3, -1)$ and $(3, -1) \rightarrow (4.5, 6)$ and would be represented in Python by [[1, 2], [3, -1], [4.5, 6]].

A rectangle in the plane with sides parallel to the coordinate axes is determined by a range of values for x and a range of values for y. A rectangle of this kind will be called a window in what follows. A window can be represented in Python as a list containing two lists each of which is a list of two numbers. For example [[1, 3.4], [-2.1, 1]] represents the window where x ranges from 1 to 3.4 and y ranges from -2.1 to 1.

These terms will be used in subsequent tasks, and they should be represented in your Python code in the exact way as described in this section.

2.2 Plotting objects in the plane

In what follows, we are going to find it useful to be able to add individual points, line segments etc to a plot. You will write functions that do this. It is important that these functions do not themselves call plt.show as we want to be able to add multiple things to a plot before displaying it on the screen using plt.show(). If you find this confusing, look at the example at the end of this section.

On Blackboard, find drawdirseg.py. It defines a function to add a directed line segment to a plot in a specified colour. The function is draw_directed_segment(dirseg, col) and has two arguments: a directed line segment dirseg represented as described above and a colour col specified in a way acceptable to matplotlib (more on this below).

Have a look at the function. Note that it does not return anything. Instead it has an effect on the plot being produced by calling plt.plot and plt.quiver. In the latter call, the setting of width=.004 works OK on my screen but you may find that you need to adjust it. Note also how the col argument is used to change to the colour in the calls to plt.plot and plt.quiver.

Task 1 Write the following Python functions based on on the Python representations of planar objects described in section 2.1:
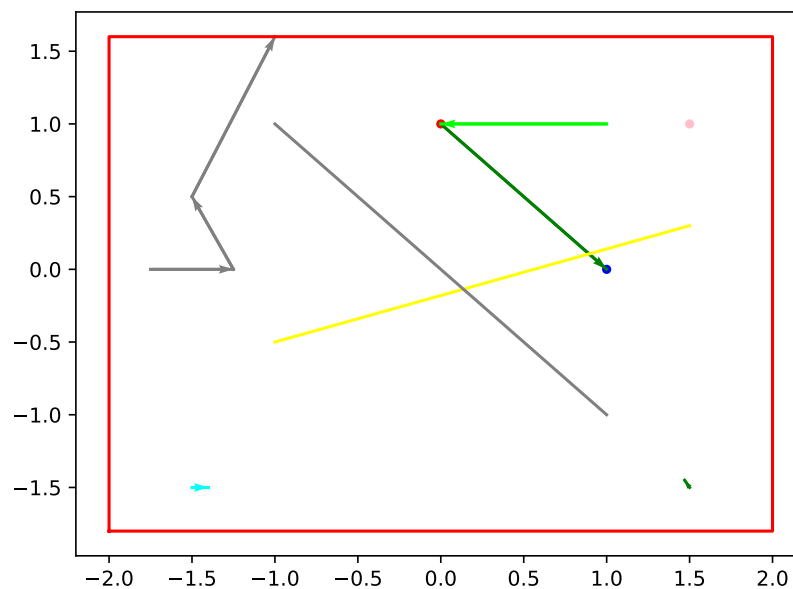draw_point(pt, col) to add the point specified by pt to a plot in the specified colour.
draw_segment(seg, col) that adds the line segment specified by seg to a plot in the specified colour.
draw_path(path, col) that adds the path specified by path to a plot as a sequence of directed line segments in the specified colour.
draw_window(win, col) to add the boundary of the window specified by win to a plot in the specified colour.

All of these can be implemented by using plt.plot or functions that are part of the assignment to do the actual work.

To test your code and the draw_directed_segment function defined above, find example1.py on Blackboard which contains some code you can use to test your your four functions and my draw_directed_segment. When run, the result should be similar to:

In the code, note the use of two different ways to specify a colour to matplotlib. In fact a colour can be specified in several different ways described in the matplotlib tutorial on colors. The two easiest are by name or as an RGB color by providing a list of three numbers each between 0 and 1 (the amounts of red, green and blue light to include in the color). If you don't know what an RGB colour is, see the Wikipedia

article about RGB colours.

In the rest of this assignment, functions written to draw things should make use of the functions defined in this section 2.2. You are allowed to copy and modify any of the code in the drawdirseg.py and the example files on Blackboard in your solution.

2.3 Intersections of lines and line segments

Task 2 (Math) Work out mathematically how to find where two lines in the plane cross: where they <u>intersect</u>. Assume that the lines are specified by equations of the the form described in section 2.1. Work out how to detect the case that the lines are parallel and do not cross.

