ENGG1003 - Lab Week 8

Brenton Schulz

Task 1: Basic 2D Array Indexing

The following 2D array:

$$x = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

can be created in Python with the code:

```
x = np.array([ [4,5,6], [7,8,9], [10,11,12] ])
```

Initialise this array in your own Python script then write additional code which uses array indexing or slicing to:

- Print the `4`
- Print the `9`
- Print the bottom row
- Print the four values in the top left as a 2x2 array:

```
[[4 5]
 [7 8]]
```

Task 2: Intermediate Indexing and Slicing 2D Arrays

Similarly to the "plus symbol" example in the Week 7 Thursday lecture (slides 3-7), create a 10x10 2D array of 1s and, using array slicing, draw a 1-pixel black square along the border (edge).

Task 3: More Complicated 2D Array Indexing and Slicing

Draw a 1-pixel white square of side length `N`, with upper left corner at row `r` and column `c`, inside an `MxM` array. Note that `r=0` and `c=0` indexes the top left corner. Initialise `N`, `r`, `c`, and `M` near the start of your Python script. Confirm your method works with several different test values.

Implement this problem with two methods:

- By writing four `for` or `while` loops which index the array elements individually - one loop for each side of the square
- With four applications of array slicing

For initial testing, recreate Figure 1 where `r=3`, `c=2`, `N=4` and `M=10`.

Extension: Write a function which takes `N`, `r`, `c`, and `M` as arguments and returns the array.

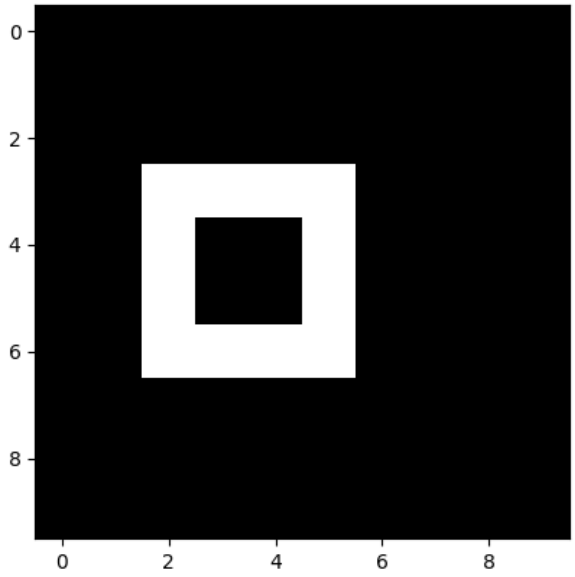

Figure 1: A square of side length 4 drawn from row index 2 and column index 3 drawn on a 10x10 2D array.

Task 4: Mapping Cartesian Coordinates to 2D Array Indices

Consider an autonomous robot which receives accurate position information from external sensors and needs to locate itself on a map stored in a 2D array[a]. Assume that the external sensors provide x-y coordinates from some known "origin" and that the robot is constrained to a room of known size. The origin is not necessarily in the corner of the room but can be located anywhere. The x and y values of the room edges are known. The robot's internal programming will require a method to calculate a 2D `row` and `column` array index given an x-y coordinate.

To do this a "mapping" is required which converts a Cartesian point, (x,y), to an array row, `r`, and column, `c` of a matrix with `M` rows and `N` columns.

Consider the sketch in Figure 2. It shows how a column index would map from a minimum value, `MIN`, at column zero, to a maximum value, `MAX`, at column `N`. When viewed from above, you can think of the `x` `MIN` and `MAX` values as the x coordinate of the left and right edges of the room, respectively.

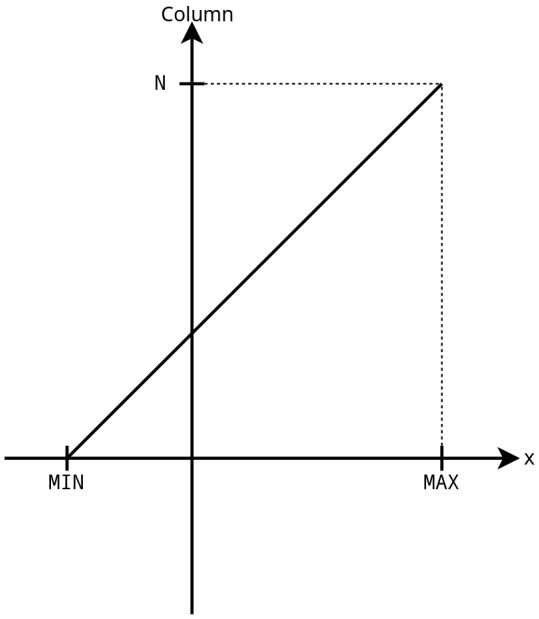

Figure 2: A linear mapping from an x coordinate to an array column index.

