MAT101 Programming – Mock Exam 2

No Submission

Exercise 1. **25** P.

Random walks In this exercise we take a look at the random walk which is used to model a wide variety of things. The idea in our setting is that the walk starts at 0 and in each step jumps by $+1$ with probability p and -1 with probability $1-p$.

a) Write a *recursive* function `random_walk_recursive(steps)` which takes as input an integer *steps* corresponding to the number of steps and returns the end position of the walk for $p = 1 - p = 0.5$ after *steps* steps and save it in a file 'random_walk_functions.py' **4** P.

b) Also write a *iterative* function `random_walk_iterative(steps)` realizing the random walk using a loop in the same file. **4** P.

c) Change the code of both of your functions in such a way that you can enter a second variable p specifying the probability of $+1$ in each step. However, your function should still work if only the number of steps is specified in which case we want to work with $p = 0.5$. **4** P.

d) In the following we work with the default $p = 0.5$. Create a new python file 'random_walk_script.py' and import both of your functions. Compare the run times of both for 100 steps. **4** P.

e) Now we take a look at the results of the walk by running it several times. Add a function `plot_results(steps, iterations)` to your scripts that does the following: it takes as input two integers, one denoting the number of steps of the walks and one the number of runs of walk. The function should first run one of your implementations as often as specified and store the results in an array. Next, visualize the output by plotting the number of occurrences of each outcome. You can do so either using a histogram or plotting the numbers of occurrences as dots.
Save the result for 100 steps and 10000 iterations. What do you observe regarding the output?
Note that we still work with $p = 0.5$ **9** P.

Exercise 2. **30** P.

Having fun with fractals Chances are you have heard of fractals. In case you have not, a fractal is a cute geometric shape containing detailed structure at arbitrarily small scales. In this part we explore some fractals visually and some of them are gorgeous indeed!

a) Let us start with Pascal's triangle. The entry in the nth row and kth column of Pascal's triangle is $\frac{n!}{k!(n-k!)}$. One idea instead is to investigate the evenness or oddness of these elements. Write a function `Pascal_triangle` that returns in an array Pascal's triangle up to n rows and then convert it to modulo 2 so that you have 1 for the entry which is odd, and 0 if it is even. **6** P.

b) It is time to plot the fractal you just got. Put each row of the binary triangle in a row of a matrix (you need to shift the rows to the left and align them for simplicity, and fill the empty cells with zero). Plot the matrix for 8, 16, and 64 rows, respectively in two dimensions. **4** P.

You have just visualised a simple fractal. There are many more out there and you can explore them if you are curious. For the rest of this part we step up to visualise one of the famous ones, called Mandelbrot set. The Mandelbrot set is the set of complex numbers, c, for which an infinite sequence of numbers, z_0, z_1, …, z_n, …, remains bounded. That is $\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$

c) Implement the *recursive* function which gives you this relation and call it `z`. The function should return the $n+1$th element of the sequence. **5** P.

d) For $c = 1$ write the first ten outputs. Do you think $c = 1$ belongs to Mandelbrot set? Why? How about $c = -1$? **3** P.

Let us now find multiple c's that are actually members of the set. First we need to generate numerous complex numbers to perform the test. It is easier if we do so in a matrix form (but it is not necessary) and proceed by checking if each of the elements of that matrix is a member of Mandelbrot set or not. Then, simply throw out the ones that are not, and make the plot for the remaining ones that are the actual members of the set.

e) Write a function `c` which takes as arguments ranges of imaginary and real parts of a complex number along with a parameter for adjusting the number of equally spaced numbers, `num_steps`, which tells you how many number you need in each range (this parameter would adjust the resolution of the final plot). The function returns the matrix of complex numbers made in these ranges. **5** P.

f) Check whether each of these elements (the c's) are member of Mandelbrot set or not via comparing the magnitude of the number *after iteration* with 2 (pick a number for iterations). Keep the ones remained smaller than 2. Those are the stable ones! **5** P.

g) Scatter plot the imaginary parts vs. real parts of the stable ones. **2** P.

Exercise 3. **10** P.

Classy geometry Classes can inherit from each other which you have seen as derived classes.

a) Define a class called `regular_polygons`. We want the objects of this class to have attributes called `n` for the number of edges and `l` for the length of edges. Also, the method `area` that gives the area of the objects ($area = \frac{n}{2} lR$ where n is the number of sides in the polygon, l is the length of one edge of the polygon, and R is the radius of the inscribed circle). **5** P.

b) What if we have a circle? It is an asymptotic case of a regular polygon. Define a new class called `circles` which inherits from the previous class, and overwrite `area` method appropriately so that it gives the area of a circle (Also `n` attribute has too have the value `numpy.inf` in this case). **5** P.