Write a Python function lines_intersect(line1, line2) where line1 and line2 are lists of three numbers, each representing a line in the plane as described in section 2.1. If the lines cross, the function should compute the point of intersection and return it using the Python representation as a list of two numbers. In the case where the lines are parallel, the function should return the special Python value None. Make sure that your function works when either line is parallel to the x-axis or the y-axis, i.e. when the corresponding value of a or b is zero.

For example, lines_intersect([1, 1, -3], [1, -1, 1]) should return [1.0, 2.0] and lines_intersect([1, 1, 0], [2, 2, 3]) should return None

Task 3 (Math) The goal in this task is to write a function that computes the intersection between two line segments. The task is broken into sub-tasks to help you.

Task 3a For any line segment in the plane, there is a unique line to which it belongs. Write a Python function line_from_segment(seg) where the argument seg is a line segment represented as described in section 2.1. The function should return an equation for the line to which the segment belongs using the representation for lines described in section 2.1.

For example, line_from_segment([[0,0], [1,1]]) might return [1,-1, 0] and line_from_segment([[3,0], [3,2]]) might return [2, 0, -6] (remember that there are other correct answers).

Task 3b Given a line segment and a point, either the point lies on the line to which the segment belongs or it does not. This sub-task is concerned with the situation where the point does lie on that line. Then the point may or may not lie on the segment. Write a Python function point_on_segment(point, seg) that returns True if the point lies on the segment and False if does not. Your calculation should assume that the point does lie on the line to which the segment belongs (subject of course to the usual issues about accuracy of calculations using floating point numbers).

For example, point_on_segment([0, -1], [[1,1], [3, 5]]) should return False and point_on_segment([2, 3], [[1,1], [3, 5]]) should return True

Hint: if the point lies on the segment, the vectors formed by the difference between the point and the two ends of the segment point in opposite directions and their dot product will therefore be negative.

Task 3c Write a Python function segments_intersect(seg1, seg2) that computes the intersection between two line segments represented as described in section 2.1. If the segments cross, the function should return the point of intersection. Otherwise, it should return None, i.e. when there no intersection or when the segments overlap and the intersection consists of a line segment.

Hint: one way to compute the intersection of two line segments is to first find the intersection of the two lines two which the segments belong. You can then check if that point lies on both segments.

Task 4 (Math) Write a Python function draw_segment_intersect(ray, seg) that computes the intersection between a ray ray and a line segment seg. If the ray crosses the segment, the function should return the point of intersection. Otherwise, it should return None, i.e. when there is no intersection or when the the ray and segment overlap and the intersection consists of a line segment.

Hint: follow a similar approach to task 3.

2.4 Plotting lines and rays in windows

Lines and rays extend to infinity which is a nuisance when plotting them. Therefore, it can be useful to specify a window limiting the parts of lines and rays that should appear. The tasks in this section provide simple applications for some functions written for earlier tasks.
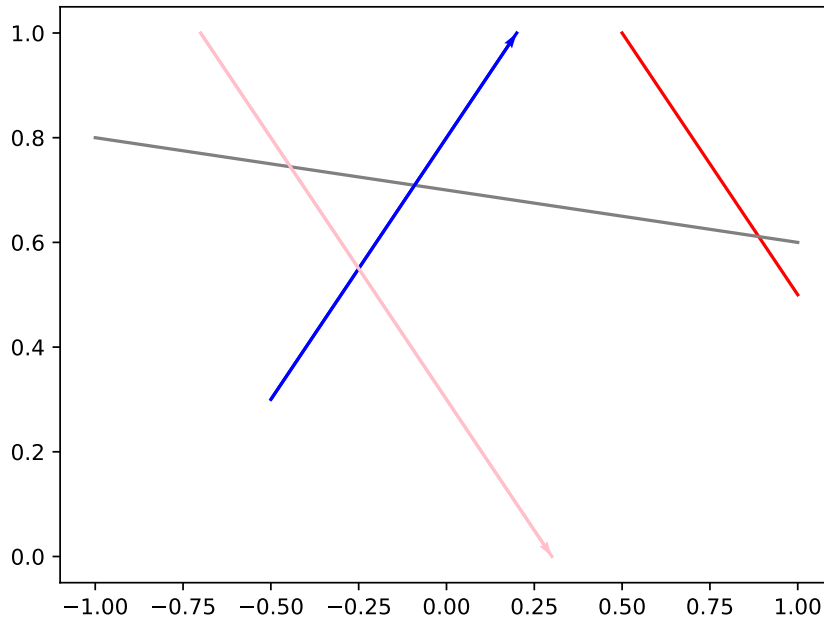
Task 5 Write draw_line_in_window(line, window, col) which adds to a plot the part of the line specified by line that lies in the window specified by window, drawn in colour col. This function should not draw the window, just the required line segment. If no part of the line lies in the window, the function should not add anything to the plot.

