ENGG1003 - Lab Week 8

Brenton Schulz

Task 1: 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 indexing, draw a 1-pixel black square along the border (edge).

Task 2: Indexing 2D Arrays

Using four `for` or `while` loops, draw a 1-pixel black 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.

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

Task 3: Mapping Cartesian Coordinates to 2D Array Indices

Many problems require an image which covers part of an x-y plane to be "projected" (drawn) onto it. 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 1. It shows how a column index would map from a minimum value, `MIN`, at column zero, to a maximum value, `MAX`, at column `N`. You can think of the `x`-`MIN` and −`MAX` values as the x coordinate of the left and right edges of the image, respectively.

NB: The sketch shows a "smooth" line but in reality array indices are only integers; the real function ends up being a staircase.

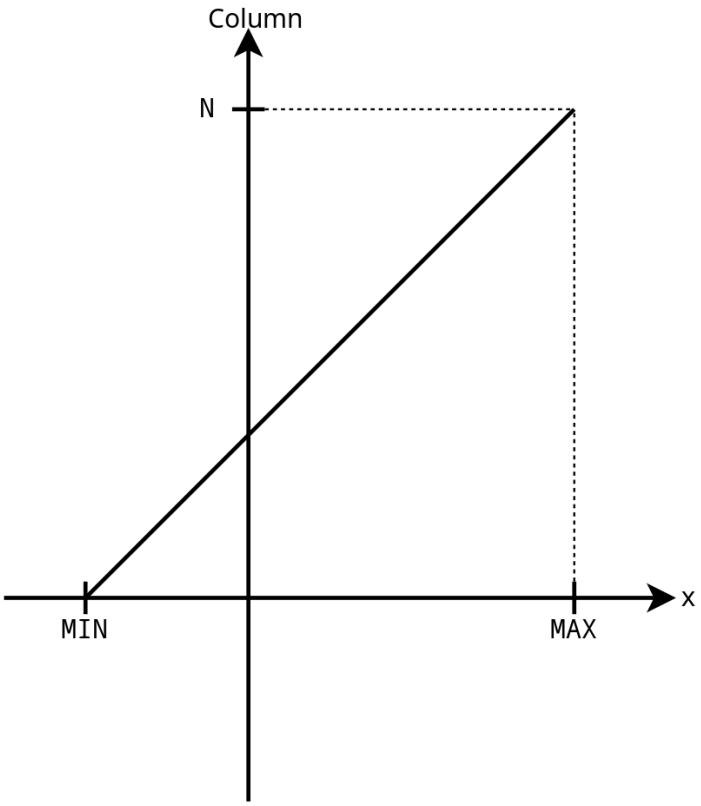

Figure 1: A linear mapping from an x or y coordinate to an array row or column.

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, take the point (x_1, y_1) to be the Figure 1 point $(0, \texttt{MIN})$ and (x_2, y_2) to be the point (`MAX`, `N`), and substitute them into the 2-point formula.

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`.

Task 4: Barnsley Fern

In this task you will modify an existing Python program to generate an image file 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, $(x_{n+1}, yn + 1)$, 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, modify this code to draw directly to an `MxN` array instead of plotting with the `turtle` library.

To do this, 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".

Extension: Instead of simply 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())
```