Task: Applying the 2-point formula:

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 \tag{1}$$

write an equation which converts a Cartesian x coordinate to a 2D array column index. To do this, observe that the "x" in the formula is the x coordinate of the robot and that "y" in the formula is the required column index. Take the point (x_1, y_1) to be the Figure 2 point $(\texttt{MIN}, 0)$ and (x_2, y_2) to be the point (`MAX`, `N`), and substitute them into the 2-point formula.

NB: When implementing this formula to index an array use the built in Python function `int()` to convert the column value to an integer.

Repeat this method to calculate a row number given the top and bottom y-coordinate extremes of an image. Create your own sketch then substitude into the 2-point formula. Note that the top (ie: maximum value of y) now maps to row zero and the bottom (minimum value of y) to row `M`.

[a] This is a basic form of "occupancy mapping".

Task 5: Barnsley Fern

In this task you will modify an existing Python program to generate an image of the *Barnsley fern* fractal.

A *fractal* is a mathematically generated image which exhibits "self-similar" geometry. As the image is zoomed in the the same patterns are seen repeated and, in theory, the image can be zoomed in forever and still show the same level of detail as it did when zoomed out.

The Barnsley fern is from a class of fractals known as iterated function systems (IFS). The general pattern for generating fractals of this type is to:

1. Pick (or be given) a point x_0, y_0
2. Generate a new point, x_1, y_1, by applying some mathematical rules
3. Draw a dot on an x-y plane where the new point lies
4. Repeat millions (or billions) of times until an image is drawn

The rules for the Barnsley fern are as follows:

- There are four functions which generate a new point, (x_{n+1}, y_{n+1}), from an old point (x_n, y_x):
  - f_1 :

    $$x_{n+1} = 0$$
    $$y_{n+1} = 0.16y_n$$

  - f_2 :

    $$x_{n+1} = 0.85x_n + 0.04y_n$$
    $$y_{n+1} = -0.04x_n + 0.85y_n + 1.6$$

  - f_3 :

    $$x_{n+1} = 0.2x_n - 0.26y_n$$
    $$y_{n+1} = 0.23x_n + 0.22y_n + 1.6$$

  - f_4 :

    $$x_{n+1} = -0.15x_n + 0.28y_n$$
    $$y_{n+1} = 0.26x_n + 0.24y_n + 0.44$$

- Each iteration, *one* of the four functions is chosen at random with a probability, p, of:
  - f_1: $p = 0.01$
  - f_2: $p = 0.85$
  - f_3: $p = 0.07$
  - f_4: $p = 0.07$

Task: Open the Barnsley Fern Wikipedia page: `https://en.wikipedia.org/wiki/Barnsley_fern`

Navigate to the "Syntax examples" section, copy the Python script (the first one) and run it (after a `pip install turtle`). Note that execution is *very* slow, about 10 points per second.

Using your results from the previous task (or the solution at the end of this document), modify this code to draw directly to an `MxN` array instead of plotting with the `turtle` library. You will need to create your own array with `np.zeros()` then index it as-per the formulas derived earlier.

Use the following values for the edges of the image:

- Left edge: $x = -2.5$

- Right edge: $x = 2.7$
- Top edge: $y = 10.0$
- Bottom edge: $y = 0$

Start with a small image (say 200x200 pixels) and a small iteration count (say, 10 000) so that debugging can be performed in a reasonable time.

Initialise the drawing to all zeros and assign a value of 1 to every pixel the algorithm "visits". Plot the final image with `plt.imshow()`.

Extension: Instead of assigning 1 to each visited pixel *increment* the value by 1. When plotting, use a plot command which specifies the minimum and maximum values so that the brightness and contrast are scaled correctly. eg, if the image is called `im_gray`:
`plt.imshow(im_gray,cmap='gray', vmin = 0, vmax = im_gray.max())`

Coordinate to Array Index Mapping Solution

Assuming an array with `M` rows and `N` columns:

$$\texttt{row} = \frac{-\texttt{M}}{\texttt{YMAX} - \texttt{YMIN}}(y - \texttt{YMIN}) + \texttt{M} \tag{2}$$

$$\texttt{column} = \frac{\texttt{N}}{\texttt{XMAX} - \texttt{XMIN}}(x - \texttt{XMIN}) \tag{3}$$

These equations assume the following interpretations for the image edges:

- Top: `YMAX`
- Bottom: `YMIN`
- Left: `XMIN`
- Right: `XMAX`

NB: Round the results with `int()`!