Hint: make use of functions written in section 2.3 to determine a line segment to draw. You may find it helpful first to write a functions to find the intersection between a line and a segment and to compute the boundary of a window a a list of line segments.

Task 6 Write a Python function draw_lines_and_window(line1, line2, window) which adds to a plot the parts of two lines that lie in the specified window and adds a markers at the point where the lines intersect if they intersect at a single point and that point lies in the window. It should also draw the boundary of the window. Make choices of colours that highlight the different objects.

Task 7 Write Python function draw_ray_in_window(ray, window, col) that adds to a plot the directed line segment that is the part of the ray ray that lies in the window window, drawn in colour col.

To help you test your functions for tasks 5 and 7, copy the code in example2.py on Blackboard. When run, the result should be similar to:
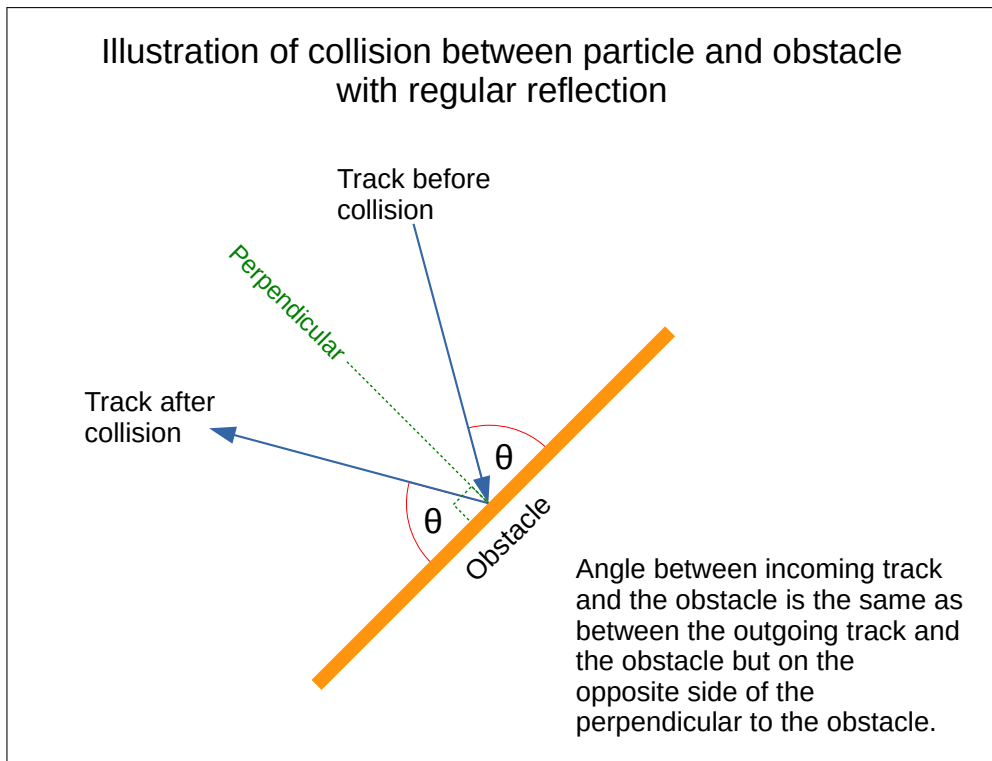
2.5 Bouncing particle

Suppose that:

- a ray represents the track of a particle starting at a specified point in the plane and moving in a specified direction, when there is nothing with which the particle can collide;

- a line segment in the plane represents an obstacle with which the particle might collide;

- if the ray and line segment intersect, the particle collides with the obstacle and bounces off in the complementary direction as shown in the figure below, i.e. there is a new ray, representing the particle's track after the collision. The new ray starts from the point of collision and the direction of the new ray depends on the orientation of the line segment and the direction of the original ray.

You might recognise this as a model of what happens when (a photon of) light hits a mirror or a small hard ball collides with a hard surface.

Illustration of collision between particle and obstacle with regular reflection

Task 8 (Math) Write Python function one_bounce(ray, seg, win) where ray represents the initial track of a particle and seg represents a single obstacle. The function should return a path representing the track of the particle until it leaves the window specified by win. This will be a path with a single segment if the particle does not collide with the obstacle and with two segments if it does. You may assume that the starting point of ray lies within the window and that the whole of the obstacle lies within the window.

For example, after seg = [[0,3], [3,0]] and win = [[-1,4], [-1, 4]], one_bounce([[1,1], math.pi/6], seg, win) should return [[1, 1], [1.634, 1.366], [0.268, -1.0]] (rounding coordinates here to 3 decimal places just for readability – **your code does not need to do any rounding**) and one_bounce([[0.75, 2], math.pi+0.1], seg, win) should return [[0.75, 2], [-1.0, 1.824]].

Hint for the Math part: consider three unit vectors v_{in}, v_{out} and s representing the direction of the inbound and outbound rays and the orientation of the segment. Let $s^\dagger$ be a unit vector orthogonal to s. From the geometry in the figure show that $v_{in}.s = v_{out}.s$ and $v_{in}.s^\dagger = -v_{out}.s^\dagger$ where . denotes the usual dot product for vectors. Hence express v_{out} in terms of v_{in} and s.

Hint for the Python part: read the help for the atan2 function in the math module, or try making some plots of various Python arctan functions to try to understand how they work.

Task 9 Write draw_one_bounce(path, seg, win) that adds a depiction of the bouncing process to a plot. Here path would be a list of directed line segments (obtained from calling one_bounce) and seg and win would be the same values as passed to one_bounce).

For example, one might do

```
window = [[0,1],[0,1]]
obstacle = [[0.25,0.75], [0.75, 0.25]]
path = one_bounce([[0.1, 0.1], math.pi/4], obstacle, window)
```

```
draw_one_bounce(path, obstacle, window)
plt.show()
```

When writing draw_one_bounce, you should choose colors for the elements being added to the plot that distinguish the different types of object.

Task 10 Write Python function multi_bounce(ray, seglist, win, maxpathlen) where ray and win are as before, seglist is a list of obstacles and maxpathlen is a limit on the number of collisions to simulate. The function should return a path representing the track of the particle until either it leaves the window or the length (number of segments) of the path is maxpathlen.

Task 11 Write draw_multi_bounce(path, seglist, win) that adds adds a depiction of the bouncing process to a plot. Here path would be a path (obtained from calling multi_bounce) and seglist and win would be the same values as passed to multi_bounce).

Task 12 Write draw_n_multi_bounce(pathlist, seglist, win) that adds to a plot a depiction of the bouncing process for n particles starting with different rays but all in the same window and with the same obstacles. Here pathlist would be a list of length n, each element of which would be a path. The paths would be obtained by calling multi_bounce n times with different rays using the same values of seglist and win each time.

Task 13 One way of "keeping the process alive" within the window (i.e. to allow the path to continue even after it reaches the edge of the window) is for the window to "wrap around", i.e. when the particle passes through an edge of the window, it reappears at the corresponding point on the opposite edge and continues in the same direction as before. Write Python function multi_bounce_wrap(ray, seglist, window, pathlen) that returns a list of paths representing the track of the particle until the total length of the paths in the list is pathlen. The first path in the list starts at the starting point and ends when the particle first hits an edge (or when the length of the path is pathlen). The next path starts where the particle reappears on the opposite edge and so on.

2.6 Finding the path from the start point and end point

This section will ask you to find a path between a startpoint and an endpoint, including reflections off of obstacles, without a starting angle for the ray being provided.

Task 14 (Math) Write Python function single_path_finder(startpoint, endpoint, seglist, window) where starpoint is a list representing a starting point of a track of a particle and endpoint is a list representing the endpoint of the track of the particle, and seglist and window are defined as before. The window should not "wrap around", that is, you can ignore the additional work you did in the previous task. The function should return a paths representing the shortest track of the particle between the startpoint and the endpoint. If it is not possible for the particle to reach the endpoint from the startpoint, the function should return the value None. You will receive most of the marks for this question if your function returns any valid track, even if it is not the shortest. If there are multiple shortest tracks, you will receive full marks as long as you return one of them.

Task 15 (Math) Modify your work from the previous task to write a new Python function multiple_path_finder(startpoint, endpoint, seglist, window, n) which

returns a list of the n shortest tracks (each itself represented as a path) a particle can take between the startpoint and endpoint. If there are no possible tracks between the startpoint or endpoint, the function should return the value None. If there are fewer then n tracks that are possible, then your function should only return those tracks that are possible.